PITÁGORAS
FACULDADE

GEANNE MAYNARA AGUIAR SANTOS

USABILIDADE DE PADRÕES DE PROJETO NO PROCESSO DE DESENVOLVIMENTO DE APLICAÇÕES QUE UTILIZAM TECNOLOGIA JEE

São Luís
2017

GEANNE MAYNARA AGUIAR SANTOS

USABILIDADE DE PADRÕES DE PROJETO NO PROCESSO DE DESENVOLVIMENTO DE APLICAÇÕES QUE UTILIZAM TECNOLOGIA JEE

Trabalho de Conclusão de Curso apresentado à Faculdade Pitágoras, como requisito parcial para a obtenção do título de graduado em Ciência da Computação.

Orientador: Prof. Edilson Carlos Silva Lima

São Luís

2017

Santos, Geanne Maynara Aguiar.

Usabilidade de padrões de projeto no processo de desenvolvimento de aplicações que utilizam tecnologia jee. / Geanne Maynara Aguiar Santos. – São Luís, 2017.

69 f.
Orientador: Prof. Edilson Carlos Silva Lima
Monografia (Graduação em Bacharel em Ciência da Computação) – Faculdade Pitágoras de São Luís, 2017.

1. Padrões de Projeto. 2. Java EE. 3. Programação orientada a objetos. I. Título.

CDU 004.43

USABILIDADE DE PADRÕES DE PROJETO NO PROCESSO DE DESENVOLVIMENTO DE APLICAÇÕES QUE UTILIZAM TECNOLOGIA JEE

Trabalho de Conclusão de Curso apresentado à Faculdade Pitágoras, como requisito parcial para a obtenção do título de graduado em Ciência da Computação.

Aprovado em: __/__/____

BANCA EXAMINADORA

Prof(ª). Titulação Nome do Professor(a)

Prof(ª). Titulação Nome do Professor(a)

Prof(ª). Titulação Nome do Professor(a)

Dedico este trabalho à minha família, por tudo que são.

AGRADECIMENTOS

Agradeço primeiramente a Deus, que se mostrou criador, que foi criativo. Seu fôlego de vida em mim me foi sustento e me deu coragem para questionar realidades e propor sempre um novo mundo de possibilidades.

Agradeço aos meus pais, principalmente a minha mãe Marinalva e minha irmã Máylla que com amor incondicional, apoio e carinho me incentivaram e não mediram esforços para que eu chegasse até esta importante etapa de minha vida.

Aos meus mestres e colaboradores da Faculdade Pitágoras, que foram importantes na minha vida acadêmica e, desempenharam com dedicação e paciência as aulas ministradas.

Meus agradecimentos aos meus colegas de classe e com certeza excelentes futuros profissionais, por confiarem em mim e estarem do meu lado em todos os momentos em que fomos estudiosos, brincalhões e cúmplices. Obrigada pela paciência, pelo sorriso, pelo abraço, pela mão que sempre se estendia quando eu precisava. Esta caminhada não seria a mesma sem vocês.

Obrigada a todos, que mesmo não estando citados aqui, contribuíram de forma direta ou indiretamente para a minha formação acadêmica.

“Alguns obstáculos são feitos para darem sentido à vida. ”

(Mark Hr.)

RESUMO

O processo de construção de software pode se tornar uma atividade complexa e dispendiosa, o emprego de padrões de projetos torna mais simples a atividade de reutilização de projetos e arquiteturas. Com a utilização da linguagem Java e da tecnologia JEE os padrões de projetos veem se tornando cada vez mais populares para a construção de aplicações web**.** O estudo tem como objetivo geral**:** avaliar o emprego de padrões de projeto no processo de desenvolvimento de aplicações que empregam tecnologia JEE. A metodologia da pesquisa trata-se de uma revisão literária, narrativa, baseado em pesquisa bibliográfica a partir da compilação de trabalhos publicados em revistas científicas, livros especializados e em bases de dados da rede CAPES e IEE Xplore Digital Library. Como resultado do estudo, observou-se as funcionalidades dos principais e mais utilizados padrões catalogados para a construção de software e, notou-se que estes são de fácil adoção, entretanto, se utilizados de maneira excessiva acarretam em complicações no projeto, mas se utilizados de forma correta, se tornam essenciais para a solução de problemas recorrentes na prática de desenvolvimento. Conclui-se que é indispensável o emprego de métodos e técnicas da orientação a objetos por meio de uma linguagem padronizada e os padrões são aplicados nessa perspectiva, que cada vez mais são considerados como ferramentas essenciais para o projeto reutilizável de software.

Palavras-chave: Padrões de Projeto; Java EE; Programação orientada a objetos.

ABSTRACT

The software construction process can become a complex and expensive activity, the use of design patterns simplifies the reuse activity of projects and architectures. With the use of Java language and JEE technology, the standards of projects are becoming increasingly popular for the construction of web applications. The study has the general objective of evaluating the use of design patterns in the process of developing applications that employ JEE technology. The research methodology is a literary review, based on bibliographic research based on the compilation of papers published in scientific journals, specialized books and databases of the CAPES network and IEE Xplore Digital Library. As a result of the study, it was observed the functionalities of the main and most used standards cataloged for the construction of software and, it was noted that these are easy to adopt, however, if used in an excessive way entail complications in the design, but if used in the right way, become essential for solving recurring problems in development practice. It is concluded that the use of object-orientation methods and techniques is indispensable through a standardized language and the standards are applied in this perspective, which are increasingly considered as essential tools for reusable software design.

Keywords: Design Patterns; Java EE; Object-oriented programming.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CRUD	Create, Read, Update e Delete
DAO	Data Access Object
DTO	Data Transfer Object
EJB	Enterprise Java Beans
GoF	Gang of Four
HTML	HyperText Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
J2EE	Java 2, Enterprise Edition
Java SE	Java 2 Standard Edition
JDBC	Java Database Connectivity
JEE	Java Enterprise Edition
JMS	Java Message Service
JSP	Java Server Pages
JVM	Java Virtual Machine
MVC	Model Controller View
POJO	Plain Old Java Object
PP	Padrão de Projetos
RESTful	Representational State Transfer
SOAP	Simple Object Access Protocol
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
XHTML	eXtensible Hypertext Markup Language
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

O processo de construção de software pode se tornar uma atividade complexa e dispendiosa. Com o passar dos anos, processos de engenharia de software têm buscado reduzir o custo, padronizar o desenvolvimento de software, melhorar a qualidade, a confiabilidade e reduzir a complexidade no gerenciamento de processos.

Contudo, a tarefa de projetar um sistema orientado a objetos pode ser bem árdua, porém projetar software reutilizável orientado a objetos é ainda mais difícil. O emprego de padrões de projetos torna mais simples a atividade de reutilização de projetos e arquiteturas. E ajudam também na escolha alternativas de projetos que tornam um sistema reutilizável e impedem alternativas que afetam a reutilização.

Paralelo a esta perspectiva está o crescimento acelerado das aplicações que empregam a tecnologia Java EE, fornecendo serviços mais rápidos e instantâneos, por conta disso, as aplicações estão vendo se tornando mais ágeis e complexas relativo as suas novas características de negócios. Levando em consideração este fator, é notável a presença de problemas nestas aplicações principalmente com relação a incorreta estruturação do sistema e a escassez de conhecimento da tecnologia adotada.

Com a expansão da utilização da linguagem Java e principalmente da tecnologia JEE os padrões de projetos veem se tornando cada vez mais populares, uma vez que seja indispensável o seu uso adequado de padrões para construir aplicações web de grande porte.

Por se tratar de uma abordagem muito popular no âmbito de desenvolvimento de sistemas por projetistas de sistemas de softwares mais experientes e pouco versado âmbito acadêmico, espera-se que o desenvolvimento deste estudo irá possibilitar maior conhecimento acerca do tema, ressaltando as suas necessidades mais comuns de emprego.

O emprego de padrões de projetos pode aprimorar a documentação e a manutenção de sistemas ao oferecer uma especificação explicita de interações de classes e objetos e o seu objetivo implícito. De outro modo, ajudam o projetista a obter o projeto adequado rapidamente. Diante o exposto, sentiu-se a necessidade de investigar problemática em questão, qual a facilidade de emprego de padrões de projetos JEE no desenvolvimento de aplicações web?

De acordo com os argumentos supracitados, o presente trabalho trata-se de uma revisão literária, já que visa avaliar o nível de praticidade no processo de desenvolvimento de aplicações web fazendo o uso de padrões de projetos JEE, onde estes se tornaram imprescindíveis para a construção de sistemas voltados a internet, web ou intranet.

O estudo tem como objetivo geral, avaliar o emprego de padrões de projeto no processo de desenvolvimento de aplicações que empregam tecnologia JEE. Os objetivos específicos são: estimar os métodos utilizados para a construção de sistemas; avaliar conceitos e aplicações dos principais padrões de projetos catalogados; investigar a maneira que padrões resolvem problemas de projeto; avaliar o processo de aplicação de padrões de projetos e suas estratégias.

Para melhor apresentação da temática acometida no trabalho, o mesmo foi organizado em capítulos e subcapítulos. O corpo do trabalho apresenta-se estruturado da seguinte maneira: primeiro capítulo referente a introdução, no qual estão expostos os seguintes temas: reutilização de software e emprego de padrões de projetos JEE na construção de aplicações web, bem como o objetivo geral e específicos e a relevância do estudo.

O segundo capítulo apresenta o referencial teórico, no qual foi cometida a fundamentação teórica com o desígnio de transmitir a compreensão acerca do assunto, logo esta foi dividida nas seguintes temáticas: o surgimento de padrões de projeto, conceitos de padrões de projetos, componentes de um padrão, anti-padrões, classificação dos padrões, o surgimento, a plataforma, a arquitetura e os padrões de projeto Java EE.

Posterior a isso, demonstra-se no terceiro capitulo a metodologia empregada para utilização de padrões de projeto no desenvolvimento de aplicações, no qual se encontra delineado o estudo aprofundado acerca dos padrões para a tecnologia JEE, com a compilação dos mais utilizados padrões de projetos.

No quarto capítulo, apresenta-se uma discussão dos padrões abordados anteriormente complementado com a apresentação de sua aplicação e funcionamento, além de uma análise de boas e más práticas com o uso de padrões. No quinto, as considerações finais, desataca os padrões mais relevantes e as sugestões.

2 REFERENCIAL TEÓRICO

2.1 O SURGIMENTO DE PADRÕES DE PROJETOS

O termo padrão de projeto (PP) foi mencionado inicialmente na engenharia civil no qual consistia em delinear um problema para um ambiente e reutilizar sua solução. Porém, a definição de PP foi originalizada no final da década de 1970 pelo arquiteto e matemático Christopher Alexander no qual publicou diversos trabalhos na área de padrões e linguagens de padrões que exemplificam e descrevem seu método para a documentação de padrões. Apesar de seus trabalhos estarem voltados à arquitetura, possuem fundamentos que podem ser extraídos para diversas áreas de conhecimento. (MALDONADO *et. al.,* 2017).

Coplien (1996) ressalta que as ideias de Alexander influenciaram diretamente a comunidade de software que fez as vendas de seus livros alcançaram altos patamares neste meio. O próprio vocabulário empregado, como o termo linguagem de padrões *(language pattern*) vem das definições deste autor. Entretanto, atualmente uma boa parte da comunidade de software busca não aplicar intepretações literais do trabalho de Alexander.

Algumas atividades práticas no desenvolvimento de software, como desenvolvimento iterativo e incremental, assemelham-se ao pensamento de Alexander, que empregava padrões de maneira análoga. Porém, as semelhanças estabelecidas não eram verdadeiras em todos os casos. De qualquer maneira, mesmo que não se possa aplicar os conceitos desenvolvidos por Alexander em todos os casos de forma literal no desenvolvimento de software, seu legado é de suma importância: sua visão e seus sistemas de valores (HARTMANN, 2005).

Segundo Appleton (1997), os padrões começaram a ser utilizados no processo de construção de software em 1987, quando Ward Cunnignhan e Kent Beck estavam trabalhando em uma interface gráfica com o usuário implementada fazendo o uso da linguagem de programação *Smalltalk*. E decidiram então, utilizar os conceitos desenvolvidos por Alexander para desenvolver uma pequena linguagem de padrões com o objetivo de guiar novos programadores. Os resultados obtidos foram publicados no artigo denominado “Utilizando linguagem de padrões para programas orientado a objetos”.

Ao final da década de 1990, Coplien (1991) compilou um catálogo de expressões idiomáticas em C++ e publicou um livro intitulado “Expressões idiomáticas e estilos de programação avançada em C++”. Para mais, na mesma da década, vários workshops foram realizados nos Estados Unidos que foram discutidos e compilados inúmeros catálogos de padrões. Em 1995, foi publicado o livro de catálogo de padrões de projetos (*GoF*) e partir daí se popularizou o emprego de padrões de projetos na comunidade de desenvolvimento de software.

Esses padrões (GoF) adotados por Gamma *et. al.* (2000) que produziram um alicerce para a maior parte dos padrões relacionados a problemas de projetos dessa natureza, como por exemplo o catálogo *Core J2EE Patterns* (ANEXO A), conhecido também como *Blueprints*.

2.2 CONCEITOS DE PADRÕES DE PROJETOS

Em uma tradução livre de seu livro, Alexander *et. al.* (1977, p. 10), apresentam: “cada padrão descreve um problema que acontece no nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira duas vezes”.

Embora seu discernimento estivesse voltado aos padrões de construções e cidades, o mesmo é valido para os padrões de projetos orientado a objetos. Cassimiro (2010) alega que, na computação, um PP descreve uma solução em um ambiente como resposta a um problema. Porém, os padrões são definidos na computação por uma comunidade que examina dissoluções análogas propostas por vários arquitetos para os mesmos problemas.

Contudo, Alexander (1979) descreve que cada padrão é uma regra de três partes, que manifesta uma semelhança entre um contexto, um problema e uma solução. Por outra forma, um padrão é algo que acontece no mundo, e ao mesmo tempo, uma regra que descreve como criar este algo e quando deve ser criado.

De acordo com Maldonado *et. al.* (2017), cada padrão delineia um problema decorrente inúmeras vezes em nosso âmbito e então detalha a solução reutilizável para o problema de tal forma que não seja necessário buscar a solução mais de uma vez, apesar de seu conceito também está relacionado à arquitetura, possui argumentos básicos que podem ser extraídos para a área de software.

Alexander *et. al.* (1977) explicam que um padrão é uma relação entre as forças que se mantém recorrentes em um contexto específico a uma configuração que resolve estas forças. Além disso, um padrão também é uma regra que explica como criar a configuração particular que resolve as forças dentro desse contexto.

Esta definição de "padrão como regra" tem sido altamente influente. GAMMA *et. al.* (2000), definem um padrão para ser a solução para um problema recorrente em um contexto específico, aplicável não só à arquitetura, mas também para o design de software.

Um conceito singular é dado por Coad (1992), que originalmente utiliza um dicionário, definindo um padrão para ser “uma forma original, ou modelo [...] de imitação” (Coad, 1992, p. 2). A noção de padrão para modelos de objetos como "um modelo de objetos com responsabilidades e interações estereotipadas" (COAD; YOURDON, 1995, p. 23).

Buschmann *et. al.* (1996), afirmam que quanto maior o a quantidade de padrões em um sistema de padrões, maior será a dificuldade de compreensão e utilização destes. Entretanto, é comum adotar um esquema de organização que permite agrupar os padrões, reduzindo assim o estímulo dos desenvolvedores em encontrar os mais apropriados para um determinado problema.

Os padrões concretos são expressos usando classes e objetos, porém representam soluções para problemas em contextos particulares. Esta é a noção mais amplamente difundida do padrão hoje, e tem sido adotado por muitos outros pesquisadores. Os PP propiciam a reusabilidade de projetos e arquiteturas evitando seu comprometimento, além de auxiliar na manutenção e documentação de projetos. (RIEHLE; ZÜLLIGHOVEN, 1996).

Lupianhez e Lucrédio (2012) afirmam que padrões tem a finalidade de propor e catalogar problemas relacionados na criação de projetos de software de orientado a objetos, baseado em soluções uma vez utilizadas com sucesso em projetos.

Contudo, pode-se perceber que existem inúmeras definições para o termo padrão, porém todas elas possuem um argumento comum relacionado à recorrência de um par problemas/soluções em um contexto específico (ALUR; CRUPI; MALKS, 2003).

Entretanto, um padrão não se refere apenas a uma especificação detalhada, mas a explanação de um problema com o acréscimo da experiência para

a dissolução deste problema, de forma abstrata, para que possa ser empregado como solução para problemas análogos (SOMMERVILLE 2007).

Os padrões ajudam a criar uma linguagem compartilhada para comunicar percepção e experiência sobre os problemas e suas soluções. De natureza igual, a codificação dessas soluções e suas relações permite capturar com sucesso o conhecimento que domina a compreensão de um bom design de arquiteturas que atendam às necessidades de seus usuários (JAMAL, 2003).

De acordo com Filho (2003), no estudo da engenharia de software, um processo possui seções que tem o objetivo de avaliar o desempenho do projeto e corrigir seus contextos ao surgir problemas que possam ser bem definidos. Diante disso, uma das alternativas empregadas para alcançar esses objetivos é o uso de padrões de projeto.

2.3 COMPONENTES DE UM PADRÃO

Embora os padrões de software se encontrar demarcados em categorias distintas, a representação dos mesmos segue uma linha comum no qual são definidos os elementos principais que identificam o padrão. Cada componente pode ser descrito da seguinte maneira:

O nome *(name)* deve possuir o nome mais significativo possível pois permite fazer referência ao conhecimento e estrutura que se descreve fazendo o uso de uma frase curta ou em uma única palavra. É importante definir bons nomes de padrões pois eles formam um vocabulário para discutir abstrações conceituais. Um padrão pode apresentar mais de um nome comumente utilizado ou reconhecível na literatura, portanto, é comum documentar esses apelidos ou sinônimos sob o título de aliasses ou conhecido como algumas formas padrão também fornece uma classificação do padrão, além de seu nome (APPLETON,1997).

O problema (*problem*) refere-se a uma declaração do problema a ser resolvido pelo padrão, apresentando a meta e objetivo do mesmo. Em alguns casos, o problema conterá uma série de premissas que devem ser realizadas para fazer sentido na aplicação do padrão (BROWN *et. al.,* 1998).

O contexto (*context)* define as premissas em que o problema e sua solução aparentam recorrer, e para quais problemas a solução é aceitável voltado a

aplicabilidade do padrão. Pode ser pensado como a configuração inicial do sistema antes que o padrão seja aplicado a ele (ALUR; CRUPI; MALKS, 2003).

A força (*force)* descreve uma relação das forças e restrições relevantes e a suas formas de interações entre si e com os objetivos desejados. As forças revelam as complexidades de um problema e definem os tipos de vantagens e desvantagens *(trade-offs)* que devem ser consideradas. Uma boa definição do padrão deve encapsular totalmente todas as forças que têm um impacto sobre ele (APPLETON, 1997).

A solução (*solution)* remete as relações estáticas e regras dinâmicas de como chegar ao resultado desejado. A solução deve descrever não apenas a estrutura estática, mas também o comportamento dinâmico. A descrição da solução do padrão pode indicar orientações pensadas e evitadas ao tentar uma implementação concreta da solução (FOWLER, 2006).

Os exemplos *(examples)* podem ser uma ou mais aplicações de exemplo do padrão que ilustram. Exemplos auxiliam o leitor a compreender o uso e a aplicabilidade do padrão. Os exemplos e analogias visuais podem ser especialmente esclarecedores. Um exemplo pode ser complementado por uma implementação de exemplo para mostrar uma maneira a solução pode ser realizada (GAMMA *et. al.*, 2000).

O contexto resultante (*resulting context)* define o estado ou configuração do sistema após a aplicação do padrão, incluindo as consequências (boas e ruins) da aplicação, e outros problemas e padrões que podem nascer a partir do novo contexto. Descreve as pós-condições e os efeitos colaterais do padrão, que é chamado de resolução de forças, pois define as forças que foram resolvidas, as não resolvidas e quais padrões podem ser aplicáveis (ALUR; CRUPI; MALKS, 2003).

A justificativa (*rationale)* é uma explicação justificativa de etapas ou regras no padrão, e também do padrão como um todo, no sentido de como e por que ele resolve as forças de uma maneira particular para atender os objetivos desejados. Ele descreve como o padrão funciona, o motivo de funcionar e o porquê de seu emprego ser adequado (BROWN *et. al.,* 1998).

Os padrões relacionados *(related patterns)* delineia as relações estáticas e dinâmicas entre este padrão e outros dentro da mesma linguagem de padrão ou sistema. Padrões relacionados frequentemente têm um contexto inicial ou resultante

que é compatível com o contexto resultante ou inicial de outro padrão (APPLETON, 1997).

Os usos conhecidos *(known uses)* apresenta casos conhecidos do padrão e sua aplicação dentro de sistemas existentes. Essa prática ajuda a validar um padrão, verificando que ele é realmente uma solução comprovada para um problema recorrente (COAD, 1992).

Gamma *et. al.* (2000) asseguram que mesmo não se fazendo obrigatoriamente necessário, o emprego de padrões inicia com uma abstração que oferece uma percepção geral. Com isso, ocorre uma boa visão do padrão, declarando rapidamente sua relevância para qualquer problema que queira ser resolvido no âmbito de desenvolvimento.

2.4 ANTI-PADRÕES

Um anti-padrão, ou (*antipattern*) é caracterizado como uma consequência da ausência de conhecimento de uma solução correta, ou da aplicação de um padrão no contexto incorreto. Anti-padrões tem como objetivo resolver soluções incorretas de problema que causa uma situação ruim, apresentando assim, uma solução uma eficaz (KOENIG, 1995).

Os anti-padrões representam uma “lição aprendida” diferente de padrões que definem “lições práticas”. Diante dessa abordagem, Appleton (1997) descreve a existência de dois tipos de padrão: a) aqueles que descrevem uma má solução para um problema que resultou em uma situação ruim, e b) aqueles que descrevem como sair de uma situação ruim e como proceder, a partir dela, para uma boa solução.

Anti-padrões são fundamentais pelo fato de ser essencial visualizar e compreender soluções ruins como sendo boas. Já que é útil apresentar a presença de padrões em sistemas malsucedidos, se faz útil também apresentar a ausência dos anti-padrões em sistemas bem-sucedidos (MALDONADO *et. al.*, 2017).

A presença de padrões "bons" em um sistema bem-sucedido não é suficiente. Deve-se mostrar que esses padrões estão ausentes em sistemas malsucedidos. De modo igual, é útil mostrar a presença de certos padrões (antipatterns) em sistemas infrutíferos, e sua ausência em sistemas bem-sucedidos (COPLIEN, 1995).

Nesse contexto, Brown *et. al.* (1998) afirmam que um padrão de projeto se torna um anti-padrão quando ele causa mais problema do que resolve. Ele também propõe uma estrutura para a descrição de anti-padrões, categorizadas por: nome do anti-padrão, sua forma geral, sintomas e consequências, variações, causas típicas, nova solução indicada, exceções conhecidas e que são relacionadas.

2.5 CLASSIFICAÇÃO DOS PADRÕES

Sabendo que padrões representam soluções para problemas recorrentes em um contexto, os mesmos foram capturados em muitos níveis de abstração em vários domínios. Inúmeras categorias foram sugeridas para a classificação de um padrão de software, sendo as mais comuns: padrões de projetos, padrão arquitetônicos, padrões de analise, padrões criacionais, padrões estruturais e comportamentais (PEREIRA, 2008).

O catálogo de padrões de projetos designados por Gamma *et. al.* (2000) são os mais conhecidos e utilizados. A sua classificação se deve a dois fatores, que são eles finalidade e escopo. O primeiro propósito, denominado finalidade, descreve o que o padrão faz. Os padrões podem ter o propósito de criação, estrutural ou comportamental.

Quadro 1: Organização dos padrões de projetos.

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method	Adapter (class)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Fonte: GAMMA *et.al.* (2000).

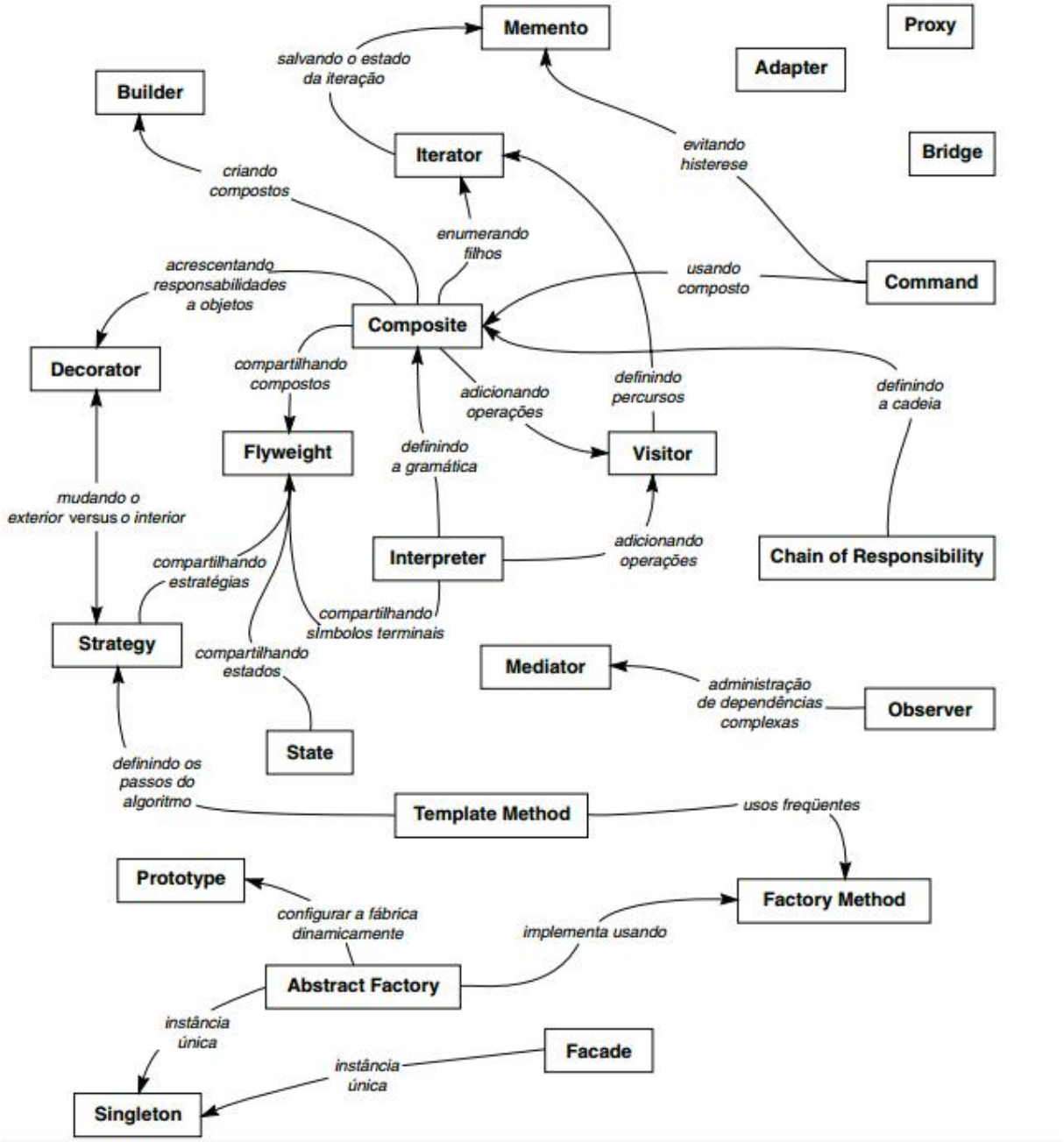
Os padrões de criação definidos por Joshi (2015) são usados para projetar a tarefa de instanciação de objetos, fazendo um sistema independente da forma de como seus objetos são criados, compostos e representados. Os padrões estruturais descritos por Gamma *et. al.* (2000), dizem respeito à forma como as classes e os objetos são compostos. Tem a possibilidade de relacionar, em estruturas complexas, objetos, ou definir a maneira da composição e herança das classes a partir de outras. E de acordo com Matos (2015), os padrões comportamentais estão relacionados com algoritmos e a atribuição de responsabilidades entre objetos. Estes descrevem não apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles.

O segundo proposito, denominado escopo, aponta se o padrão se aplica primariamente a classe ou objetos. Os padrões para classes lidam com os relacionamentos entre classes e subclasses por meio da herança e em tempo de compilação, tornando-os assim estáticos. Os padrões para objetos lidam com relacionamentos entre objetos que podem ser alterados em tempo de execução sendo mais dinâmicos. Basicamente, todos fazem o uso da herança em certa medida. (ALBUQUERQUE; ROJAS; RIBEIRO, 2010).

Os padrões criacionais direcionados para classes transmitem para suas subclasses algum passo da criação de objetos, enquanto que os padrões criacionais voltados para objetos delegam esse processo para outro objeto. Entretanto, os padrões estruturais direcionados para classes utilizam a herança para compor classes, enquanto que os padrões estruturais direcionados para objetos definem maneiras de montar objetos (GAMMA *et. al.*, 2000).

Os padrões comportamentais estão associados especificamente com a relação entre objetos. De acordo com Maldonado *et. al.* (2017), os padrões comportamentais voltados para classes utilizam a herança para descrever a distribuição de comportamento, enquanto que os voltados para objetos compõem um grupo de objetos para descrever a comunicação de uma tarefa que um único objeto não pode executar sozinho.

Figura 1: Representação do relacionamento entre padrões de projetos.



Fonte: GAMMA *et.al.* (2000).

Entretanto, Gamma *et. al.* (2000) afirmam que existem diversas formas de organizar os padrões. Alguns deles são utilizados em conjuntos frequentemente. Como por exemplo, o *Composite* é frequentemente utilizado com o *Iterator* e o *Visitor.* É possível também, que padrões mesmo que possuam intenções distintas resultem em projetos análogos. Como por exemplo, os diagramas estruturais *Composite* e *Decorator* são semelhantes.

2.6 O SURGIMENTO DO JAVA EE

A Sun Microsystems em 2002 formou uma equipe intitulada *Green Team* com a finalidade de desenvolver inovações tecnológicas. Esta equipe criou um interpretador, que mais tarde seria a Máquina Virtual Java (JVM), afim de facilitar o desenvolvimento de aplicações compatíveis em vários dispositivos, como por exemplo, aparelhos médicos, televisões, aparelhos de TV a cabo, dentre outros. Porém, o projeto não foi bem-sucedido por questões de custo e conflito de interesses (LE ROY JUNIOR, 2014).

Com o advento da *Web* em 1995, a Sun notou a oportunidade de desenvolver aplicações compatíveis em vários sistemas operacionais e navegadores (denominados *applets*), não somente renderizando HTML, mas também proporcionado aplicações do lado do cliente. Então, surgiu-se assim o Java 1.0. Mesmo sendo criado com essa finalidade, os *applets* se tornaram obsoletos e a tecnologia voltou seus objetivos para o lado do servidor (ALUR; CRUPI; MALKS, 2003).

Em 1997 é lançada a versão 1.1 do J2EE, que continha a primeira versão do componente da versão Enterprise, o EJB. Dois anos depois, é lançada a versão J2EE 1.2 (*Java 2 Entreprise Edition*) incluindo bibliotecas e APIs e seus principais objetivos eram aplicações distribuídas, tolerantes a falhas, multicamadas e baseadas em componentes reutilizáveis do lado do servidor. Contudo, a plataforma muda de nome em 2006 com o lançamento da versão 5, para Java EE (LE ROY JUNIOR, 2014).

Desde então, as aplicações de software tiveram de se adaptar a novas soluções técnicas como SOAP ou serviços web RESTful. A plataforma Java EE evoluiu para responder a essas necessidades técnicas, fornecendo várias formas de trabalhar através de especificações padrão. Ao longo dos anos, o Java EE mudou e se tornou mais rico, mais simples, mais fácil de usar, mais portátil e mais integrado (GONCALVES, 2013).

Entretanto, o JEE se tornou a plataforma mais popular voltada ao desenvolvimento, pois carrega um conjunto de recursos que o transformou em uma ótima escolha (YENER; THEEDOM, 2015).

Além disso, Le Roy Junior (2014) afirma que o surgimento de padrões e boas práticas é decorrente ao crescimento de aplicações JEE e a necessidade de

alcançar cada vez mais requisitos não-funcionais que estão presentes em grandes aplicações web.

2.7 A PLATAFORMA JAVA EE

Java EE é um acrônimo para *Java Enterprise Edition*, e define uma plataforma padrão voltada para a construção de aplicações Java para internet ou de médio e grande porte, contendo bibliotecas e funcionalidades para implementar software Java distribuído, tendo componentes modulares como base que executam em servidores de aplicação e suportam segurança, escalabilidade, integridade, e outros requisitos de aplicações corporativas (FARIA, 2013).

Contudo, Guo (2004) descreve que a plataforma Java EE define um padrão simples que se aplica a todos os aspectos de arquitetar e desenvolver aplicativos baseados em servidor de várias camadas.

De acordo com Alur, Crupi e Malks (2003), desde o início da linguagem Java, a plataforma sofreu uma popularização e crescimento. Diversas outras tecnologias tornaram-se parte da plataforma Java e novas APIs e padrões foram desenvolvidas com a finalidade de solucionar problemas no desenvolvimento de software.

Segundo Jendrock *et. al.* (2014), a plataforma JEE utiliza um modelo de aplicativo distribuído para aplicativos corporativos. A lógica do aplicativo é dividida em componentes de acordo com a função e os componentes do aplicativo que compõem um aplicativo Java EE são instalados em várias máquinas, dependendo da camada no ambiente Java EE multicamada ao qual pertence o componente do aplicativo.

A plataforma JEE oferece inúmeros benefícios para a empresa, uma delas é a possibilidade de aplicações para acessar os serviços oferecidos de maneira independente de fornecedor. As empresas que utilizam esta plataforma possuem liberdade de escolha, podendo assim definir o fornecedor mais adequado para a situação (PÉRES JUNIOR, 2003).

Singh (2002) descreve que o Java EE utiliza um modelo de aplicações multicamadas. As tecnologias que compõem esta plataforma de desenvolvimento são os *Enterprise JavaBeans*, que descreve componentes que implementam regras de negócio e os *Servlets* e os *JavaServer Pages*, que são classificados como componentes *Web.*

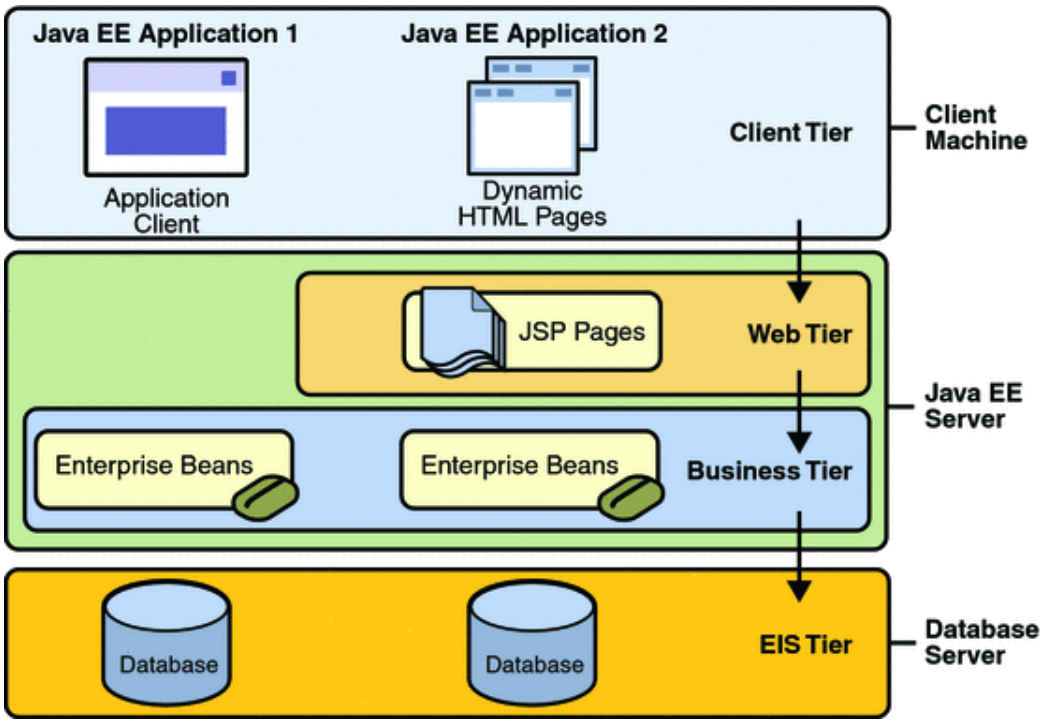
O objetivo principal da plataforma Java EE é simplificar o desenvolvimento fornecendo uma base comum para os vários tipos de componentes no JEE. Os desenvolvedores se beneficiam de melhorias de produtividade com mais anotações (*annotations*) e menos configuração XML, mais *Plain Old Java Objects* (POJOs) e pacotes (*packages*) simplificados (JENDROCK *et. al.* 2014).

2.8 ARQUITETURA JEE

O JEE é uma plataforma baseada em padrões para o desenvolvimento de aplicativos da Web e da empresa. Esses aplicativos são normalmente projetados como aplicativos multicamadas, contendo uma camada *front-end* consistindo de uma estrutura web, uma camada intermediária fornecendo segurança e transações e uma camada *back-end* que fornece conectividade para um banco de dados ou um sistema legado. Essas aplicações devem ser responsivas e capazes de dimensionamento para acomodar o crescimento da demanda do usuário (GUPTA, 2013).

Entretanto, Alur, Crupi e Malks (2003) classificam os padrões em cinco camada de arquitetura lógica: camada do cliente, camada de apresentação, camada de negócios, camada de integração e camada de recursos. Pois consideram que estas são camadas básicas existentes em uma aplicação Java.

Figura 2: Representação da arquitetura multicamadas de uma aplicação Java EE.



Fonte: Gupta (2017).

A camada cliente (*cliente tier*) descreve uma interface de entrada e saída com a finalidade de interagir com o sistema e é executada na máquina do cliente. Em aplicações web, esta camada é implementada com ajuda do Web Browser que serve para apresentar e interpretar todo o conteúdo gerado pela camada de apresentação (*web tier*) que normalmente são *Javascript* e HTML. Esta camada interage com a *web tier* fazendo o uso de protocolos como HTTP e HTPPS (AQUINO JUNIOR, 2002).

A camada de apresentação (*web tier)* é a primeira camada do servidor de aplicação e tem como principal função disponibilizar os serviços da *business tier* para o ambiente web, proporcionando conteúdo estático e dinâmico gerados pelo componentes *web.* Normalmente, o conteúdo gerado por esta camada é HTML, mas esta pode gerar qualquer outro formado como por exemplo JSP, XML e XHTML, uma vez compatível com o protocolo HTTP (LUPIANHEZ; LUCRÉDIO, 2012).

A camada de negócios *(business tier)* é referente ao centro do sistema, na qual são implementadas todas as regras de negócio da aplicação. Para a implementação desta camada, a plataforma JEE faz uso de *Enterprise Java Beans* (EJB), porém o modelo é bastante flexível capaz de suportar a tecnologia COBRA ou componentes implementados utilizando a API básica do Java SE (AQUINO JUNIOR, 2002).

A camada de integração *(intagration tier)* é responsável pela comunicação com recursos e sistemas externos, como armazenamento de dados. A camada de negócios é ligada a esta camada quando os objetos de negócio estabelecem dados ou serviços presentes na camada de recursos. Os componentes nessa camada podem fazer o uso de tecnologia de algum conector JDBC ou algum *middleware* único para trabalhar com a camada de recursos. (LUPIANHEZ; LUCRÉDIO, 2012).

A camada de recursos *(resource tier)* é onde contém dados de negócios e recursos externos, tais como *mainframes* e sistemas legados, sistemas de integração de comércio eletrônico entre empresas (B2B) e serviços como autorizações em transações de cartões de crédito, por exemplo (ALUR; CRUPI; MALKS, 2003).

Contudo, uma camada é equivalente a uma das divisões lógicas dos aspectos variados tratados em um sistema. Péres Júnior (2003) assegura que em cada camada é atribuída sua análoga ou ímpar responsabilidade no sistema. E que

cada camada é logicamente separada entre si, e não se encontra interligada especificamente com a camada contígua.

2.9 PADRÕES DE PROJETO JEE

Os padrões JEE define um conjunto de soluções com base o JEE para problemas comuns. Eles estão voltados aos conhecimentos e experiências coletivas adquiridas pelos arquitetos da Java Sun Center, obtidas com a execução e sucesso numerosos compromissos JEE (GONCALVES, 2013).

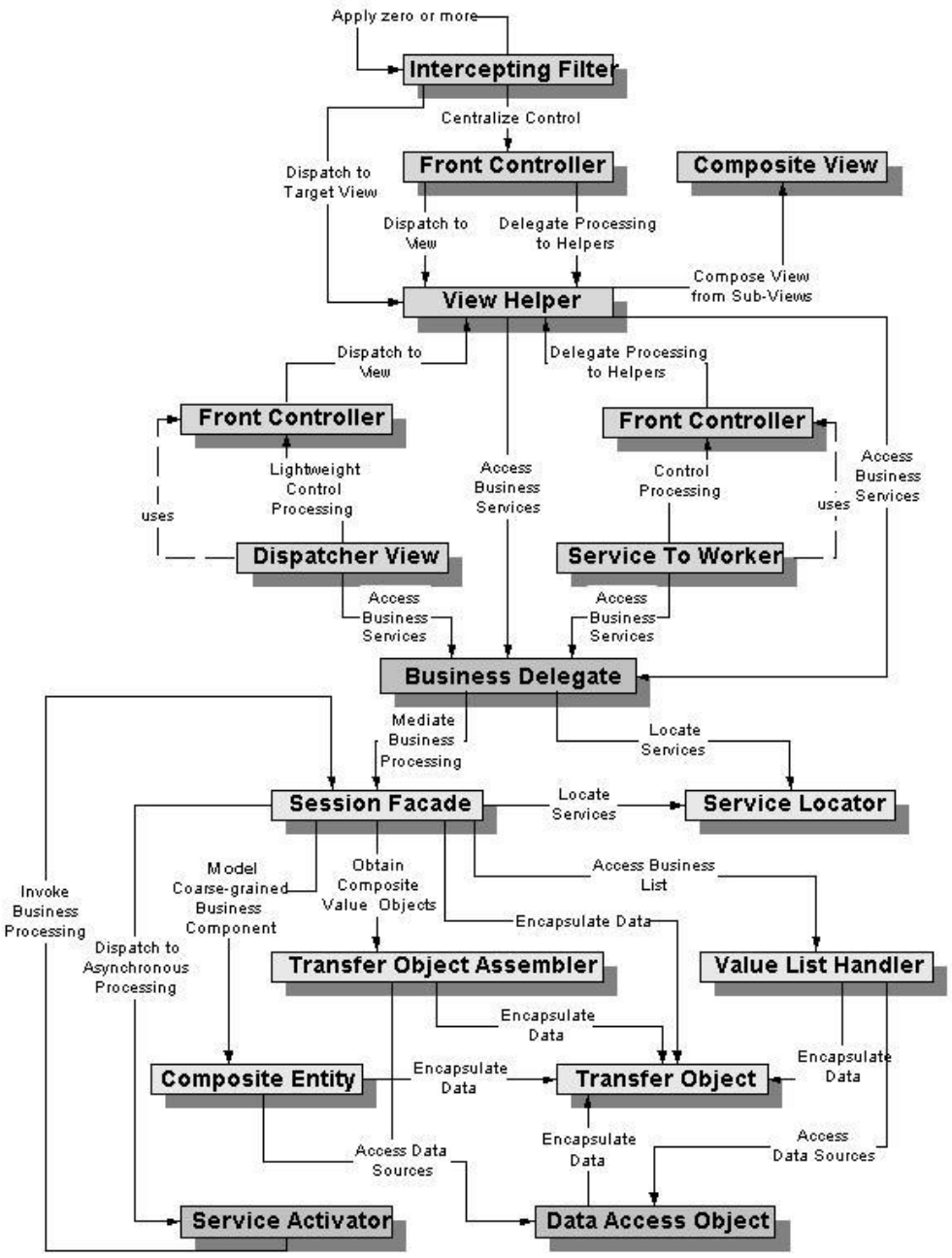
O Sun Java Center é a organização de consultoria da Sun, focada em arquitetar soluções baseadas em tecnologia Java para clientes. O Sun Java Center tem vindo a arquitetura soluções para a plataforma JEE desde os seus primeiros dias, concentrando-se na obtenção de qualidade de serviço (QoS) tais como escalabilidade, disponibilidade, desempenho, segurança, fiabilidade e flexibilidade (JENDROCK, 2014).

Os padrões claramente e simplesmente expressam técnicas comprovadas. Eles tornam mais fácil o reuso de projetos e arquiteturas. Alur, Crupi e Malks (2003) alegam que cada padrão está localizado entre um padrão de *design* e um padrão de arquitetura, mesmo que as estratégias documentem partes de cada padrão a um nível inferior de abstração.

Para Tanaka e Pansanato (2005), a única estrutura que é explicita refere-se em classificar cada padrão dentro de uma das três camadas lógicas de arquitetura existentes: apresentação, negociação e integração. Alur, Crupi e Malks (2003), apresentam que os padrões da camada de apresentação contêm os padrões relacionados aos *servlets* e à tecnologia JSP. Os padrões da camada de negócios contêm os padrões relacionados à tecnologia EJB. Os padrões de níveis de integração contêm os padrões relacionados a JMS e JDBC.

Cada padrão pode ser descrito de forma concisa, uma vez que seu nome é facilmente associativo a sua funcionalidade. Além disso, inúmeras são as vantagens de utilização de padrões de projeto JEE. Yener e Theedom (2015) destacam algumas dessas principais vantagens, que são: reuso, expressividade, facilidade de aprendizado e redução o retrabalho.

Figura 3: Relacionamento entre padrões Java EE.



Fonte: ALUR; CRUPS; MALKS (2003).

Cada padrão da camada de apresentação é descrito por Alur, Crups e Malks (2003) da seguinte maneira: *Intercepting Filter:* facilita o pré-processamento e pós-processamento de uma solicitação; *Front Controller:* fornece um controlador centralizado para gerenciar o manuseio de uma solicitação; *Context Object:* encapsula o estado em um modo independente de protocolo para ser compartilhado em toda a aplicação; *Application Controller:* centraliza e modula o gerenciamento de ação e

visualização; *View Helper:* encapsula a lógica que não esteja relacionada à formatação da apresentação em componentes auxiliares *(Helper); Composite View:* cria uma visualização *(View)* agregada a partir subcomponentes bem pequenos; *Service To Worker:* combina um componente distribuidor *(Dispatcher)* com os padrões *Front Controller* e *View Helper; Dispatcher View:* combina um componente distribuidor com os padrões *Front Controller* e *View Helper* transferindo muitas atividades para o processo de visualização.

Bien (2009) descreve cada padrão da camada de negócios da seguinte forma: *Business Delegate:* encapsula o acesso a um serviço de negócios; *Service Locator:* encapsula a pesquisa de serviços e componentes; *Session Façade:* encapsula os componentes da camada de negócios e exibe um serviço de granulação grossa para os clientes remotos; *Application Service:* centraliza e agrega o comportamento afim de oferecer uma camada uniforme de serviços; *Business Object:* usa um modelo de objetos para separar a lógica de negócios e os dados; *Composite Entity:* implementa um *business object* persistente empregando POJOs locais e *beans* de entidade; *Transfer Object:* transfere dados por meio de uma camada; *Transfer Object Assembler:* monta um objeto composto a partir de várias fontes de dados; *Value List Handler:* manipula a busca, armazena os resultados em cache e seleciona itens nos resultados.

E por fim, Broemmer (2003) descreve cada padrão da camada de integração da seguinte forma: *Data Acess Object:* abstrai e encapsula o acesso ao armazenamento persistente; *Service Activator:* recebe mensagens e chama o processamento de modo assíncrono; *Domain Store:* oferece um mecanismo transparente de persistência para os objetos de negócios; *Web Service Broker:* expõe um ou mais serviços usando protocolos *Web* e XML.

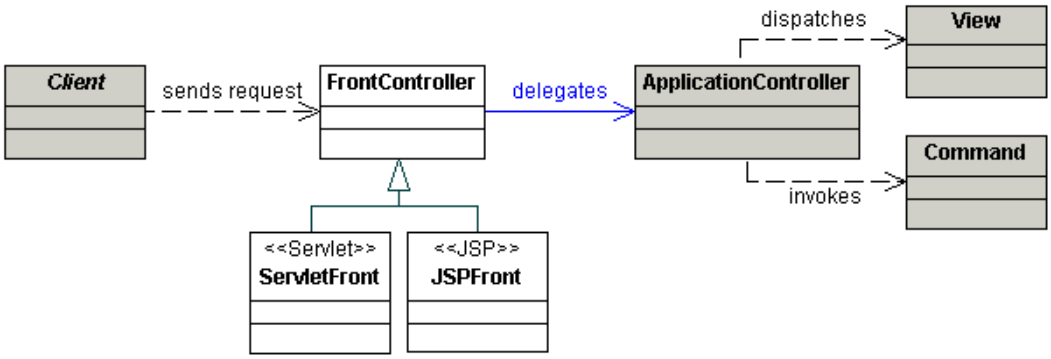
3 PRINCIPAIS PADRÕES DE PROJETO DO JAVA EE

3.1 FRONT CONTROLLER

O padrão *Front Controller* é um contêiner que armazena a lógica de processamento comum que ocorre dentro da camada de apresentação e que pode ser colocada erroneamente em uma *View*. Um controlador lida com solicitações e gerencia a recuperação de conteúdo, a segurança, o gerenciamento de visualizações e a navegação, delegando a um componente do *Dispatcher* para despachar um modo de exibição (ALUR; CRUPI; MALKS, 2003).

Para Fowler (2006), o padrão *Front Controller* é empregado com a finalidade de fornecer um mecanismo centralizado de manipulação de solicitação para que todas as solicitações sejam tratadas por um único manipulador. Esse manipulador pode fazer a autenticação, autorização, registro ou rastreamento de solicitação e, em seguida, passar os pedidos aos manipuladores correspondentes.

Figura 4: Diagrama de classe do padrão *Front Controller.*



Fonte: ALUR; CRUPS; MALKS (2003).

Martins (2012, p. 6) descreve o que o objetivo do *Front Controller* é: “centralizar o processamento de requisições em uma única fachada. O *Front Controller* permite criar uma interface genérica para processamento de comandos”. Entretanto, Basham, Sierra e Bates (2008), afirmam que o padrão *Front Controller* é utilizado para coletar código de processamento de solicitações comum, muitas vezes redundante, em um único componente. Isso permite que o controlador de aplicativo seja mais coeso e menos complexo.

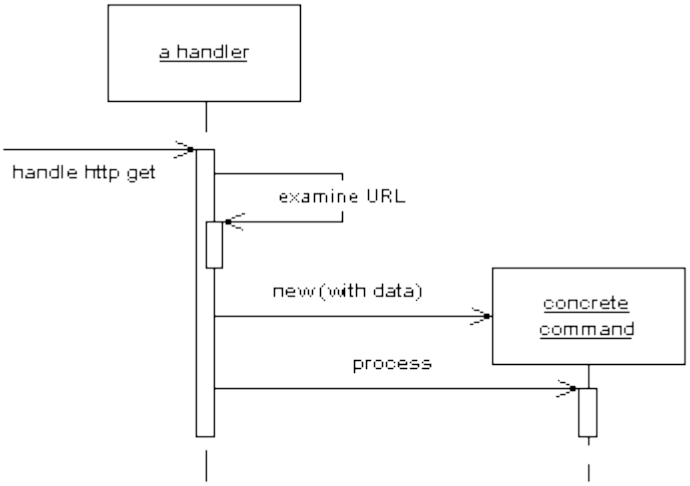
O *Front Controller é* responsável pelo roteamento de solicitações de usuários de entrada. Além disso, eles podem reforçar a navegação em aplicativos da web. Quando os usuários estão em seções de um aplicativo da Web onde eles podem navegar livremente, o controlador frontal simplesmente retransmite o pedido para a página apropriada (MAATTA *et. al.*, 2002).

Um *Front Controller* lida com todas as chamadas para um site da *web* e é normalmente estruturado em duas partes: um manipulador *web* e uma hierarquia de comandos. O manipulador *web* é o objeto que realmente recebe postagem ou obtém solicitações do servidor *web*. Ele captura apenas informações suficientes do URL e a solicitação para decidir que tipo de ação deve ser iniciada e, em seguida, delega-se a um comando para executar a ação (FOWLER, 2006).

Fowler (2006) destaca que uma variação interessante do *Front Controller* é o uso um manipulador web em dois estágios. Neste caso, o manipulador web é ainda separado em um manipulador web degenerado e um *dispatcher*. O manipulador web degenerado captura os dados básicos dos parâmetros HTTP e os entrega ao *dispatcher*, de tal forma que o *dispatcher* é completamente independente da estrutura do servidor web. Isso torna o teste mais fácil porque o código de teste pode direcionar o *dispatcher* diretamente sem ter que ser executado no servidor web.

Entretanto, vale ressaltar que tanto o *handle* quanto os *commands* fazem parte do controlador. Como resultado, os comandos podem (e devem) escolher qual exibição usar para a resposta. A única responsabilidade do manipulador está na escolha do comando a ser executado. Uma vez feito isso, ele não desempenha mais parte nesse pedido (FOWLER, 2006).

Figura 5: Representação do funcionamento do *Front Controller*.



Fonte: FOWLER (2006).

O manipulador *web* (*handler)* é quase sempre implementado como uma classe em vez de como uma página de servidor, uma vez que não produz qualquer resposta. Os comandos também são classes em vez de páginas de servidor e na verdade não é necessário conhecimento do ambiente da *web*, embora muitas vezes passam as informações para o HTTP. O manipulador *web* em si é geralmente um programa bastante simples que não faz nada além de decidir o comando a ser executado (MUELLER, 2004)

O manipulador web pode tomar a decisão de qual comando executar estática ou dinamicamente. A estática envolve analisar o URL e usar lógica condicional para decidir qual comando executar. E a dinâmica geralmente envolve tomar um pedaço padrão do URL e usá-lo para a instanciação dinâmica criando uma classe de comando (FOWLER, 2006).

O caso estático tem a vantagem de lógica explícita, verificação de erros de compilação no despacho e muita flexibilidade no aspecto de seus URLs. O caso dinâmico permite que você adicione novos comandos sem alterar o manipulador da Web (KAYAL, 2008).

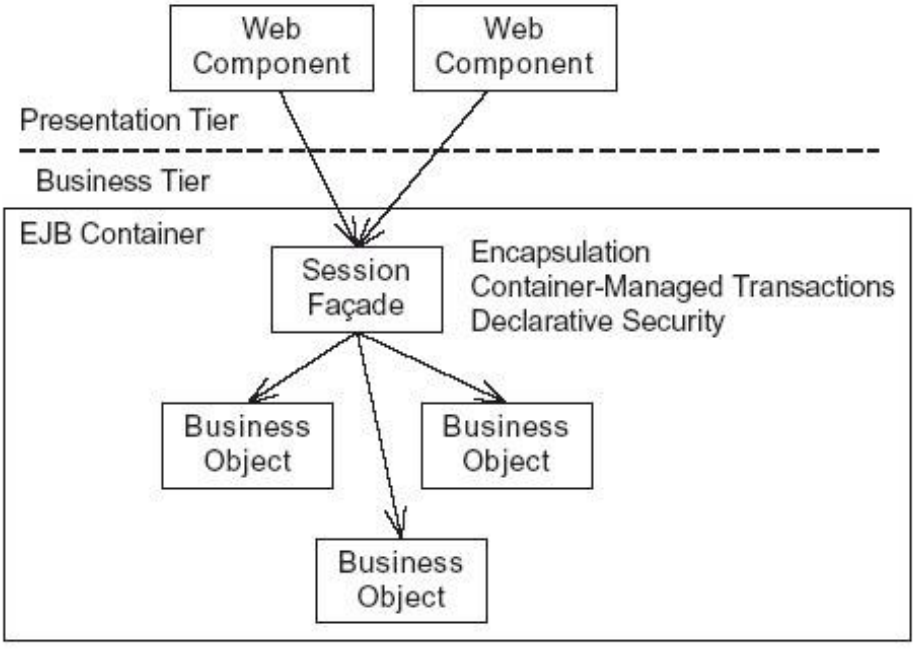
Por outro lado, o *Front Controller* é um padrão bastante útil para se usar em conjunto com o *Intercepting Filter.* De acordo Alur, Crupi e Malks (2003) um *Intercepting Filter* é essencialmente um decorador que envolve o manipulador do controlador frontal (*Front Controller)*. Isso permite a criação uma cadeia de filtro (ou *pipeline*) de filtros para lidar com diferentes problemas, como autenticação, log, identificação de localidade.

3.2 SESSION FAÇADE

O padrão *Session Facade* define um componente de negócios que contém nível superior e centraliza interações complexas entre componentes de negócios de nível inferior. Segundo Alur, Crupi e Malks (2003), um *Session Façade* é implementado como uma sessão *Enterprise Bean*. Ele fornece aos clientes uma interface única para a funcionalidade de um subconjunto da aplicação. Também desacopla componentes de baixo nível de negócios uns dos outros, tornando os projetos mais flexíveis e compreensíveis.

O *Session Façade* fornece serviços simples aos clientes, ocultando as complexidades das interações dos serviços de negócios. Um *Session Façade* pode invocar várias implementações do *Application Service* ou *Business Objects*. Além disso, o *Session Façade* também pode encapsular um *Value List Handler* (BASHAM; SIERRA; BATES, 2008).

Figura 6: Representação da lógica de encapsulamento com um *Session Façade* EJB.

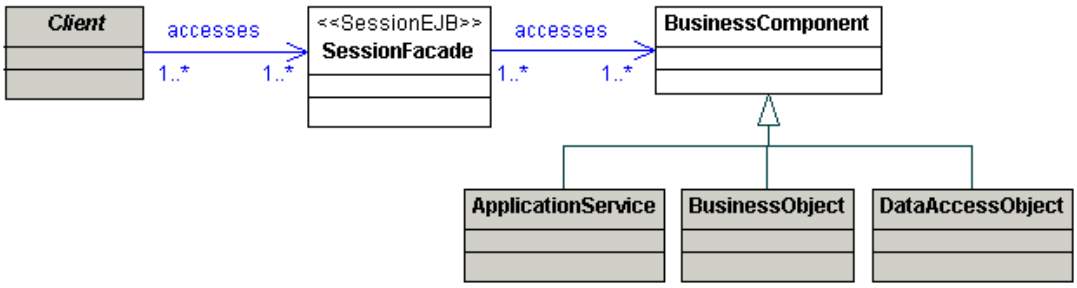


Fonte: RICHARDSON (2016).

Brown (2001) pressupõe que, se os clientes EJB acessarem *beans* de entidade diretamente na rede, eles devem efetuar várias chamadas remotas, levando

a um aumento do tráfego de rede e um desempenho reduzido. Um *Session Façade* resolve esses problemas apresentando ao cliente objetos com uma interface unificada para os EJBs subjacentes. Os objetos do cliente interagem somente com a fachada (um *Stateless Session Bean*), que reside no servidor e invoca os métodos EJB apropriados.

Figura 7: Diagrama de classe do padrão *Session Facade.*



Fonte: ALUR; CRUPS; MALKS (2003).

O padrão *Session Façade* sobrepõe também a vantagem de aplicar a execução de um caso de uso em uma chamada de rede e fornecer uma camada limpa na qual é encapsulada a lógica de negócios e de fluxo de trabalho usada para atender o caso de uso. Este padrão é geralmente implementado como uma camada de *Stateless Session* (embora o padrão também possa ser implementado com *Stateful Session Beans*). (MARINESCU, 2002)

O *Session Facade* organiza a lógica de negócio para o cliente. Buschmann *et. al.* (1996, p. 295) descrevem que na sua solução deve-se "usar um *bean* de sessão como uma fachada para encapsular a complexidade das interações entre os objetos de negócios que participam de um fluxo de trabalho". Além disso, O padrão *Session Façade* faz o uso do *Stateless Session* EJB para implementar sua lógica de negócios na camada EJB, e pode usar interfaces locais de *Entity Beans* para fazer chamadas para banco de dados.

3.3 DATA TRANSFER OBJECT

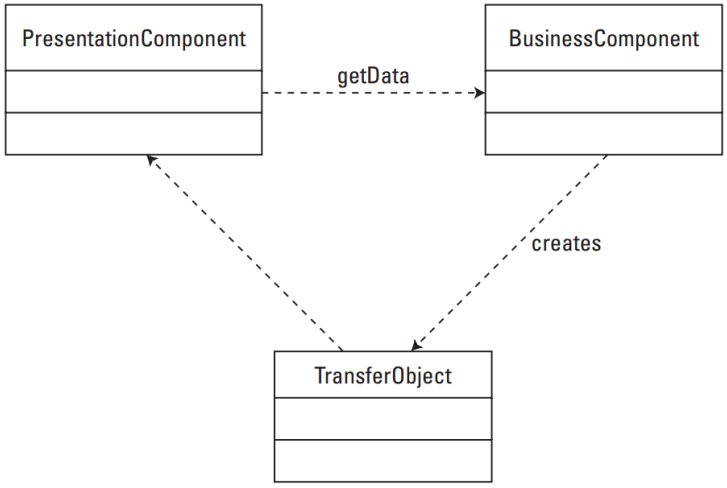
O padrão *Data Transfer Object* (DTO), ou *Transfer Object*, é um padrão de *design* que é comumente usado em aplicativos *Enterprise Java Beans* (EJB). Um

DTO, também chamado erroneamente de *Value Object,* encapsula um conjunto de valores, permitindo que clientes remotos solicitem e recebam todo o conjunto de valores com uma única chamada remota (PANTALEEV; ROUNTEV, 2007).

O padrão DTO fornece as melhores técnicas e estratégias para a troca de dados em níveis (isto é, através dos limites do sistema) para reduzir a sobrecarga da rede, minimizando o número de chamadas para obter dados de outro nível (ALUR; CRUPI; MALKS, 2003).

Embora a principal razão para usar um DTO é a divisão em lotes, que seriam múltiplas chamadas remotas em uma única chamada, vale ressaltar que outra vantagem é encapsular o mecanismo de serialização para a transferência de dados através do fio. Ao encapsular a serialização como esta, os DTOs mantêm esta lógica fora do resto do código e também fornece um ponto claro para alterar a serialização (FOWLER, 2006).

Figura 8: Exemplo do funcionamento do padrão DTO.

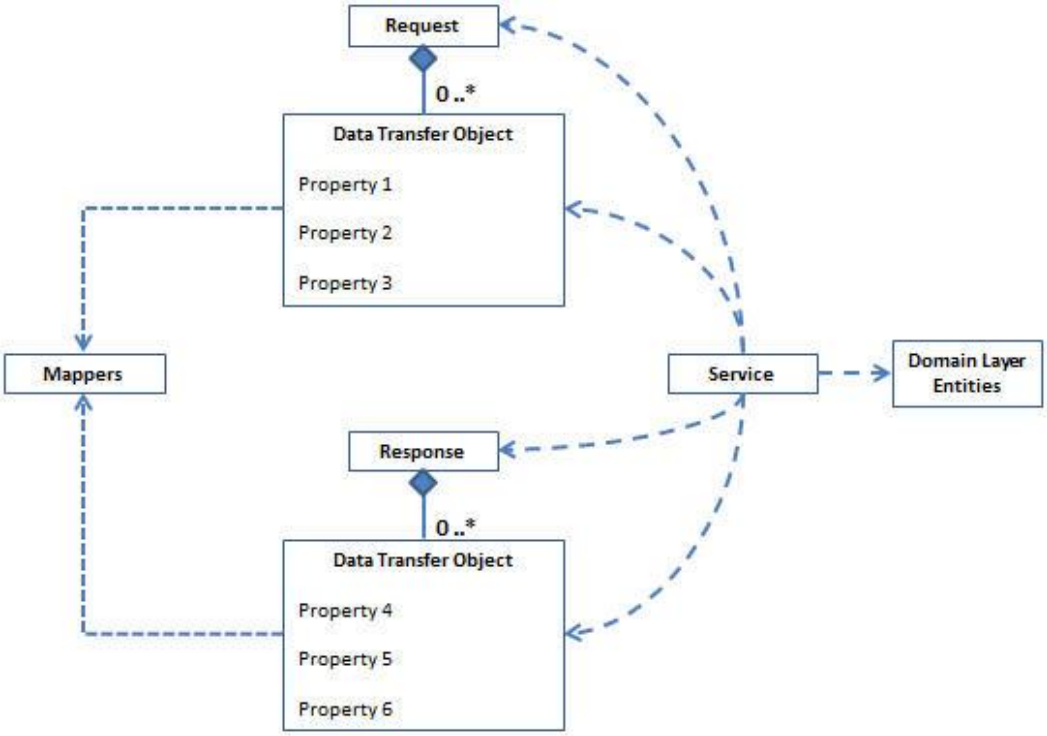


Fonte: McGovern et. al. (2003)

McGovern et. al. (2003) explica que o *TransferObject* é uma classe serializável que agrupa atributos relacionados e encapsula os dados como resultado da chamada para a classe *BusinessComponent*. Um único método pode ser usado para enviar e recuperar *TransferObject.* O EJB pode pegar um objeto de transferência já criado ou criar um novo *TransferObject* e preenchê-lo com valores da fonte de dados para atributos dentro do objeto de transferência e enviá-lo.

De acordo com Marinescu (2002) os objetos de transferência de dados podem ser usados tanto para as operações de leitura como para as operações de atualização em um sistema distribuído. Quando um cliente precisa atualizar alguns dados no servidor, ele pode criar um DTO que envolve todos os serviços necessários para executar as atualizações e enviá-lo para o servidor (geralmente para um *Session Façade*) para processamento.

Figura 9: Lógica de Mapeamento do padrão *Data Transfer Object*



Fonte: SERVICE DESIGN PATTERN (2011).

O DTO facilita a manipulação de dados para os serviços de *request* e *respose* pois esses serviços não precisam de APIs especificas da estrutura (por exemplo, para o XML). O DTO também desacopla entidades de camada de domínio das estruturas de solicitação e resposta porque elas são criadas como entidades separadas cujo único objetivo é definir como os dados são recebidos e retornados de um serviço. Os dados podem ser mapeados dentro e fora do DTO através de código personalizado ou com tecnologias de ligação de dados (COUCH; STEINBERG, 2002)

Quando se trabalha com uma interface remota, como o *Remote Facade*, cada chamada para ele é dispendioso. Como resultado, é necessário reduzir o número de chamadas, e isso significa que precisa transferir mais dados em cada chamada.

Uma maneira de fazer isso é usar muitos parâmetros. No entanto, isso geralmente é difícil de programar - na verdade, muitas vezes é impossível com linguagens como Java que retornam apenas um único valor (FOWLER, 2006).

Portanto, Alur, Crupi e Malks (2003) afirmam que a solução é criar um objeto de transferência de dados (DTO) que pode armazenar todos os dados para a chamada. Ele precisa ser serializável para atravessar a conexão, geralmente um *assembler* é usado no lado do servidor para transferir dados entre o DTO e quaisquer objetos de domínio.

O DTO tem o objetivo de conduzir dados entre um EJB e seu cliente, onde cada uma está em uma JMV distinta. Segundo Calçado (2016), o padrão define o porquê, como e quando empacotar dados em toda a rede em pacotes em massa chamando os objetos de transferência de dados (DTOs). Os dois padrões de acompanhamento (*Domain* e *Custom* DTO) fornecem orientação sobre como DTOs devem ser projetados.

3.4 DATA ACCESS OBJECT

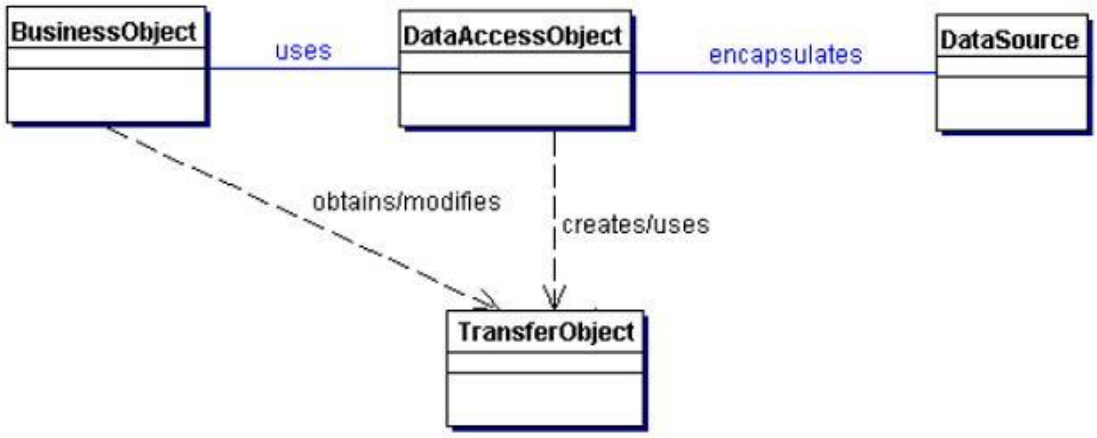
O *Data Access Object* (objeto de acesso a dados, em português) permite acoplamento livre entre os níveis de negócios e de recursos. O DAO adota toda a lógica de acesso a dados para criar, recuperar, excluir e atualizar dados de um armazenamento persistente. Objeto de acesso a dados usa objeto de transferência para enviar e receber dados (ALUR; CRUPI; MALKS, 2003).

O padrão de acesso a dados encapsula todos os mecanismos de acesso à fonte de dados, incluindo conectividade, consultando o banco de dados e criando objetos de acesso a dados diferentes com base na fonte de dados necessária para o aplicativo. A interface que o DAO fornece aos seus clientes não é alterada quando os dados migram de uma origem de dados para outra (MCGOVERN *et. al. ,*2003).

Para Couch e Steinberg (2002) o princípio básico do padrão DAO é a abstração das consultas de banco de dados isolados do código de implementação do *bean* e em um conjunto separado de classes. Neste padrão, a implementação é iniciada com um padrão *Factory* para produzir uma interface básica para o banco de dados subjacente. Essa interface fornece métodos para acessar os dados imensos,

enquanto cada implementação dele trata das nuances específicas de cada produto do banco de dados.

Figura 10: Diagrama de classe exibindo os relacionamentos para o padrão *Data Access Object*



Fonte: ALUR; CRUPS; MALKS (2003).

O padrão DAO elimina a necessidade de conhecimento prévio da fonte de dados e dos tipos de *drives* e interfaces utilizados para acesso à persistência. Para Micka e Kouba (2013), o DAO desacopla os dados de sua abstração identificando quatro classes: a classe de negócio *Business Object*, que que utiliza o padrão através de chamada à classe *Data Access Object*. A classe *Data Access Object,* implementa a forma de acesso aos dados, retornando um objeto da classe *TransferObject*. E a classe *Data Source* que representa a forma de acesso que é encapsulada pela classe *Data Access Object* e normalmente refere-se a uma classe que implementa a interface JDBC, no caso de um sistema JEE.

O padrão DAO ainda é um padrão essencial, e sua solução original ainda é válida, embora a motivação para sua implementação tenha mudado em sua ênfase. Para Couch e Steinberg (2002), além de proteger contra o impacto de uma mudança improvável no tipo de fonte de dados, sua importância está na testabilidade e em seu uso na estrutura do código para mantê-lo limpo no código de acesso de dados. Ademais, há a possibilidade em usá-lo como uma forma de encapsular sistemas de armazenamento de dados herdados para simplificar o acesso a implementações complexas de fontes de dados.

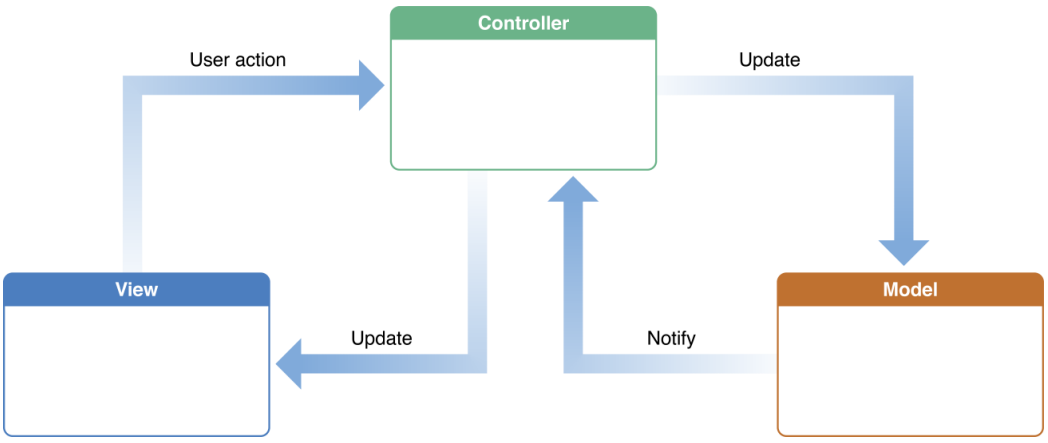
Objetos de acesso a dados fornecem a portabilidade para aplicativos de uma fonte de dados para outra fonte de dados. Muitos aplicativos modernos exigem um banco de dados persistente para seus objetos.

O acesso aos dados varia consoante a fonte dos dados. O acesso ao armazenamento persistente, como um banco de dados, varia muito dependendo do tipo de armazenamento (bancos de dados relacionais, bancos de dados orientados a objetos, arquivos simples dentre outros) e a implementação do fornecedor (ALUR; CRUPI; MALKS, 2003).

3.5 MODEL-VIEW-CONTROLLER

O padrão de design *Model-View-Controller* (MVC) é uma maneira de separar a interface do usuário da substância do aplicativo. Segundo Golcalves (2013), o MVC é um padrão arquitetônico usado para isolar a lógica de negócios da interface do usuário. A lógica de negócios não se mistura bem com o código UI. Quando os dois são misturados, as aplicações são muito mais difíceis de manter e menos escaláveis. Quando o MVC não é aplicado, resulta em uma aplicação mal acoplada; com este tipo de aplicação, é mais fácil modificar a aparência visual da aplicação ou as regras empresariais subjacentes sem que uma afete a outra.

Figura 11: Representação do funcionamento do padrão MVC.



Fonte: APPLE DEVELOPER (2012).

O objetivo do MVC é dividir a aplicação em três camadas, obtendo respostas para o novo paradigma de aplicação existente. De acordo com Cassimiro

(2010), o *Model* representa as entidades do banco de dados, onde nele contém as classes apenas com atributos *get* e *set.* Deste modo, as regras do sistema são definidas na camada de negócio, que na maioria das vezes são compostas por métodos que possuem as regras de negócios. A *View* apresenta os dados para o usuário da aplicação. Porém, ela mostra os dados do *Model* no momento da interação com o usuário. O *Controller* é o encarregado por processar e responder aos eventos do usuário na *View* e solicita as alterações no *Model*

Bucanek (2009) afirma que a maioria dos padrões abordam problemas específicos, e o MVC descreve a arquitetura de um sistema de objetos. Ele pode ser aplicado a subsistemas isolados ou aplicações inteiras. O padrão de projeto MVC não é definido de forma clara, diferente de muitos outros padrões, deixando complicações em implementações alternativas. É mais uma filosofia do que uma “receita”.

Contudo, a arquitetura MVC separa os dados do aplicativo (contidos no modelo) dos componentes de apresentação gráfica (a visualização) e da lógica de processamento de entrada (o controlador). O MVC originalmente apareceu em *Smalltalk*-80 como um método para separar interfaces de usuário a partir de dados de aplicação subjacentes (DEITEL; DEITEL; SANTRY, 2002).

De acordo com Wojciechowski *et al.* (2004), o MVC organiza um aplicativo interativo em três módulos separados: o primeiro para o modelo de aplicativo com sua representação de dados e lógica de negócios, o segundo para exibições que fornecem apresentação de dados e entrada do usuário e o terceiro para um controlador para fazer o *dispatch* das requisições.

4 APLICAÇÃO E FUNCIONAMENTO DOS PADRÕES JAVA EE

4.1 PADRÃO FRONT CONTROLLER

O padrão do *Front Controller* é aplicável e útil para todos os tipos de aplicativos, sejam eles aplicativos para *web* ou *desktop*, o padrão não está limitado a qualquer linguagem de programação única ou estrutura. Antes dos *framework*s MVC serem lançados no mercado, o padrão de projeto ainda estava em uso (BASHAM, SIERRA; BATES 2008).

Contudo, em uma aplicação *web*, o *Front Controller* é quase sempre implementado como um *servlet* (*Servlet Front*). Embora seja tecnicamente possível usar uma página JSP, porém essa prática não é aconselhada, pois as páginas JSP não devem ser usadas para implementar uma lógica complexa. Uma segunda opção mais agradável é o uso um filtro de *servlet* com o *Front Controller* (GILART-IGLESIAS *et. al.,* 2011).

Na implementação da estratégia *Servlet Front* (ANEXO B), o controlador gerencia os aspectos do processamento de solicitações que estão relacionados ao fluxo de processamento e controle de negócios. Essas responsabilidades estão relacionadas, mas são logicamente independentes, da formatação de exibição e são mais apropriadamente encapsuladas em um *servlet* em vez de em uma página JSP.

Entretanto, esta estratégia possui alguns inconvenientes potenciais. Kayal (2008) cita uma em particular, ela não alavanca alguns dos utilitários de ambiente de tempo de execução da página JSP, como a população automática de parâmetros de solicitação em propriedades no *Helper*. Contudo, esse inconveniente é mínimo pois é relativamente fácil criar ou obter utilitários semelhantes para uso geral. (ALUR; CRUPI; MALKS, 2003).

McGovern *et. al.* (2003) explica que o *servlet* do *Front Controller* chama o padrão de comando para processá-lo. O padrão de comando, por sua vez, cria a página de exibição correspondente que deve ser passada para o *dispatcher.* O resultado de saída é enviado para o *dispatcher* para assim mostrar a exibição apropriada.

Além disso, há também a estratégia *command* e *controller*, que é baseado no padrão *Command* (GoF). De acordo com Alur, Crupi e Malks (2003), esta é a combinação dos padrões *Command* e *Front Controller,* e fornece uma interface

genérica à qual o controlador frontal pode delegar a responsabilidade. A utilização do padrão de comando requer alterações mínimas ou nenhuma ao *Controller* e ao *Helper*. Como o processamento de comandos e a invocação são desacoplados, o processador de comandos pode ser usado com vários clientes.

Figura 12: Código simples do padrão *Front Controller* para a estratégia *Command* e *Controller*.

```
public class FrontController extends HttpServlet {
    public void init() throws ServletException {}
    public void service(HttpServletRequest request,
        HttpServletResponse response) throws
    ServletException, IOException {
        String displayPage;
        try {
            Helper h = new Helper(request);
            CommandPattern cmd = h.getCommandPattern();
            displayPage = cmd.execute(request, response);
        } catch (Exception e) {
            //a wrapper for log4j.jar from apache open source.
            Log4jWrapper.log("FrontController Pattern", e);
            //take the user to a error page.
        }
        dispatch(request, response, displayPage);
    }
}

```

Fonte: McGovern *et. al.* (2003).

Em um site complexo, há diversas coisas semelhantes que devem ser tradadas ao lidar com uma requisição. Segundo Kayal (2008) essas coisas incluem segurança, internacionalização e exibição específica para determinados usuários. Se o comportamento do controlador de entrada estiver disperso em vários objetos, grande parte desse comportamento pode acabar duplicado. Além disso, é difícil alterar o comportamento em tempo de execução.

Portanto, como solução, um controlador deve ser utilizado como o ponto inicial de contato para lidar com uma solicitação. O controlador chama serviços de segurança, como autenticação e autorização, delega processamento à camadas de negócios, gerencia uma escolha de visão adequada, realiza tratamento de erros, define estratégias de geração de criação de conteúdo (ALUR; CRUPI; MALKS, 2003).

Entretanto, a única desvantagem deste padrão é que nem toda aplicação pode fazer uso dele por causa do esforço envolvido. Apenas os aplicativos que possuem estrutura personalizada fazem o uso deste padrão de projeto (BASHAM, SIERRA; BATES 2008).

4.2 PADRÃO SESSION FAÇADE

O *Session Facade* abstrai as interações de objeto de negócios subjacentes e fornece uma camada de serviço que expõe apenas as interfaces necessárias. Assim, ele esconde, do ponto de vista do cliente, as interações complexas entre os participantes. O padrão *Session Façade* gerencia as interações entre os dados de negócios e os objetos de serviço de negócios que participam do fluxo de trabalho, e encapsula a lógica de negócios associada aos requisitos (ALUR; CRUPI; MALKS, 2003).

Assim, o *bean* de sessão (que representa o *Session Façade*) gerencia as relações entre objetos de negócios. O *bean* de sessão também gerencia o ciclo de vida desses participantes criando, localizando (procurando), modificando e excluindo-os conforme exigido pelo fluxo de trabalho. Em uma aplicação complexa, o *Session Facade* pode delegar esse gerenciamento de estilo de vida em um objeto separado (GILART-IGLESIAS *et. al.,* 2011).

Um uso comum de *beans* de sessão é como uma fachada que encapsula as interações entre objetos no *Business Tier*. O *bean* de sessão serve para abstrair esta complexidade, proporcionando uma interface mais simples para os clientes. Este padrão é descrito em detalhes em Padrões J2EE - Padrão de Fachada de Sessão ALUR, CRUPI E MALKS (2003)

É considerado uma boa prática retirar a lógica do *bean* entre entidades e passar para *beans* de sessão para minimizar o acoplamento entre os *beans* de entidade. Os *beans* de entidade podem ser acessados através de interfaces locais, pois a fachada de *beans* de sessão fornece acesso a clientes remotos. Esta abordagem é mais eficaz quando existem vários *beans* de entidade estreitamente relacionados (MARINESCU, 2002).

Para apresentar um exemplo de implementação deste padrão, Alur, Crupi e Malks (2003) elucidam a situação de um Aplicativo de Serviços Profissionais (PSA), onde o fluxo de trabalho relacionado a *beans* de entidade (como *Project* e *Resource)* é encapsulado na classe *ProjectResourceManagerSession*, implementado usando o padrão *Session Facade*.

Este exemplo (ANEXO C) mostra a interação entre as entidades de Recursos e Projeto, bem como outros componentes de negócios, como *Value List Handlers* e *Tansfer Object Assemblers,* além de apresentar as implementações das interfaces *Remote* e *Home* para o padrão *Session Facade*.

Portanto, o padrão *Session Façade* oferece um meio pelo qual o servidor pode diferenciar entre diferentes clientes. Em uma transação, esse padrão ajuda o servidor a rastrear informações sobre o cliente; para melhoria de desempenho no servidor este padrão pode armazenar em cache as informações do usuário (BUSCHMANN; HENNEY; SCHMIDT, 2007).

4.3 PADRÃO DATA TRANSFER OBJECT

Segundo Bien (2009), o DTO é apenas um contêiner de dados que é usado para transportar dados entre camadas. Ele contém atributos que podem ser usados em classes públicas sem qualquer *getters/*setters. Os DTOs são fracos, e em geral não contêm qualquer lógica de negócios.

Um *Data Transfer Object* (Objeto de Transferência de Dados), em muitas das vezes é mais do que simplesmente um amontoado de campos com os getters e setters. Este padrão é significativo, pois permite a movimentação de vários pedaços de dados sobre uma rede em uma única chamada, um truque essencial para sistemas distribuídos (FOWLER, 2006).

Alur, Crupi e Malks (2003) descrevem que, em sua implementação, o padrão DTO fornece uma camada de acesso de serviço centralizada uniforme aos clientes, encapsulando a complexidade das interações entre vários objetos de negócios.

Para Buschmann, Henney e Schmidt (2007), quando se usa outros padrões como MVC, por exemplo, se faz necessário a realização de chamadas para consultar e atualizar dados em objetos de componentes remotos. Um DTO possui uma composição própria, contendo apenas os dados correspondentes aos atributos, consultas para aceder a eles e uma forma de inicialização e opcionalmente uma de definição dos valores dos dados.

De acordo com Kayal (2008), é possível reduzir o número de chamadas remotas usando o padrão de design DTO. Este padrão propõe o uso de um objeto de transferência (*Transfer Object)*, que é projetado para transferir um conjunto de dados do cliente para o servidor, ou vice-versa.

Marinescu (2002) afirma que frequentemente, os DTOs são usados para melhorar o desempenho de casos de uso em entidades que estão profundamente

interligadas. Em linhas gerais, é mais rápido transferir apenas um subconjunto do grafo interconectado de entidades, em vez de toda a árvore. Por esse propósito, até mesmo as relações 1:1 (de um para um) poderiam ser condensadas em um DTO. O DTO pode ser criado diretamente pelo *EntityManager* como resultado de um *NamedQuery.*

Figura 13: Código de implementação do DTO em uma classe Java *Serializable.*

```
1  import java.io.Serializable;
2  public class SomeDTO implements Serializable {
3      private long attribute1;
4      private String attribute2;
5      private String attribute3;
6      ...
7      public long getAttribute1();
8      public String getAttribute2();
9      public String getAttribute3();
10     ...
11 } //SomeSTO
12
```

Fonte: MARINESCU (2002).

Em um caso de uso de Java EE mais comum, o DTO atua como um proxy no lado do cliente e representa uma árvore cara para construir e transportar de entidades separadas. A construção DTO pode ser realizada diretamente no serviço, em um *EntityManager*, ou pode ser encapsulado em um DAO (MARINESCU, 2002).

Pawlak, Seinturier e Retaillé (2005) afirmam que um DTO pode ser usado em dois casos: quando uma unidade de computação cliente precisa acessar mais de uma parte de dados retornada pela camada de negócios (downloads múltiplos), ou quando uma unidade de computação de cliente precisa enviar mais de uma parte de dados para ser concluída (vários carregamentos).

4.4 PADRÃO DATA ACCESS OBJECT

O papel do DAO é semelhante ao do padrão do *Adapter,* que atua como um intermediário entre duas classes participantes, convertendo a interface de uma classe para ser usada com a outra. Isso promove a reutilização de funcionalidades mais antigas. O padrão *Adapter* fornece a interface que um cliente espera, utilizando

os serviços fornecidos por uma classe que implementa uma interface diferente (MCGOVERN *et. al.* ,2003).

Duas estratégias diferentes podem ser usadas para implementar esse padrão, depende se a fonte de dados vai ser a mesma implementada em aplicações corporativas diferentes. Quando a fonte de dados é a mesma, a implementação de base usa o padrão de Design *Factory* para fazer o padrão de acesso de dados produzir um número finito de DAOs. Esses DAOs são baseados na exigência do aplicativo corporativo (KAYAL, 2008).

Alur, Crupi e Malks (2003) afirmam que a criação de DAOs é altamente flexível ao adotar os padrões de *Abstract Factory* e os padrões *Factory Method.* Quando os aplicativos usam um único tipo de armazenamento persistente (como Oracle RDBMS, por exemplo), e não há necessidade de mudar o armazenamento subjacente de uma aplicação para outra, implementar a estratégia DAO *Factory Method* para produzir uma variedade de DAOs necessários pela aplicação.

Figura 14: Código do padrão DAO usando a estratégia *Factory Method.*

```
public class OracleDAOFactory extends DAOFactory {

    // package level constant used look up the
    // DataSource name using JNDI
    static String DATASOURCE_DB_NAME =
        "java:comp/env/jdbc/CJPOraDB";

    public static CustomerDAO getCustomerDAO()
    throws DAOException {
        return (CustomerDAO) createDAO(CustomerDAO.class);
    }

    public static EmployeeDAO getEmployeeDAO()
    throws DAOException {
        return (EmployeeDAO) createDAO(EmployeeDAO.class);
    }

    // create other DAO instances
    ...

    // method to create a DAO instance. Can be optimized to
    // cache the DAO Class instead of creating it everytime.
    private Object createDAO(Class classObj)
    throws DAOException {
        // create a new DAO using classObj.newInstance() or
        // obtain it from a cache and return the DAO instance
    }
}
```

Fonte: ALUR, CRUPI e MALKS (2003).

Enquanto a estratégia DAO *Factory Method* é mais comumente usada, é possível a estender ainda mais a flexibilidade da implementação da fábrica adotando o padrão *Abstract Factory.* Esta necessidade de maior flexibilidade surge com a probabilidade em alterar dados persistente com frequência. Para isso, deve-se encapsular vários tipos de fontes de dados, de modo que cada fábrica de DAO forneça a implementação do padrão para um tipo de armazenamento persistente. Embora a

maioria das aplicações tipicamente utilizem um único tipo de fonte de dados (tais como a Oracle RDBMS), o uso da estratégia a DAO *Factory Method* pode ser adequada (ALUR, CRUPI; MALKS, 2003).

Figura 15: Código do padrão DAO usando a estratégia *Abstract Method.*

```
// Abstract class DAO Factory
public abstract class DAOFactory {

    // List of DAO types supported by the factory
    public static final int CLOUDSCAPE = 1;
    public static final int ORACLE = 2;
    public static final int SYBASE = 3;
    ...

    // There will be a method for each DAO that can be
    // created. The concrete factories will have to
    // implement these methods.
    public abstract CustomerDAO getCustomerDAO()
    throws DAOException;
    public abstract EmployeeDAO getEmployeeDAO()
    throws DAOException;
    ...

    public static DAOFactory getDAOFactory(int whichFactory) {
        switch (whichFactory) {
            case CLOUDSCAPE:
                return new CloudscapeDAOFactory();
            case ORACLE:
                return new OracleDAOFactory();
            case SYBASE:
                return new SybaseDAOFactory();
                ...
                default:
                    return null;
        }
    }
}
```

Fonte: ALUR, CRUPI e MALKS (2003).

Um padrão de design DAO ajuda um aplicativo para executar várias operações CRUD no banco de dados. As classes DAO fornecem métodos para métodos de inserção, exclusão, atualização e pesquisa. A finalidade básica de criar o DAO é o acoplamento fraco e a não-repetição do código (MCGOVERN et. al. ,2003).

Em qualquer aplicativo que vai interagir com o banco de dados, é necessário executar operações CRUD nas tabelas do banco de dados e uma vez que as operações de tabela podem ser feitas por diferentes classes e, portanto, torna-se pesado repetir o mesmo código em várias classes. Além disso, mesmo depois de repetir o código, torna-se difícil manter o código de interação da base de dados sempre que forem necessárias alterações na forma como a interação do banco de dados está sendo feita (BIEN, 2009).

4.5 PADRÃO MODEL VIEW CONTROLLER

O padrão MVC é usado para separar as tarefas da aplicação. O modelo representa um objeto ou um JAVA POJO que carrega dados. Ele também pode ter a lógica para atualizar o controlador se seus dados mudam. A visão representa a visualização dos dados que o modelo contém. E o controlador age tanto no modelo como na visão. Ele controla o fluxo de dados no objeto do modelo e atualiza a visualização sempre que os dados mudam. Mantém a visão e o modelo separados (KASSEM *et. al.*, 2000).

Na *web,* há duas entidades interagindo: o servidor e o cliente. O servidor é o único responsável por manter o modelo. O cliente executa um navegador *Web* que executa solicitações para o servidor. Essas solicitações são conduzidas adequadamente, provocam mudanças no modelo e dão origem a uma resposta que o cliente processa para o usuário. Uma interação (por exemplo, rolar uma lista de entradas) pode não envolver o servidor (FORD, 2007).

Na *web*, o controlador é responsável pela manipulação de eventos do usuário, preparando a visualização e empurrando-a para o renderizador. O MVC também pode ser usado na *web*, e uma abstração de frameworks de desenvolvimento web podem fornecer uma arquitetura MVC bem projetada, onde o programador tem apenas a tarefa de preencher os espaços vazios e todo o levantamento pesado web é tomado cuidadosamente pela arquitetura (DEITEL; DEITEL; SANTRY, 2002).

Um exemplo de implementação usando MVC é a construção de uma calculadora (ANEXO D) elucidado por Code Project (2008). Onde um formulário abre a *View* e os eventos são passados para o *Controller* que chama os métodos no *Model*, como *ADD* / *Subtract* / *NumberPress*. O modelo cuida de todo o trabalho e mantém o estado atual da calculadora.

Em uma típica instanciação de padrões MVC o controlador leva uma interface para a vista e modelo. É importante ressaltar que a visualização normalmente interagirá com o controlador se precisar de notificação de eventos que são disparados através da visualização (como um clique de botão). Neste caso, há um de construtor controladores que passam uma referência de si próprio para a classe *View.*

Entretanto, neste exemplo, a visão não interage com o modelo, ele simplesmente recebe as solicitações de atualização do controlador. O controlador acessa a visão através da propriedade *Total*. E a visão também passa eventos de clique para o controlador. Por outro lado, a visão não deve tomar conhecimento do

controlador, exceto para dar-lhe notificação de alguns eventos passados para um *IController* (classe de manipuladores de eventos no controlador). E a visão tem a função de mostrar a calculadora para o usuário final.

4.6 PRÁTICAS ACONSELHÁVEIS E DESACONSELHÁVEIS DO JAVA EE

Nas últimas décadas, houveram muitos estudos sobre as boas práticas da plataforma Java EE. Atualmente, há inúmeras pesquisas que fornecem informações de como os aplicativos Java EE devem ser escritos. Entretanto, existem tantos recursos - muitas vezes com recomendações contraditórias - que caminhar neste embaraço de informações tornou-se um obstáculo para adoção do Java EE (BOTZUM *et al.*, 2007).

Contudo, Brown *et. al.* (2003) destacam uma série de orientações para o uso dos padrões Java EE. Dentre elas estão: sempre usar o MVC, aplicar testes de unidade automatizados e arneses de teste em cada camada, sempre usar fachadas sessão sempre que usar componentes EJB, usar *beans* de sessão sem estado em vez de *beans* de sessão com estado e usar transações gerenciadas por contêiner.

As práticas desaconselháveis das camadas de apresentação, de negócios e de apresentação são menos numerosas do que as soluções ótimas, pois entram em conflito com a maioria das recomendações dos padrões. Uma prática desaconselhável comum é expor todos os atributos do *enterprise bean* através dos métodos *Getter/Setter.* Isso força os clientes a realizar várias chamadas remotas de granulação fina e a criar o potencial para inserir uma grande quantidade de ruído entre as camadas, ou seja, cada chamada de método carrega um *overhead* de rede que afeta o desempenho e a escalabilidade (ALUR; CRUPI; MALKS, 2003).

Entretanto Bien (2009) assegura como referência para solução dessa prática, a utilização de objetos de valor para transferir dados agregados do cliente e para ele, ao invés de expor os *getters* e *setters* para cada atributo.

Alur, Crupi e Malks (2003) descrevem que outra prática não recomendada é a utilização um *bean* de entidade como objeto somente leitura, pois qualquer *bean* de entidade está sujeito às semânticas de transações baseados em seus níveis de isolamento de transações definidos no escritor de distribuição. Fazer o uso de um *bean* de entidade como objeto somente leitura desperdiça recursos caros e resulta em transações de atualizações desnecessárias para o armazenamento persistente

Com isso, Laka (2014) adverte como referência para solução, o encapsulamento de todos os acessos às origens de dados empregando o padrão DAO. Essa prática fornece uma camada centralizada de códigos de acesso a dados além de simplificar o código do *bean* de entidade.

Os padrões de projeto são um dos temas mais importantes, desafiadores e úteis no software. Um bom conhecimento fornece um grande conjunto de ferramentas para problemas comuns enfrentados. O Java EE leva esse passo adiante e apresenta uma maneira muito mais fácil e integrada de usar padrões de design em projetos empresariais. Os padrões foram criados para facilitar, contudo, se forem usados extensivamente sem razão, eles tendem a complicar o projeto. Ter conhecimento de um padrão não significa necessariamente ter que usá-lo a menos que se saiba que ele resolva um problema potencial (YENER; THEEDOM, 2015)

CONSIDERAÇÕES FINAIS

Conclui-se que nas condições do presente estudo realizado com padrões de projetos e seu uso em desenvolvimentos de aplicações Java EE, observou-se que com a expansão da complexidade de sistemas, tornou-se indispensável o emprego de métodos e técnicas da orientação a objetos por meio de uma linguagem padronizada.

Compreendeu-se então, que nesse contexto são aplicados os padrões de projeto, e proporciona a reutilização técnicas e soluções para resolver problemas recorrentes. Os padrões de projetos cada vez mais são considerados como ferramentas essenciais para o projeto reutilizável de software.

Com relação aos métodos utilizados para a construção de sistemas web com o emprego de padrões de padrões que utilizam o Java EE encontra-se a divisão em camadas, que permite o uso de várias tecnologias para o mesmo serviço e possibilita a escolha da mais adequada nas características do problema.

Há vários padrões que podem ser adotados no desenvolvimento de aplicações Java EE, porém não se faz necessário a utilização de todos padrões catalogados, o estudo mostrou que é necessário analisar com bastante atenção seu emprego para que a aplicação não cause um resultado negativo, oposto do esperado.

Os objetivos destacados foram alcançados, visto que os padrões representam a experiência destilada que, através da sua assimilação, transmite conhecimentos especializados a desenvolvedores inexperientes. Eles ajudam a forjar a fundação de uma visão arquitetônica compartilhada, e coletiva de estilos.

Contudo, vale ressaltar que padrões são ferramentas extremamente valiosas para capturar e comunicar conhecimentos adquiridos e experiências para melhorar a qualidade e produtividade do software, abordando questões fundamentais no desenvolvimento de software.

Com base dos argumentos supracitados, deve-se incentivar mais estudos, com mais padrões de projetos, utilizando outras linguagens da programação orientada a objetos e, se possível com seu desenvolvimento prático na aplicação de cada um. Para avaliar o emprego de cada padrão e realizar comparações de condições favoráveis e desfavoráveis que podem ocorrer.

Sugere-se, portanto, que o estudo seja continuado com o emprego mais ou de todos os padrões catalogados para investigar a sua facilidade de uso em outras

plataformas de desenvolvimento, com a adoção de tecnologias mais avançadas objetivando melhores decorrências no desenvolvimento de software.

A principal delimitação do presente estudo foi a escassez de artigos e livros da língua portuguesa, além de estudos atuais voltados a esta temática.

REFERÊNCIAS

ALEXANDER, Christopher *et. al*. **A Pattern Language:** Towns, Buildings, Construction. New York: Oxford University Press, 1977. 1171p.

ALEXANDER, Christopher. **The Timeless Way Of Building**. New York: Oxford University Press, 1979. 546 p.

ALUR, Deepak; CRUPI, John; MALKS, Dan. **Core J2EE Patterns:** Best Practices and Design Strategies. 2 ed. [S.l.]: Pearson, 2003. 496 p.

APPLE DEVELOPER. **Cocoa Core Competencies**. Disponível em: <https://developer.apple.com/library/content/documentation/general/conceptual/devpedia-cocoacore/mvc.html>. Acesso em: 21 abr. 2017.

APPLETON, Brad. **Patterns and Software**: Essential Concepts and Terminology. Object Magazine Online, [S.l.], v. 3, n. 5, May 1997. Disponível em:<http://www.bradapp.com/docs/patterns-intro.html>. Acesso em:19 fev. 2017.

AQUINO JUNIOR, Gibeon Soares de. **Desenvolvimento de Sistemas Web com Java**: Frameworks, Padrões e Diretrizes para a Camada de Apresentação. 2002. 141 f. Dissertação de Mestrado (Pós-Graduação em Ciência da Computação) - Universidade Federal de Pernambuco, Recife, 2002. Disponível em: <http://repositorio.ufpe.br/bitstream/handle/123456789/2573/arquivo5091_1.pdf>. Acesso em: 06 mar. 2017.

BASHAM, Bryan; SIERRA, Kathy; BATES, Bert. **Head First Servlets and JSP™.** 2 ed. Sebastopol, CA: O'Reilly, 2008. 883 p.

BIEN, Adam. **Real World Java EE Patterns**: Rethinking Best Practices. [S. l.], 2009. 280 p.

BOTZUM, Keys *et al.* The Top Java EE Best Practices. **IBM WebSphere Developer Technical Journal,** [S.l.], 24 jan. 2007. IBM Developer Works, p. 1-18. Disponível em: <https://www.ibm.com/developerworks/websphere/techjournal/0701_botzum/0701_botzum-pdf.pdf>. Acesso em: 26 maio 2017

BROEMMER, Darren. **J2EE Best Practices:** java design patterns, automation, and performance: 1 ed. Canadá: Wiley, 2003. 496 p.

BROWN, K. *et al.* **Enterprise Java Programming With IBM Websphere.** 2 ed. [S.L.]: IBM Press, 2003. 960 p.

BROWN, W. J. *et. al.* **Antipatterns**: Refactoring Software, Architectures, and Projects in Crisis. 1 ed. Canadá: Wiley & Sons, Inc., 1998. 336 p.

BUCANEK, James. **Learn Objective-C for Java Developers.** New York: Apress, 2009. 520 p.

BUSCHMANN, F., *et al.* **Pattern-Oriented Software Architecture**: A System of Patterns, vol. 1. England: John Wiley & Sons, 1996.

BUSCHMANN, Frank; HENNEY, Kevlin; SCHMIDT, Douglas C.. **Pattern-Oriented Software Architecture**: A pattern language for distributed computing. Vol. 4. England: John Wiley & Sons Ltd, 2007. 639 p.

CALÇADO, Phil. **Fragmentos de um programador**: Artigos e insights da carreira de um profissional. [S.l.]: Casa do Código, 2016. 141 p.

CASSIMIRO, Matheus Higino de Oliveira. **Padrões Arquiteturais e Seus Benefícios no Processo de Manutenção do Software.** 2010. 43 p. TCC (Bacharel em Ciência da Computação) Faculdade de Ciências Empresariais – FACE, Universidade FUMEC, Belo Horizonte, 2010. Disponível em: <http://professores.dcc.ufla.br/~terra/publications_files/students/>. Acesso em: 19 fev. 2017.

COAD, Peter. **Object-Oriented Patterns**. Communications of the ACM, New York, v.35, n.9, p.152-159, Sep. 1992.

COAD, Peter; YOURDON, Edward**. Análise Baseada em Objetos**. Rio de Janeiro: Campus, 1991. 225p.

CODE PROJECT. **Simple Example Of MVC (Model View Controller) Design Pattern For Abstraction**. Disponível em: <https://www.codeproject.com/articles/25057/simple-example-of-mvc-model-view-controller-design>. Acesso em: 30 abr. 2017.

COPLIEN, James O.. **Advanced C++ Programming Styles and Idioms**. Reading, MA: Addison-Wesley, 1991. 544 p.

COUCH, Justin; STEINBERG, Daniel H.. **Java 2 Enterprise Edition Bible**. New York: Hungry Minds, Inc., 2002. 705 p.

DANTAS, Alexandre *et. al.* Suporte a Padrões no Projeto de Software. In: XVI Simpósio Brasileiro de Engenharia de Software, 2002, Gramado – RS. **Suporte a Padrões no Projeto de Software.** Rio de Janeiro - RJ: [s.n.], 2002. p. 450-455. Disponível em: <http://www.lbd.dcc.ufmg.br/bdbcomp/>. Acesso em: 24 fev. 2017.

DEITEL, Harvey M.; DEITEL, Paul J.; SANTRY, Sean E.. **Advanced Java 2 Platform:** How To Program. 1 ed. New Jersey: Prentice Hall, 2002. 1496 p.

FARIA, Thiago. **Java EE 7 com JSF, PrimeFaces e CDI.** [S.l.]: AlgaWorks, 2013. 199 p.

FORD, Neal. **Art Of Java Development**. Greenwich: Manning, 2007. 627 p.

FOWLER, Martin. **Padrões de Arquitetura de Aplicações Corporativas**. 1 ed. Porto Alegre: Bookman, 2006. 492 p.

GAMMA, E. *et. al.*. **Padrões de Projeto:** soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000. 370 p.

GILART-IGLESIAS, Virgilio et al. **A Model For Developing J2EE Applications Based On Design Patterns.** Departamento de Tecnología informática y Computación, Universidad de Alicante, Alicante, 2011. Disponível em: <http://gaia.dtic.ua.es/grupoM/recursos/articulos/IADIS-AC-05.pdf>. Acesso em: 15 abr. 2017.

GONCALVES, Antonio. **Beginning Java EE 7**. 1 ed. New York: Apress, 2013. 710 p.

GUO, Zhiguo. **J2EE Architecture and Patterns in Enterprise Systems**. 2004. 60 p. Tese (Master of Information Science) - University of Tampere, Tempere, 2004. Disponível em:<https://tampub.uta.fi/bitstream/handle/10024/91880/gradu00360.pdf>. Acesso em: 05 mar. 2017.

GUPTA, Abhishek. **Java EE: The Basics**. Disponível em: <https://dzone.com/articles/java-ee-basics>. Acesso em: 11 mar. 2017.

GUPTA, Arun. **Java EE 7 Essentials**. 1 ed. Sebastopol: O'Reilly Media, Inc., 2013. 362 p.

JAMAL, Said**. Pattern-Based Approach for Object Oriented Software Design**. 2003. 209 f. Tese (Mestrado em Ciência da Computação) - Department of Computer Science, K.U.Leuven, Bélgica, 2003. Disponível em: <http://www.cs.kuleuven.be/publicaties/doctoraten>. Acesso em: 03 mar. 2017.

JENDROCK, E. *et. al*. **THE JAVA EE 7 TUTORIAL**: Volume 1. 4 ed. United States: Addison-Wesley, 2014. 696 p.

JOSHI, Rohit. **Java Design Patterns**: reusable solutions to commom problems. [S.l.]: JCG, 2015. 183 p.

KASSEM, Nicholas *et. al.*. **Designing Enterprise Applications**: with the Java 2 Platform, Enterprise Edition. Palo Alto, CA: Sun Microsystems, 2000. 362 p.

KASSEM, Nicholas. **Designing Enterprise Applications With The Java 2 Platform, Enterprise Edition**. Califórnia: Sun Microsystems, Inc., 2000. 362 p.

KAYAL, Dhrubojyoti. **Pro Java™ EE Spring Patterns**: best practices and design strategies implementing java™ EE patterns with the spring framework. 1 ed. New York: Apress, 2008. 345 p.

KOENIG, A., **Patterns and antipatterns,** In: Journal of Object Oriented Programming, mar. – abr., 1995.

LAYKA, Vishal. **Learn Java For Web Development**: Modern Java Web Development. 1 ed. New York: Apress, 2014. 472 p.

LE ROY JUNIOR, Vagner**. Padrões Arquiteturais no Java EE 7.** 2014. 11 f. Curso de Pós-Graduação em Arquitetura de Software Distribuído - Pontifícia Universidade Católica de Minas Gerais, Belho Horizonte - MG, 2014. Disponível em: <http://docplayer.com.br/10137270-Padroes-arquiteturais-no-java-ee-7.html>. Acesso em: 28 fev. 2017.

LUPIANHEZ, Leandro; LUCRÉDIO, Daniel. Utilizando padrões de Projeto JEE no desenvolvimento de aplicações web: um estudo de caso. **Revista Tecnologias, Infraestrutura e Software**, São Carlos, v. 1, n. 1, p. 09-19, jul. 2012. Disponível em: <http://revistatis.dc.ufscar.br/index.php/revista/article/view/10>. Acesso em: 09 mar. 2017.

MALDONADO *et. al.*. **Padrões e Frameworks de Software (Notas Didáticas)**, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, ICMC/USP, São Paulo, SP, Brasil, 2017.

MARINESCU, Floyd. **EJB™ Design Patterns**: Advanced Patterns, Processes, and Idioms. Canadá: John Wiley & Sons, 2002. 283 p.

MARTINS, Cláudio. Padrões de Projeto em Aplicações Web: Desenvolvendo projetos web consistentes baseados em reuso de soluções. **Revista Java Magazine - Gerência de Projetos**, Rio de Janeiro, n. 107, p. 1-10, jan. 2012.

MATOS, Claudio Costa. O Uso de Padrões GOF na Análise e Desenvolvimento de Sistemas. **Revista UNILUS Ensino e Pesquisa**, Santos – São Paulo, v. 13, n. 30, p. 41-53, jan.- mar. 2016. Disponível em: <http://revista.unilus.edu.br/index.php/ruep/article/view/715>. Acesso em: 13 mar. 2017.

MICKA, Pavel; KOUBA, Zdenek. **DAO Dispatcher Pattern**: A Robust Design of the Data Access Layer .Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic, 2013. Disponível em: <https://www.thinkmind.org/download.php?articleid=patterns_2013_1_10_70030>. Acesso em: 24 abr. 2017.

MUELLER, John Paul. **Mining Amazon Web Services**: Building Applications with the Amazon API. 1. ed. San Francisco: Sybex, 2004. 376 p.

PANTALEEV, Alexandar; ROUNTEV, Atanas. Identifying Data Transfer Objects in EJB Applications. **Fifth International Workshop on Dynamic Analysis**, Minneapolis, MN, n. 5, p. 1-7, maio. 2007. Disponível em: < http://ieeexplore.ieee.org/document/4273462/>. Acesso em: 22 abr. 2017.

PAULA FILHO, Wilson De Pádua. **Engenharia de Software**: fundamentos, métodos e padrões. 3 ed. Rio de Janeiro: TLC, 2003. 602 p.

PAWLAK, Renaud; SEINTURIER, Lionel; RETAILLÉ, Jean-Philippe. **Foundations Of AOP For J2EE Development**. New Yoork: Apress, 2005. 353 p.

PEREIRA, Aline de Sousa. **Padrões de Projeto**: uma compilação dos mais utilizados em projetos de software. 2008. 45 f. TCC (Bacharel em Sistemas de Informação) - Faculdade de Minas, Belo Horizonte, 2008. Disponível em: <http://algol.dcc.ufla.br/~terra/publications_files/students/2008_faminas_pereira.pdf>. Acesso em: 13 mar. 2017.

PÉRES JÚNIOR, Valdo Noronha. **Estratégias para a utilização da tecnologia J2EE com a arquitetura de cinco camadas**. 2003. 124 f. Dissertação (Mestrado em Ciência da Computação) - Instituto de Ciências Exatas da Universidade Federal

de Minas Gerais, Belo Horizonte, 2003. Disponível em: <http://homepages.dcc.ufmg.br/~wilson/pesquisa/DissertacaoValdo.pdf>. Acesso em: 26 fev. 2017.

RICHARDSON, Chris. **J2EE design decisions**. Disponível em: <http://www.javaworld.com/article/2071722/enterprise-java/j2ee-design-decisions.html>. Acesso em: 23 abr. 2017.

RIEHLE, Dirk; ZÜLLIGHOVEN, Heinz. **Understanding and Using Patterns in Software Development**. Theory and Practice of Object Systems - Special issue on patterns, New York, v. 2, n. 1, p. 3-13, 1996.

RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. **The Unified Modeling Language Reference Manual.** 2 ed. Boston: Pearson, 2005. 742 p.

SERVICE DESIGN PATTERN. **Data Transfer Object**. Disponível em: <http://www.servicedesignpatterns.com/requestandresponsemanagement/datatransferobject>. Acesso em: 24 abr. 2017.

SINGH, I. *et. al*. **Designing Enterprise Applications with the J2EE Platform**. 2 ed. Califórnia: Addison-Wesley, 2002. 440 p.

SOMMERVILLE, Ian. **Engenharia de Software.** 8 ed. São Paulo: Pearson, 2007. 568 p.

SUN MICROSYSTEMS. **The Java EE 5 Tutorial**: For Sun Java System Application Server 9.1. Disponível em: < https://docs.oracle.com/cd/E19575-01/819-3669/819-3669.pdf>. Acesso em: 30 abr. 2017.

TANAKA, Sérgio Akio; PANSANATO, Marcos H.. **Aplicação de padrões de projeto no desenvolvimento de software**. Revista Terra e Cultura, [S. l.], ano 21, n. 41, p. 47-52, jul. – dez. 2005. Disponível em:<http://web.unifil.br/docs/revista_eletronica/terra_cultura/41/terra e cultura_41-4.pdf>.Acesso em: 26 fev. 2017.

WOJCIECHOWSKI, J. et al. MVC Model, Struts Framework and File upload Issues in Web Applications Based on J2EE Platform. **Proceedings of the International Conference Modern Problems of Radio Engineering, Telecommunications and Computer Science,** Lviv-Slavsko, Ukraine, p. 342-345, fev. 2004. Disponível em: <http://ieeexplore.ieee.org/abstract/document/1365980/>. Acesso em: 21 abr. 2017.

YENER, Murat; THEEDOM, Alex. **Professional Java EE Design Patterns.** Indianapolis, Indiana: Wiley & Sons, Inc., 2015. 264 p.

ANEXOS

ANEXO A

Catálogo de Padrões JEE.

Nome do padrão	Descrição
Intercepting Filter	Viabiliza pré- e pós-processamento de requisições
Front Controller	Oferece um controlador centralizado para gerenciar o processamento de uma requisição
Context Object	Encapsula estado de forma independente de protocolo para compartilhamento pela aplicação
Application Controller	Centraliza e modulariza o gerenciamento de *Views* e de ações
View Helper	Encapsula lógica não-relacionada à formatação
Composite View	Cria uma *View* composta de componentes menores
Service To Worker e Dispatcher View	Combinam *Front Controller* com um *Dispatcher* e *Helpers*. O primeiro concentra mais tarefas antes de despachar a requisição. O segundo realiza mais processamento depois
Business Delegate	Desacopla camadas de apresentação e de serviços
Service Locator	Encapsula lógica de consulta e criação de objetos de serviço
Session Facade	Oculta complexidade de objetos de negócio e centraliza controle
Application Service	Centraliza e agrega comportamento para oferecer uma camada de serviços uniforme
Business Object	Separa dados de negócios e lógica usando modelo de objetos
Composite Entity	Implementa *Business Objects* persistentes combinando *Entity beans* locais e POJOs
Transfer Object	Reduz tráfego e facilita transferência de dados entre camadas
Transfer Object Assembler	Constrói um *Value Object* composto de múltiplas fontes
Value List Handler	Lida com execução de queries, *caching* de resultados, etc.
Data Access Object	Abstrai fontes de dados e oferece acesso transparente aos dados
Service Activator	Facilita o processamento assíncrono para componentes EJB
Domain Store	Oferece um mecanismo transparente de persistência para objetos de negócio
Web Service Broker	Expõe um ou mais serviços usando XML e protocolos *Web*

Fonte: Sun Microsystems (2007).

ANEXO B

Implementação do padrão *Front Controller* para a estratégia *Servlet Front.*

```
public class EmployeeController extends HttpServlet {
    // Initializes the servlet.
    public void init(ServletConfig config) throws
    ServletException {
        super.init(config);
    }

    // Destroys the servlet.
    public void destroy() {}

    /** Processes requests for both HTTP
     * <code>GET</code> and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, java.io.IOException {
        String page;

        /**ApplicationResources provides a simple API
         * for retrieving constants and other
         * preconfigured values**/
        ApplicationResources resource =
            ApplicationResources.getInstance();
        try {

            // Use a helper object to gather parameter
            // specific information.
            RequestHelper helper = new
            RequestHelper(request);

            Command cmdHelper = helper.getCommand();

            // Command helper perform custom operation
            page = cmdHelper.execute(request, response);

        } catch (Exception e) {
            LogManager.logMessage(
                "EmployeeController:exception : " +
                e.getMessage());
            request.setAttribute(resource.getMessageAttr(),
                "Exception occurred : " + e.getMessage());
            page = resource.getErrorPage(e);
        }
        // dispatch control to view
        dispatch(request, response, page);
    }

    /** Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
    throws ServletException, java.io.IOException {
        processRequest(request, response);
    }

    /** Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
    throws ServletException, java.io.IOException {
        processRequest(request, response);
    }

    /** Returns a short description of the servlet */
    public String getServletInfo() {
        return "Front Controller Pattern" +
            " Servlet Front Strategy Example";
    }

    protected void dispatch(HttpServletRequest request,
        HttpServletResponse response,
        String page)
    throws javax.servlet.ServletException,
        java.io.IOException {
            RequestDispatcher dispatcher =
                getServletContext().getRequestDispatcher(page);
            dispatcher.forward(request, response);
        }
}
```

Fonte: Sun Microsystems (2007).

ANEXO C

Implementação do padrão *Session Façade* para *Session Bean* e interfaces *Remote* e *Home*

```
package corepatterns.apps.psa.ejb;

import java.util.*;
import java.rmi.RemoteException;
import javax.ejb.*;
import javax.naming.*;
import corepatterns.apps.psa.core.*;
import corepatterns.util.ServiceLocator;
import corepatterns.util.ServiceLocatorException;

// Note: all try/catch details not shown for brevity.

public class ProjectResourceManagerSession
implements SessionBean {

    private SessionContext context;

    // Remote references for the
    // entity Beans encapsulated by this facade
    private Resource resourceEntity = null;
    private Project projectEntity = null;
    ...

    // default create
    public void ejbCreate()
    throws CreateException {}

    // create method to create this facade and to
    // establish connections to the required entity
    // beans
    // using primary key values
    public void ejbCreate(
        String resourceId, String projectId, ...)
    throws CreateException, ResourceException {

        try {
            // locate and connect to entity beans
            connectToEntities(resourceId, projectId, ...);
        } catch (...) {
            // Handle exceptions
        }
    }

    // method to connect the session facade to its
    // entity beans using the primary key values
    private void connectToEntities(
        String resourceId, String projectId)
    throws ResourceException {
        resourceEntity = getResourceEntity(resourceId);
        projectEntity = getProjectEntity(projectId);
        ...
    }

    // method to reconnect the session facade to a
    // different set of entity beans using primary key
    // values
    public resetEntities(String resourceId,
        String projectId, ...)
    throws PSAException {

        connectToEntities(resourceId, projectId, ...);
    }

    // private method to get Home for Resource
    private ResourceHome getResourceHome()
    throws ServiceLocatorException {
        return ServiceLocator.getInstance().getHome(
            "ResourceEntity", ResourceHome.class);
    }

    // private method to get Home for Project
    private ProjectHome getProjectHome()
    throws ServiceLocatorException {
        return ServiceLocator.getInstance().getHome(
            "ProjectEntity", ProjectHome.class);
```

Implementação do padrão *Session Façade* para *Session Bean* e interfaces *Remote* e *Home* (Continuação).

```
    }

    // private method to get Resource entity
    private Resource getResourceEntity(
        String resourceId) throws ResourceException {
        try {
            ResourceHome home = getResourceHome();
            return (Resource)
            home.findByPrimaryKey(resourceId);
        } catch (...) {
            // Handle exceptions
        }
    }

    // private method to get Project entity
    private Project getProjectEntity(String projectId)
    throws ProjectException {
        // similar to getResourceEntity
        ...
    }

    // Method to encapsulate workflow related
    // to assigning a resource to a project.
    // It deals with Project and Resource Entity beans
    public void assignResourceToProject(int numHours)
    throws PSAException {

        try {
            if ((projectEntity == null) ||
                (resourceEntity == null)) {

                // SessionFacade not connected to entities
                throw new PSAException(...);
            }

            // Get Resource data
            ResourceTO resourceTO =
                resourceEntity.getResourceData();

            // Get Project data
            ProjectTO projectTO =
                projectEntity.getProjectData();
            // first add Resource to Project
            projectEntity.addResource(resourceTO);
            // Create a new Commitment for the Project
            CommitmentTO commitment = new
            CommitmentTO(...);

            // add the commitment to the Resource
            projectEntity.addCommitment(commitment);

        } catch (...) {
            // Handle exceptions
        }
    }

    // Similarly implement other business methods to
    // facilitate various use cases/interactions
    public void unassignResourceFromProject()
    throws PSAException {
        ...
    }

    // Methods working with ResourceEntity
    public ResourceTO getResourceData()
    throws ResourceException {
        ...
    }

    // Update Resource Entity Bean
    public void setResourceData(ResourceTO resource)
    throws ResourceException {
        ...
    }

```

Implementação do padrão *Session Façade* para *Session Bean* e interfaces *Remote* e *Home* (Continuação).

```
    // Create new Resource Entity bean
    public ResourceTO createNewResource(ResourceTO resource) throws ResourceException {
        ...
    }

    // Methods for managing resource's blockout time
    public void addBlockoutTime(Collection blockoutTime)
    throws RemoteException, BlockoutTimeException {
        ...
    }

    public void updateBlockoutTime(
        Collection blockoutTime)
    throws RemoteException, BlockoutTimeException {
        ...
    }

    public Collection getResourceCommitments()
    throws RemoteException, ResourceException {
        ...
    }

    // Methods working with ProjectEntity
    public ProjectTO getProjectData()
    throws ProjectException {
        ...
    }

    // Update Project Entity Bean
    public void setProjectData(ProjectTO project)
    throws ProjectException {
        ...
    }

    // Create new Project Entity bean
    public ProjectTO createNewProject(ProjectTO project)
    throws ProjectException {
        ...
    }

    ...

    // Other session facade method examples

    // This proxies a call to a Transfer Object Assembler
    // to obtain a composite Transfer Object.
    // See Transfer Object Assembler pattern
    public ProjectCTO getProjectDetailsData()
    throws PSAException {
        try {
            ProjectTOAHome projectTOAHome = (ProjectTOAHome)
            ServiceLocator.getInstance().getHome(
                "ProjectTOA", ProjectTOAHome.class);
            // Transfer Object Assembler session bean
            ProjectTOA projectTOA =
                projectTOAHome.create(...);
            return projectTOA.getData(...);
        } catch (...) {
            // Handle / throw exceptions
        }
    }

    // These method proxies a call to a ValueListHandler
    // to get a list of projects. See Value List Handler
    // pattern.
    public Collection getProjectsList(Date start,
            Date end) throws PSAException {
            try {
                ProjectListHandlerHome projectVLHHome =
                    (ProjectVLHHome)
                ServiceLocator.getInstance().getHome(
                    "ProjectListHandler",
                    ProjectVLHHome.class);
                // Value List Handler session bean
                ProjectListHandler projectListHandler =
```

Implementação do padrão *Session Façade* para *Session Bean* e interfaces *Remote* e *Home* (Continuação).

```
                    projectVLHHome.create();
                return projectListHandler.getProjects(
                    start, end);
            } catch (...) {
                // Handle / throw exceptions
            }
        }

        ...

        public void ejbActivate() {
            ...
        }

    public void ejbPassivate() {
        context = null;
    }

    public void setSessionContext(SessionContext ctx) {
        this.context = ctx;
    }

    public void ejbRemove() {
        ...
    }
}

The remote interface
for the Session Facade is listed in Example 8.16.

Example 8.16 Implementing Session Facade - Remote Interface

package corepatterns.apps.psa.ejb;

import java.rmi.RemoteException;
import javax.ejb.*;
import corepatterns.apps.psa.core.*;

// Note: all try/catch details not shown for brevity.

public interface ProjectResourceManager
extends EJBObject {

    public resetEntities(String resourceId,
        String projectId, ...)
    throws RemoteException, ResourceException;

    public void assignResourceToProject(int numHours)
    throws RemoteException, ResourceException;

    public void unassignResourceFromProject()
    throws RemoteException, ResourceException;

    ...

    public ResourceTO getResourceData()
    throws RemoteException, ResourceException;

    public void setResourceData(ResourceTO resource)
    throws RemoteException, ResourceException;

    public ResourceTO createNewResource(ResourceTO resource)
    throws ResourceException;

    public void addBlockoutTime(Collection blockoutTime)
    throws RemoteException, BlockoutTimeException;

    public void updateBlockoutTime(Collection blockoutTime)
    throws RemoteException, BlockoutTimeException;

    public Collection getResourceCommitments()
    throws RemoteException, ResourceException;

    public ProjectTO getProjectData()
    throws RemoteException, ProjectException;
```

Implementação do padrão *Session Façade* para *Session Bean* e interfaces *Remote* e *Home* (Continuação).

```
301
302     public void setProjectData(ProjectTO project)
303     throws RemoteException, ProjectException;
304
305     public ProjectTO createNewProject(ProjectTO project)
306     throws RemoteException, ProjectException;
307
308     ...
309
310     public ProjectCTO getProjectDetailsData()
311     throws RemoteException, PSAException;
312
313     public Collection getProjectsList(Date start,
314         Date end) throws RemoteException, PSAException;
315
316     ...
317 }
```

```
1  package corepatterns.apps.psa.ejb;
2
3  import javax.ejb.EJBHome;
4  import java.rmi.RemoteException;
5  import corepatterns.apps.psa.core.ResourceException;
6  import javax.ejb.*;
7
8  public interface ProjectResourceManagerHome
9  extends EJBHome {
10
11     public ProjectResourceManager create()
12     throws RemoteException, CreateException;
13     public ProjectResourceManager create(String resourceId, String projectId, ...)
14     throws RemoteException, CreateException;
15 }
```

Fonte: Sun Microsystems (2007).

ANEXO D

Código de Implementação do padrão *Model-View-Controller* para a construção de uma calculadora.

```
static class Program {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main() {
        // Note: The view should not send to the model but it is often useful
        // for the view to receive update event information from the model.
        // However you should not update the model from the view.
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        frmCalcView view = new frmCalcView();
        CalculatorModel model = new CalculatorModel();
        CalcController controller = new CalcController(model, view);
        Application.Run(view);
    }
}
/// <summary>
/// The controller process the user requests.
/// Based on the user request, the Controller calls methods in the View and
/// Model to accomplish the requested action.
/// </summary>
class CalcController: IController {
    ICalcModel model;
    ICalcView view;

    public CalcController(ICalcModel model, ICalcView view) {
        this.model = model;
        this.view = view;
        this.view.AddListener(this); // Pass controller to view here.
    }

    public void OnClick(int number) {
        view.Total = model.SetInput(number).ToString();
    }

    public void OnAdd() {
        model.ChangeToAddState();
    }
}
```

```
/// <summary>
/// Windows Form that will host our MVC based functionality.
///
/// </summary>
public partial class frmCalcView: Form, ICalcView {
    IController controller;
    public frmCalcView() {
        InitializeComponent();
    }
    /// <summary>
    /// The view needs to interact with the controller to pass the click events
    /// This could be done with delegates instead.
    /// </summary>
    /// <param name="controller"></param>
    public void AddListener(IController controller) {
        this.controller = controller;
    }
    private void lbl_Click(object sender, EventArgs e) {
        // Get the text out of the label to determine the letter and pass the
        // click info to the controller to distribute.
        controller.OnClick((Int32.Parse(((Label) sender).Text)));
    }
    private void lblPlus_Click(object sender, EventArgs e) {
        controller.OnAdd();
    }

    #
    region ICalcView Members
    public string Total {
        get {
            return textBox1.Text;
        }
        set {
            textBox1.Text = value;
        }
    }#
    endregion
}
```

Código de Implementação do padrão *Model-View-Controller* para a construção de uma calculadora (Continuação).

```
/// <summary>
/// Calculator model, The model is independent of the user interface.
/// It doesn't know if it's being used from a text-based, graphical, or web interface
/// This particular model holds the state of the application and the current value.
/// The current value is updated by SetInput
/// </summary>
class CalculatorModel: ICalcModel {
    public enum States {
        NoOperation,
        Add,
        Subtract
    };
    States state;
    int currentValue;
    public States State {
        set {
            state = value;
        }
    }
    public int SetInput(int number) {
        if (state == States.NoOperation) {
            currentValue = number;
        } else if (state == States.Add) {
            currentValue = Add(currentValue, number);
        }
        return currentValue;
    }
    public void ChangeToAddState() {
        this.state = States.Add;
    }
    public int Add(int value1, int value2) {
        return value1 + value2;
    }
    public int Subtract(int value1, int value2) {
        throw new System.ApplicationException(" Not implemented yet");
    }
}
```

Fonte: Code Project (2008).