Universidade Estadual de Londrina

VINÍCIUS FERNANDES DE JESUS

ARQUITETURA DE SOFTWARE: UMA PROPOSTA PARA A PRIMEIRA APLICAÇÃO

LONDRINA–PR

2013

VINÍCIUS FERNANDES DE JESUS

ARQUITETURA DE SOFTWARE: UMA PROPOSTA PARA A PRIMEIRA APLICAÇÃO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Profa. Dra. Jandira Guenka Palma

LONDRINA–PR

2013

Vinícius Fernandes de Jesus
Arquitetura de software: uma proposta para a primeira aplicação/ Vinícius Fernandes de Jesus. – Londrina–PR, 2013-
45 p. : il. (algumas color.) ; 30 cm.

Orientador: Profa. Dra. Jandira Guenka Palma

– Universidade Estadual de Londrina, 2013.

1. Arquitetura de Software. 2. Projeto de Software. 3. Processo de Software. I. Jandira Guenka Paulo. II. Universidade Estadual de Londrina. III. Faculdade de Computação. IV. Proposta de um processo de construção da arquitetura de software para aprendiz

CDU 02:141:005.7

VINÍCIUS FERNANDES DE JESUS

ARQUITETURA DE SOFTWARE: UMA PROPOSTA PARA A PRIMEIRA APLICAÇÃO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Profa. Dra. Jandira Guenka Palma
Universidade Estadual de Londrina
Orientador

Prof. Dr. Segundo Membro da Banca
Universidade/Instituição do Segundo Membro da Banca

Prof. Msc. Terceiro Membro da Banca
Universidade/Instituição do Terceiro Membro da Banca

Londrina–PR, 24 de novembrode 2013

LONDRINA–PR

2013

AGRADECIMENTOS

Agradeço à minha família, por todo suporte durante todos esse anos, sem ela nenhum passo dessa jornada seria dado.

Também agradeço à minha orientadora, que me fez apreciar ainda mais o tema, me proporcionou grandes oportunidades e contribuiu muito para minha evolução.

Agradeço a equipe da empresa Guenka Software, por todo o apoio, a experiência fornecida contribuiu muito para realizar esse trabalho.

É com grande orgulho que agradeço aos meu professores de graduação que me deram suporte durante todos esses anos, em especial aos professores Daniel e Jandira.

Tenho grande gratidão pela Maria Fernanda, por todo incentivo prestado durante a graduação, principalmente nos momentos mais críticos.

Por último, mas não menos importante, agradeço aos meus amigos, por terem me suportado durante todo esse tempo.

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."
Martin Fowler

DE JESUS, V. F.. **Arquitetura de software: uma proposta para a primeira aplicação**. 45 p. Trabalho de Conclusão de Curso (Graduação). Bacharelado em Ciência da Computação – Universidade Estadual de Londrina, 2013.

RESUMO

Dentro do ciclo de vida de um software, a maior parte dos esforços das empresas se concentram nas etapas posteriores à etapa de construção do software, onde ocorre a manutenção e extensão do mesmo. Assim, visando diminuir os esforços mencionados, a fase de construção da arquitetura do software se torna importante. Porém, o processo de construção da arquitetura de um software é difícil, exigindo um elevado conhecimento e experiência. Desta forma, com base nos conhecimentos obtidos durante o levantamento bibliográfico sobre arquitetura de software, este trabalho apresenta uma primeira aplicação de arquitetura de software através da construção de um sistema corporativo, que visa facilitar o início na arquitetura de software, além de estimular o uso da mesma.

Palavras-chave: arquitetura de software. projeto de software. padrão de projeto.

DE JESUS, V. F.. **Software Architecture: a proposal for first application**. 45 p. Final Project (Undergraduation). Bachelor of Science in Computer Science – State University of Londrina, 2013.

ABSTRACT

In the software life cycle, most of efforts are concentrated in maintenance phase. Therefore, trying to reduce these efforts, the software architecture construction process becomes more important. However, this process is difficult and requires a high level of knowledge and experience. Hence, through the experiences acquired by the knowledge gained in the bibliographic, this presents a first software architecture application wich facilitates the first contact with software architecture and stimulates the use of it.

Keywords: software architecture. software design. padrão de projeto.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

GOF	Gang of Four
SRP	Single Responsibilty Principle
OCP	Open Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Principle
DIP	Dependency Inversion Principle
MDD	Desenvolvimento Dirigido por Modelos
UML	Unified Modeling Language
DDD	Domain-Driven Design
DSL	Domain-Specific Language
MPI	Monitoramento da Produção Industrial
MES	Manufacturing Execution System
CLP	Controlador lógico programável
OSI	Open Systems Interconnection

SUMÁRIO

1 INTRODUÇÃO

O processo de construção de um software é uma tarefa difícil, que envolve dedicação e alto custo [3]. Porém, construir um software com uma arquitetura robusta, que agregue manutenibilidade, extensibilidade e escalabilidade para o mesmo, é uma tarefa ainda mais difícil [4], exigindo experiência e um alto conhecimento sobre o assunto [1].

Essa dificuldade tem se elevado, devido ao fato de que os softwares estão cada vez mais complexos e que os clientes necessitam que o sistema esteja pronto o quanto antes. Assim sendo, empresas diminuem o tempo da construção do software, adiantando a entrega do mesmo. [5].

Apesar de diminuir o tempo gasto com a construção do software, um sistema, depois de pronto, ainda possuirá um longo tempo de vida, se tornando um sistema legado [5]. Nesta etapa do ciclo de vida de um software se encontra a manutenção e extensão do mesmo, onde empresas, de um modo geral, concentram um alto investimento [5, 1]. Segundo Sommerville, a etapa de manutenção do software consiste em todas as alterações feitas no mesmo após sua entrega [1].

Essas alterações no software sempre serão necessárias, sendo possível dizer que essas mudanças pertencem a 3 (três) tipos de manutenção: [6, 1]

Correção de erros: Correção de erros de escrita de código, que são os de menor custo; erros em termos de projeto de software, um pouco mais caro que o tipo anterior; e, com um maior custo em relação aos tipos de erros anteriores, tem-se os erros de requisitos de software.

Mudança de ambiente: Manutenção feita para adaptar o sistema para um novo ambiente de software, em que ambiente seria um suporte para execução do sistema, como hardware ou sistema operacional.

Mudança de requisito: Mudanças feitas para adaptar novos requisitos devido a alterações na regra de negócio ou de empresas clientes.

De um modo geral, empresas concentram a maior parte dos recursos gastos durante o processo de software na etapa de manutenção [7]. Segundo Pressman, é comum a existência de empresas que concentram de 60% até 70% dos recursos em manutenção de software [6]. Segundo pesquisas recentes, o custo investido na etapa de manutenção é um pouco maior que 60% do custo total do processo de software [8, 9].

Conforme mostrado por Pressman e Sommerville, alguns fatores são apontados como sendo os responsáveis pelo alto custo anteriormente citado [6, 1]:

Arquitetura de software mal estruturada: Durante o desenvolvimento e a manutenção não há uma devida atenção para a arquitetura do sistema.

Código fonte ilegível: Desenvolvedores não se preocupam com a legibilidade do código, tornando difícil a compreensão do mesmo para novos membros da equipe.

Falta (ou inexistência) de documentação: Pelo fato de "atrasar" o desenvolvimento de software, a documentação é deixada de lado. Além disso, há casos em que as documentações são desatualizadas, não condizendo com a implementação.

Rotatividade da equipe: De um modo geral, empresas do mercado de TI possuem uma grande rotatividade de desenvolvedores, o que afeta a equipe de desenvolvimento original do sistema.

Como pode ser observado, dentre os fatores que podem dificultar a manutenção e extensão de um software, aumentando assim o custo dessas atividades, está a arquitetura de software [9]. Uma arquitetura de software projetada de forma robusta, condizente com os requisitos do sistema, facilita a compreensão, manutenção e extensão do software; diminuindo o custo para essas atividades [1, 9, 10].

Porém, construir um software com uma arquitetura robusta, que agregue manutenibilidade, extensibilidade e escalabilidade para o mesmo, é uma tarefa difícil, que necessita experiência e um alto conhecimento no assunto [1].

Dessa forma, o objetivo do trabalho consiste em construir a arquitetura de software de um sistema corporativo, com a finalidade de facilitar sua manutenção e extensão.

O trabalho está organizado de uma forma que no capítulo 2 ocorre a revisão bibliográfica, que consiste nos conceitos necessários para compreender a construção da arquitetura do sistema; no capítulo 3, o processo utilizado na construção é apresentado; já no capítulo 4, a construção da arquitetura é descrita; e por fim, o capítulo 5 apresenta a conclusão do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo é divido em três partes, onde a primeira contém os conceitos mais abstratos da arquitetura de software; já a segunda parte apresenta conceitos utilizados para decompor a arquitetura em termos de orientação a objetos e por fim são descritos processos relacionados a construção de arquitetura de software.

2.1 Arquitetura de Software

Arquitetura de sofware é um termo que não possui um consenso geral em sua definição, tornando difícil sua definição [11]. Apesar dessa falta de consenso, pelas definições de Sommerville [1], Ralph Johnson [11] e Grady Booch [12], pode-se interpretar que a arquitetura de um software consiste na estrutura dos componentes do sistema e nas regras de comunicação entre esses componentes [13].

Dessa forma, uma arquitetura de software, quando adequadamente documentada, facilita a compreensão da estrutura de um sistema, evitando a compreensão a partir do código fonte e ajudando nas comunicações com a equipe de desenvolvimento e com clientes [1].

Além disso, a arquitetura de um software permite perceber, de forma rápida, decisões na construção do software que influenciam no sucesso de software [6]. Assim, o desenvolvimento da arquitetura de um sistema afeta fatores como reuso, manutenibilidade, extensibilidade e escalabilidade [1].

2.1.1 Arquiteto de Software

Um arquiteto de software é o responsável por criar a arquitetura de um sistema a partir de seus requisitos, possuindo a tarefa de tomar decisões para organizar a estrutura e a interoperação entre os componentes do sistema [6].

O arquiteto de software sempre está preocupado com o andamento do projeto [11], realizando intensas colaborações, seja no auxílio à equipe de desenvolvimento em lógicas de programação, seja no auxílio à coleta de requisitos com especialistas sobre o domínio do problema [14].

Além dessas atividades, o arquiteto de software também possui a responsabilidade de ser o mentor da equipe de desenvolvimento, tornando-a capaz de tomar decisões importantes relacionadas com a arquitetura do sistema [11].

2.1.2 Estilos de Arquitetura

Estilos de arquitetura de software são maneiras de se estruturar um software [1]. De um modo geral, todo software possui algum tipo de estilo de arquitura.

Esses estilos de arquitetura de software são como estilos de arquitetura na construção civil. Com um estilo de arquitetura definido, um arquiteto civil consegue facilmente identificar a estrutura geral de uma construção, mas não pode prever nada detalhado sobre ela [6]. Desta forma, o estilo de arquitetura de um software fornece informações superficiais sobre como o software é organizado.

Os estilos de arquitetura servem como um modelo para projetar um software. Desta forma, esses estilos funcionam como uma estrutura guia para detalhar os componentes e módulos do software [6].

Há vários tipos de estilos de arquitetura de software, como arquitetura em camadas, piper and filter, modelo de repositórios, arquitetura cliente-servidor, arquitetura orientada a objetos, entre outros [1, 6].

Cada estilo de arquitetura possui uma maneira de organizar o sistema e uma característica que representa quando deve ser utilizado. Essas características, de uma maneira geral, são extraídas dos requisitos não funcionais do software [6]. Vale ressaltar que esses estilos de arquitetura não são mutuamente exclusivos.

A seguir dois estilos de arquitetura são descritos.

2.1.2.1 Arquitetura em camadas

Nesse estilo de arquitetura o sistema é separado em camadas, onde cada camada possui uma funcionalidade bem definida e independente das outras [6].

As camadas se comunicam entre si através de uma interface de comunicação bem definida. De um modo geral, uma camada fornece serviços para sua camada superior e utiliza serviços de sua camada inferior. Porém, há casos em que camadas não adjacentes comunicam diretamente entre si, para que não haja muita demora na comunicação entre elas [15].

Uma vantagem desse estilo de arquitetura é que pode-se facilmente construir um software multiplataforma. Isso se dá pelo fato de que é possível substituir a implementação de uma camada sem afetar nenhuma outra camada. Além disso, também há a facilidade em adicionar novas camadas ao sistema [6].

Uma dificuldade enfrentada ao se utilizar o estilo em questão é que a separação lógica das camadas não é muito evidente e isso pode ser um problema para a qualidade do software [1].

Como exemplo desse estilo de arquitetura considere o padrão de arquitetura MVC

(Model View Controller) [3, 1] e também o modelo de redes OSI (Open Systems Interconnection) [15, 16].

2.1.2.2 Arquitetura de Repositórios

O estilo de arquitetura de repositórios é um estilo em que o software é dividido em vários componentes e esses componentes se comunicam através de dados compartilhados em um repositório central (por exemplo banco de dados) [15].

Desta forma, cada componente do sistema realiza uma leitura e/ou escrita nesse repositório central, respeitando a estrutura em que os dados são compartilhados [6].

Uma vantagem desses estilo é que pode-se adicionar, remover ou modificar um componente sem realizar nenhuma alteração nos outros componentes [1].

Porém, uma desvantagem é que todos os componentes dependem do repositório central. Sendo assim, qualquer erro que ocorra no repositório afetará os outros componentes [1].

O modelo de arquitetura blackboard é um exemplo de uma aplicação do estilo em questão, conforme ilustrado por Shaw e Garlan [15].

2.2 Decomposição da Arquitetura

A decomposição da arquitetura de software consiste em projetar os componentes do sistema, elaborando sua estrutura e interface de comunicação. De forma contrária ao projeto de arquitetura de software, essa decomposição também aborda mais detalhes do projeto do software.

Neste trabalho, a forma de decomposição abordada se limita ao paradigma orientado a objetos. A mesma é feita através do uso de princípios e técnicas do projeto orientado a objetos. Desta forma, entende-se como pré-requisito deste trabalho o conhecimento de conceitos básicos do paradigma em questão, como classes, objetos, herança, polimorfismo, interfaces e composição.

2.2.1 Princípios do Projeto Orientado a Objetos

A seguir, alguns princípios de projeto orientado a objetos serão passados. Esses princípios são utilizados no momento em que se inicia a criação de classes e objetos. Tais princípios são aplicações de importantes fundamentos da orientação a objetos e por isso segui-los agrega qualidade ao projeto do componente.

2.2.1.1 Single Responsibilty Principle

Este princípio, que foi primeiramente descrito por Tom DeMacro [17] e Meilir Page-Jones [18] com o nome de coesão, afirma que uma classe deve possuir uma única responsabilidade, em que esta seria uma possibilidade de mudança [19].

Quando uma classe possui mais de uma responsabilidade, ela automaticamente possui mais de uma razão para sofrer alteração. Assim, mudanças em uma responsabilidade poderá afetar outra responsabilidade distinta. Com isso, uma alteração poderá afetar o sistema sem percepção alguma [19].

2.2.1.2 Open-Closed Principle

Introduzido por Bertrand Meyer, este princípio diz que o módulo de um software deve estar aberto para extensão e fechado para modificação [20].

O termo "aberto para extensão" significa que o comportamento do módulo pode ser expandido. Por exemplo, em um módulo de tradução de texto entre línguas naturais, uma extensão seria adicionar a tradução para uma nova língua (Português, Inglês, etc) [19].

Já o termo "fechado para modificação" refere-se à não modificação a base do módulo, mantendo intacto o que já foi construído até o momento [20].

Em outras palavras, esse princípio denota que para realizar a expansão de um módulo o seu estado atual não deve ser modificado, basta adicionar um novo comportamento que respeite as características desse módulo.

A importância desse princípio se dá pela constante mudança sofrida pelo software durante o seu ciclo de vida [19]. Respeitar o princípio em questão permite que essas mudanças sejam feitas facilmente.

2.2.1.3 Liskov Substitution Principle

Diretamente relacionado às propriedades de orientação a objetos herança e polimorfismo, o princípio em questão herda o nome de sua criadora, Barbara Liskov [21].

Para entender este princípio, considere as classes A e B, onde a classe B herda da classe A. Agora considere o programa P que possui uma referência para o objeto *o1* do tipo A e que faz uma chamada ao método m de *o1*. Segundo o princípio de Liskov, substituir o tipo do objeto *o1* para o tipo B não irá modificar o comportamento do programa [21].

Ao violar este princípio, um programa, ao ser estendido, passará por modificações. Essas modificações deixam o mesmo instável e dificultam sua manutenção. Além disso, o princípio Open Closed também será violado [19].

Em sua essência, este princípio quer dizer que uma subclasse deve respeitar as definições estabelecidas por sua classe pai. Sempre vale lembrar que uma classe herda propriedades e comportamentos de outra classe. Embora isso pareça trivial, sua violação é muito comum [19].

2.2.1.4 Interface Segregation Principle

Associado com o projeto de interfaces de um sistema, esse princípio diz que o cliente de uma interface não pode depender de métodos que ele não faz uso [19].

Em termos de projeto, isso quer dizer que uma interface não deve ser poluída, ou seja, ela deve possuir um pequeno conjunto de funcionalidades [19].

Violar esse princípio pode aumentar o acoplamento de um software, já que funcionalidades destintas se encontram no mesmo lugar e logo existe uma dependência entre elas [19].

2.2.1.5 Dependency Inversion Principle

Este princípio, muito utilizado na construção de *frameworks* [22], diz que um módulo não pode depender de um outro módulo, ambos devem depender de abstrações. Além disso, detalhes devem depender de abstrações, nunca o contrário [19].

O fato de depender de uma abstração torna o módulo mais fácil de se estender, já que basta trocarmos o detalhe (implementação) que respeita essa abstração. Além do mais, depender de abstrações diminui o acoplamento do software [19].

Esse princípio pode ser aplicado sempre que há alguma comunicação entre classes. Para aplicá-lo, Robert C. Martin utiliza algumas heurísticas [19]:

- Nenhuma variável de uma classe deve ter referência para uma classe concreta.
- Nenhuma classe deve ser subclasse de uma classe concreta.
- Nenhum método deve sobrescrever um método implementado pela classe pai.

O princípio em questão se assemelha ao princípio utilizado pelo GOF na construção de design patterns (veja 2.2.2), que diz o seguinte “Programe para uma interface, não para uma implementação” [4].

2.2.2 Design Patterns

Design patterns (padrões de projeto no português) são soluções tradicionais para problemas recorrentes no projeto de software [1]. Essas soluções são genéricas e buscam

fazer o melhor uso de princípios e conceitos de orientação a objetos e geralmente fornecem reuso, flexibilidade e baixo acoplamento [4].

Sua origem vem da arquitetura civil, quando o arquiteto civil Christopher Alexander escreveu o livro "A Pattern Language" que fala sobre os padrões de projeto da construção civil [23, 6]. Foi neste livro que Ralph Johnson, John Vlissides, Richard Helm e Erich Gamma, conhecidos como Gang of Four (GOF), se basearam para popularizar os padrões de projeto de software, em meados da década de 90 [4].

Um benefício do uso de design patterns é que equipes de desenvolvimento de software, ao encararem algum problema no projeto de um sistema, podem reutilizar soluções existentes e então poupar tempo na construção do sistema [3].

Padrões de projeto podem ser aplicados para qualquer linguagem de programação orientada a objetos. Além disso, o uso de padrões de projeto permite que o desenvolvedor explique uma determinada solução apenas citando o nome do padrão utilizado [4].

Apesar de não ser regra, em geral um padrão possui quatro elementos [4]:

Nome: Uma palavra ou duas que descreve o problema, a solução e a consequência de um padrão. O uso de nome em padrões de projeto aumenta o vocabulário de projeto de software e possibilita uma fácil referência aos padrões, podendo ser utilizado em conversas de equipe e documentações.

Problema: Informa quando o padrão deve ser aplicado. Em geral, descreve o problema de projeto e seu contexto.

Solução: Representa os elementos do projeto (classes, interfaces e objetos), as responsabilidades de cada um e o relacionamento entre eles.

Consequências: São os resultados da aplicação do padrão de projeto. Esses resultados geralmente são dados em relação a espaço utilizado, tempo de resposta, flexibilidade, extensibilidade e portabilidade. A análise desses resultados ajuda a ponderar na escolha entre alternativas de padrões de projeto.

Os padrões de projeto apresentados pelo GOF são classificados através de dois critérios [4]. O primeiro critério diz respeito ao propósito do padrão. Cada padrão pode ter um dos três propósitos:

Criacional: Quando o padrão está relacionado com a criação de objetos.

Estrutural: Padrão que se refere a composição de classe e objetos.

Comportamental: Aborda os padrões que se referem a comunição entre objetos e classes.

O segundo critério de classificação é o escopo do padrão, que pode ser dois valores:

Classe: Quando o padrão se refere a relação entre classes e subclasses, conhecido como herança.

Objeto: Tipo de padrão que se refere a relação entre objetos, ou seja, se diz respeito a composição dos mesmos.

A lista dos padrões do GOF e sua classificação é exibida na tabela 1.

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

Tabela 1 – Classificação dos padrões

2.2.2.1 Padrões de Arquitetura

Assim como no projeto de software, a ideia de padrões pode ser aplicada para o projeto de arquitetura, com uso de padrões de arquitetura.

Padrões de arquitetura são soluções para problemas no projeto de arquitetura, tendo uma característica mais geral do que padrões de projeto [3].

Em geral, esse padrões ajudam a estruturar a arquitetura do sistema, auxiliam a construir o estilo de arquitetura do sistema [1, 6].

2.3 Processos do projeto de arquitetura

Nesta seção, uma lista de processo de contrução da arquitetura de software é apresentada.

Nas duas subseções a seguir serão apresentadas ideias sobre o processo do projeto de um software, que engloba desde a construção do estilo de arquitetura do software até a decomposição de seus módulos, segundo o autor Sommerville[1]. Em sequência, o processo proposto por Kaartinen, Palviainen e Koskimies[2] é apresentado.

Primeiramente será explicado um modelo geral do processo de projeto de software. Em seguida, será mostrado um conjunto das atividades mais importantes no processo de projeto de software utilizando orientação a objetos[1].

2.3.1 Modelo Geral do Processo de Projeto de Software

O autor exibe um processo geral para se projetar um software, ilustrado pela figura 1. Note que esse processo é independente do tipo de processo de software utilizado.

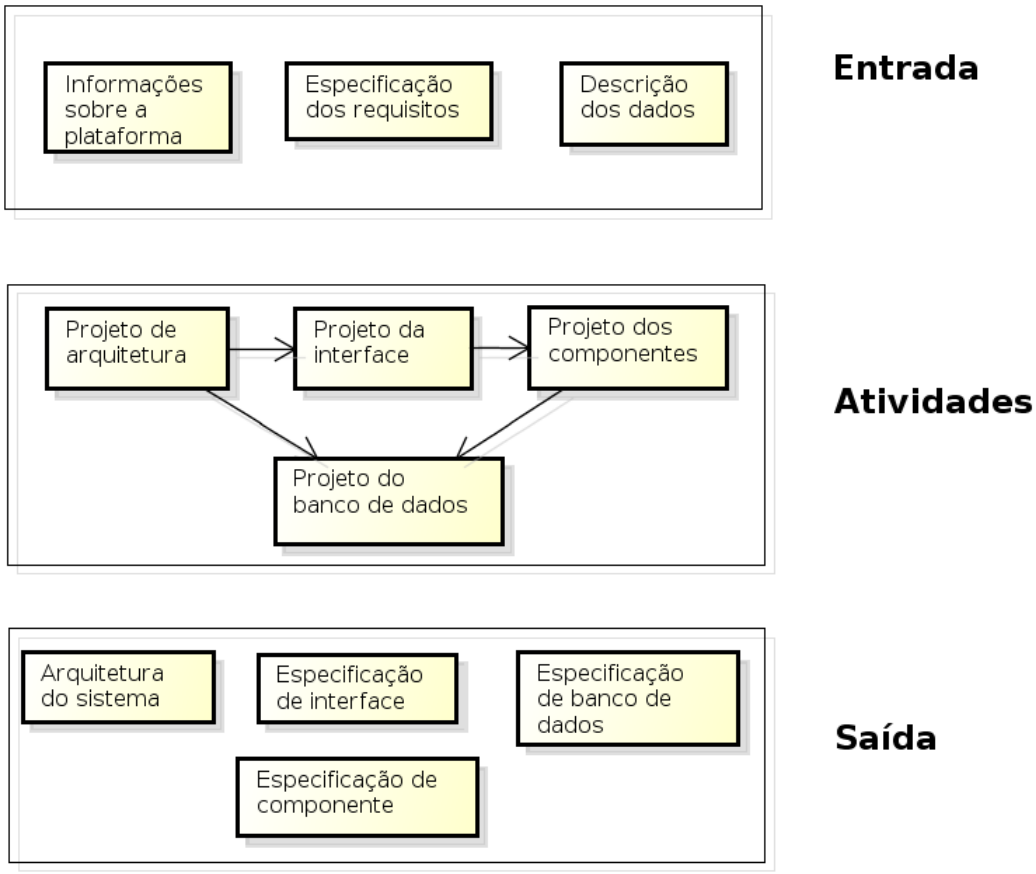

Figura 1 – Processo geral de projeto de software. Figura retirada de Sommerville[1].

Primeiramente, tem-se as entradas das atividades do projeto de software, fomentadas pela coleta dos requisitos. Essas entradas são descritas da seguinte forma:

Informações sobre a plataforma: Informações sobre o ambiente em que o software vai ser executado. Esse ambiente é composto por softwares que interagem com o sistema, como banco de dados e sistemas operacionais.

Especificação dos requisitos: São especificações sobre o que o software deve fazer.

Descrição dos dados: Descrições sobre os dados utilizados no sistema, como por exemplo dados cadastrais.

O autor descreve quatro atividades dentro do projeto de software, sendo projeto de arquitetura, projeto da interface, projeto dos componentes e projeto do banco de dados.

O projeto de arquitetura é o momento de definição da estrutura geral do sistema (estilos de arquitetura 2.1.2), dos principais componentes e de seus relacionamentos.

A segunda atividade, projeto de interface, consiste em definir a interface de comunicação utilizada pelos componentes. Respeitar essa interface de comunicação permite que o projeto de cada componente seja feito de maneira independente.

A terceira atividade consiste em elaborar cada componente do sistema, de forma a projetar como o componente irá operar. Esse projeto pode ser a construção de um novo componente ou a adaptação de um componente existente.

A última atividade, projeto do banco de dados, consiste em projetar a estrutura de dados do sistema e como essas estruturas serão representadas no banco de dados.

Conforme ilustrado, essas quatro atividades produzem algumas saídas da etapa de projeto. A estrutura de cada saída varia de forma considerável. Segundo Sommerville, essas saídas podem ser representadas com um conjunto de diagramas, caso a abordagem do MDD (Desenvolvimento Dirigido por Modelos) for utilizada. No caso do uso de métodos ágeis, essas saídas podem ser representadas dentro do código.

2.3.2 Projeto de Software Orientado a Objetos

De uma forma geral, um processo de projeto de software orientado a objetos possui algumas importantes atividades, sendo elas:

Contexto do sistema e interações: Essa atividade consiste em compreender a relação do software que está sendo projetado com o seu ambiente externo. Isso ajuda a estruturar a maneira como o software vai interagir com outros sistema e quais funcionalidades ele ficará responsável.

Projeto da arquitetura: É nesta atividade que a arquitetura do sistema será definida, identificando os principais componentes do sistema e o relacionamento entre eles. Para defini-la, são utilizados conceitos sobre o domínio do problema (coletados na etapa anterior) e conceitos sobre arquitetura de software, como estilos de arquitetura (2.1.2) e padrões de arquitetura (2.2.2.1).

Identificar objetos do sistema: Consiste em reconhecer as principais classes de objetos do sistema, através das informações sobre o sistema. Ao analisar informações como funcionalidade e propriedade de um módulo, ideias são formadas sobre possíveis objetos do sistema.

Projetar modelos: Consiste em criar modelos, como diagramas UML, que representam partes do projeto de sistema, como a estrutura dos componentes, o fluxo de interação entre os objetos e os estados dos objetos.

Especificar interfaces: Consiste em especificar a interface de comunicação entre os componentes. Essa interface tem que ser bem projetada, pois futuras mudanças podem causar um grande retrabalho.

Essas atividades são dependentes uma das outras e variam a forma de execução de acordo com o sistema que será projetado.

2.3.3 Pattern-Driven Process

A subseção em questão se concentrará no trabalho realizado por Kaartinen, Palviainen e Koskimies[2], onde é feita a proposta de um processo genérico de projeto de software dirigido por padrões de projeto. Esse processo se baseia nos fatores de qualidade de um sistema, geralmente retirados dos requisitos, em um catálogo de padrões de projeto existente e em uma arquitetura base.

Como entrada do processo, é assumida a especificação dos requisitos e uma arquitetura de software base. Uma arquitetura base compreende a estrutura de alto nível do sistema, funcionando como um framework que permite inserir detalhes do sistema no projeto, sendo extraída dos requisitos de software e podendo ser projetada com o uso de padrões de arquitetura (2.2.2.1) e estilos de arquitetura (2.1.2).

Tal processo contém quatro fases, as quais são análise (Analysis), priorização (Prioritization), realização (Realization) e avaliação (Evaluation); que são ilustradas na figura 2 e descritas a seguir.

2.3.3.1 Análise (Analysis)

A fase de análise é uma importante fase, tendo em mente que as outras apenas completam e asseguram o que foi obtido na análise. Essa etapa consiste em refinar os requisitos em relação as preocupações com o fator de qualidade e associar soluções para cada requisito de qualidade.

Por exigir um alto conhecimento sobre o domínio do problema, e também sobre familiaridade com soluções de projeto de software, a etapa em questão é considerada muito criativa, o que torna interessante trabalhos colaborativos entre os membros da equipe de desenvolvimento de software.

Para encontrar essas soluções, o catálogo de padrões de projetos são utilizados. Este catálogo é constituído de padrões de projetos convencionais e também de alguns obtidos durante a aplicação processo.

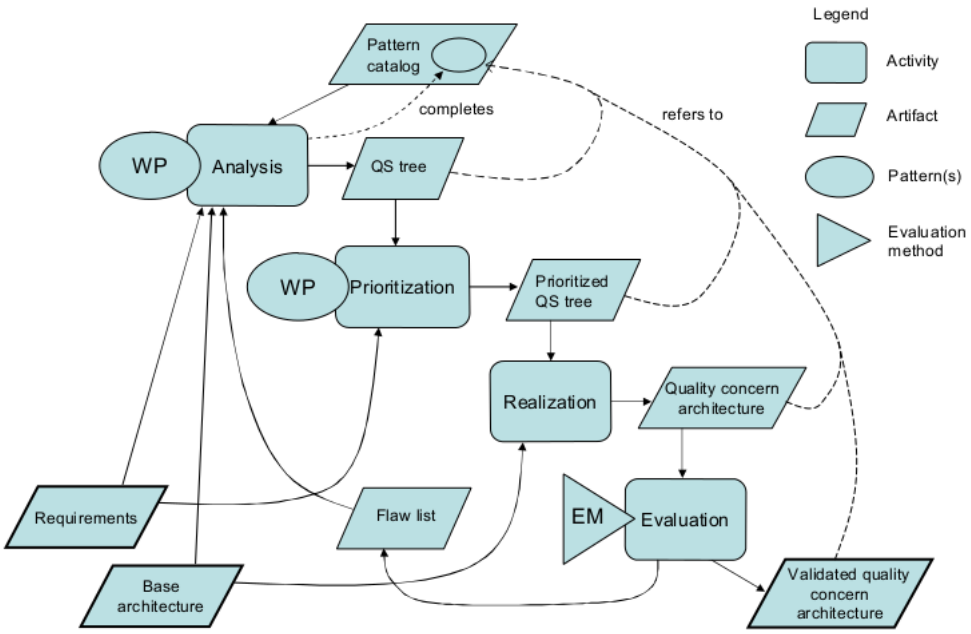

Figura 2 – Ilustração do modelo proposto por Kaartinen, Palviainen e Koskimies[2]

Como saída da análise, uma árvore QS (Quality-Solution) é obtida, que consiste em uma árvore, representada em forma de tabela para facilitar a representação, que contém uma relação entre os requisitos de qualidade, sua importância para o sistema, a solução, seus efeitos. A árvore QS resultante nessa fase é similar a mostrada na figura 3, mas com a última coluna (Prio) vazia.

2.3.3.2 Priorização (Prioritization)

Posterior a análise, está a fase de priorização, que consiste em filtrar os requisitos de qualidade refinados na etapa anterior e as solução, obtidas também na fase anterior. Note que um requisito de qualidade pode possuir mais de uma solução e por isso a necessidade da segunda filtragem dita. Uma árvore QS com priorização é obtida como saída da priorização.

A priorização dos requisitos é uma atividade que varia muito entre os sistemas, pois depende do que é julgado como importante no mesmo (num acordo entre os stakeholders). Assim, requisitos deixados para trás são geralmente menos importantes ou possuem soluções cujas consequências impactam negativamente no sistema.

2.3.3.3 Realização (Realization)

O objetivo desta etapa consiste em mapear as soluções com maiores prioridades, obtidas anteriormente na árvore QS, à arquitetura base existente e customizá-las para o sistema alvo. Essa customização torna-se necessária a partir do momento em que os padrões de projeto são genéricos e precisam ser adapitados ao sistema.

Quality requirement	Refined quality requirement	Prio	Scenario	Solution	Liabilities	Prio
Changeable UI	Extensible menus	D	Add a new menu command in one day	Observer	Performance(L) Usability(L)	
	Changeable look&feel	N	Change into MS look&feel in one month	MVC	Performance(M)	
	Localizable UI	D	Introduce new language in one week	Localization Table	Performance(L)	
OS portability		Q	Port to Linux in 2 months	Interface		
Functional modifiability	Changeable renting procedure	D	Change reservation policy in one day	Strategy	Performance(L)	1
				Template Method	Modifiability(M)	2
	Changeable charging	D	Add new credit card support in one month	Interface	Security(H)	
Data modifiability	Changeable rented item	D	Introduce new rented item in one week	Interface		
	Changeable customer data	D	Introduce new customer attribute in one day	Metatype	Performance(H) Security(M)	

Figura 3 – Exemplo de uma árvore QS, retirado de Kaartinen, Palviainen e Koskimies[2]

O resultado da realização é uma descrição de arquitetura, que consiste em algum documento que contenha as decisões de projeto que atendem aos requisitos de qualidade em questão.

2.3.3.4 Avaliação (Evaluation)

Por fim, há a etapa de avaliação, cuja funcionalidade está em determinar se a solução obtida na etapa anterior satisfaz o requisito em questão. Para obter esse resposta, o requisito deve ter um cenário que indique se ele está ou não completo.

Caso a solução não estiver adequada, o problema é documentado e adicionado em uma lista de erros. O processo continua até que a lista de falhas seja esvaziada.

3 PROCESSO UTILIZADO NA CONSTRUÇÃO DA ARQUITETURA DE UM SISTEMA CORPORATIVO

Conforme mostrado na figura 4, o processo utilizado para construir a arquitetura do sistema corporativio possui 4 etapas, sendo elas:

- Entender o propósito do software.
- Projeto da arquitetura.
- Definição de interfaces.
- Detalhar arquitetura.

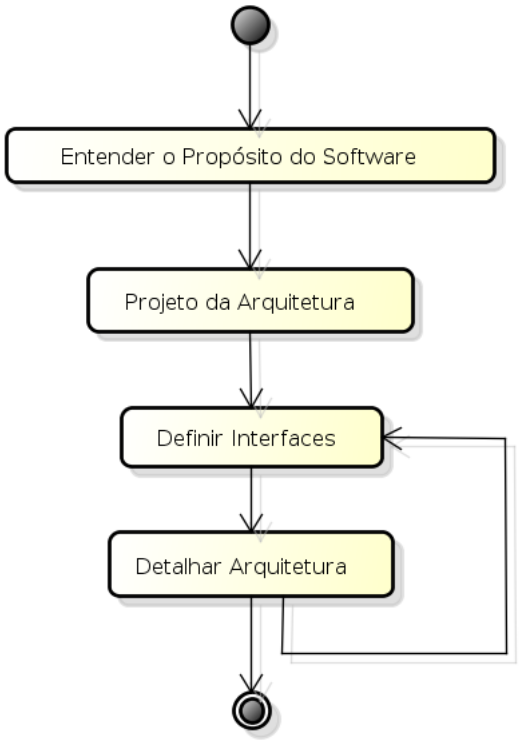

Figura 4 – Diagrama de atividades do processo proposto.

As duas primeiras são as mais abstratas, sendo distantes da orientação a objetos e programação. Já as duas últimas etapas, que são feitas de maneira incremental, onde surgem as classes do sistema e estão diretamente relacionadas com programação.

3.1 Entender o Propósito do Software

Entenda essa etapa como sendo o ponto de partida para se construir a arquitetura do software, visto que a mesma é uma representação da funcionalidade do mesmo.

Neste momento, o objetivo é identificar as características fundamentais do software, conhecer as funcionalidades essenciais do mesmo, ou seja, deve-se conhecer o domínio em que o sistema se encontra. Para isso, é necessário utilizar algum processo de análise de requisitos, como abordado por Sommerville[1] e Pressman[6][1]. Colocando de uma maneira mais simplista, é necessário entender o que software deve fazer.

A etapa em questão, conforme ilustrado na figura 5, é subdividida em três atividades, sendo elas:

Coletar: Consiste em coletar informações sobre o software, obtendo-as através de conversas com (possíveis) usuários e com especialistas no domínio do problema. Ao reconstruir a arquitetura de um sistema legado, também há a possibilidade de realizar a coleta pelo contato com o sistema em seu estado atual.

Processar: Ocorre a análise das informações coletadas anteriormente, que consiste em extrair as funcionalidades essenciais do software. É nessa atividade, feita pela equipe de desenvolvimento, que as funcionalidades do software são compreendidas.

Validar: Verifica se o software foi compreendido de maneira correta. Essa validação deve ser feita com clientes do sistema ou especialista no assunto. Se a resposta da validação for positiva, a etapa "Entender o Propósito do Software" é concluída, caso contrário a atividade de coleta deve ser retomada.

3.2 Projeto da Arquitetura

Após compreender as funcionalidades do software, vem a etapa de projetar a arquitetura do mesmo, através do uso de estilos e padrões de arquitetura, conforme descritos em 2.1.2 e 2.2.2.1[2], respectivamente.

A arquitetura a ser projetada servirá como guia no desenvolvimento do software e deve condizer com as características essenciais do mesmo. Desta forma, é necessário considerar, ao realizar a etapa em questão, as informações coletadas na etapa anterior.

A arquitetura, daqui proveniente, tem o objetivo de representar os elementos mais importantes do software e o fluxo essencial entre estes elementos. Desta forma, a arqui-

[1] Em Sommerville[1], o assunto é abordado no capítulo 4 e em Pressman[6] nos capítulos 5, 6 e 7.
[2] Para uma abordagem mais detalhada sobre estilo de arquitetura veja Garlan e Shaw[15] no capítulo 3, já para padrões de arquitetura veja o livro Fowler[3].

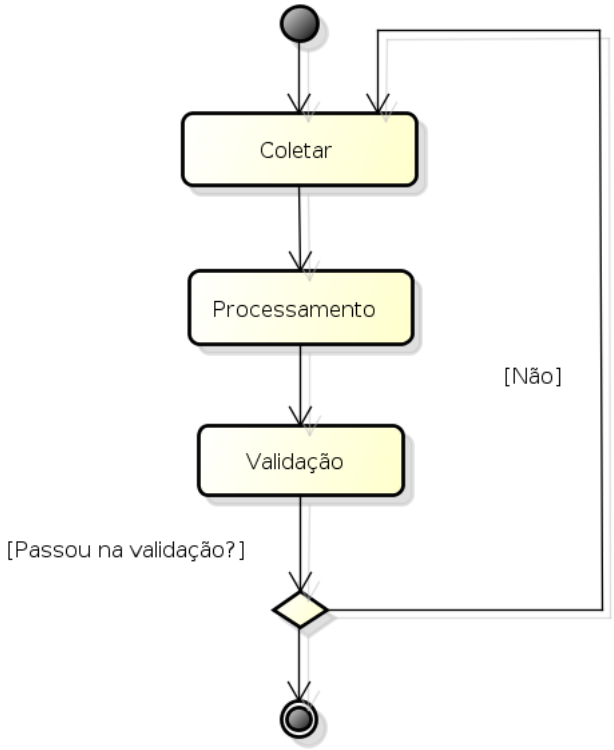

Figura 5 – Entender o Propósito do Software - digrama de atividades.

tetura deve servir como um guia para entender, de maneira macro, os passos necessários para a execução do software.

A figura 6 exibe as três atividades contidas nessa etapa, que podem ser descritas como segue:

Extrair elementos do software: Identificar os elementos que compõem o fluxo essencial do software. Um exemplo é um elemento responsável em comunicar com o banco de dados ou responsável por se comunicar com a impressora.

Relacionar elementos: Consiste em projetar a interação dos elementos identificados na etapa anterior. Em geral, no fluxo do software é possível identificar quais elementos se comunicam. Para auxiliar em como estruturar essa comunição, estilos e padrões de arquitetura devem ser utilizados.

3.3 Definir Interfaces

Com o projeto de arquitetura feito, tem-se que definir a interface de comunicação entre os elementos identificados na fase anterior. Assim, como mostrado na figura 7, essa etapa possui duas atividades:

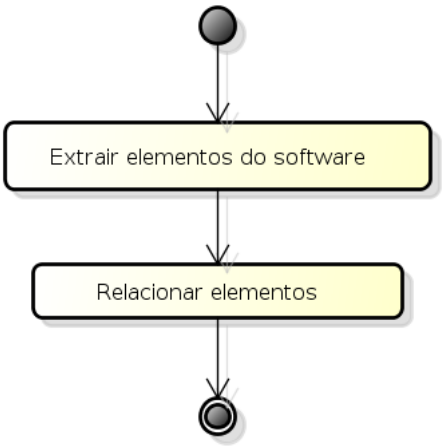

Figura 6 – Projeto da Arquitetura - digrama de atividades.

Identificar protocolo: Consiste em identificar os dados que são transmitidos entre os elementos, durante o fluxo essencial do software. Um exemplo de dado trocado pode ser um número inteiro ou um texto.

Projetar classes: Com os dados identificados, pode-se derivar classes através da análise desses dados. Como as classes criadas servirão para representar dados, é comum que os resultados sejam estruturas de dados [24] ou os padrões de projetos *Data Transfer Object* e *Value Object* [3, 25][3].

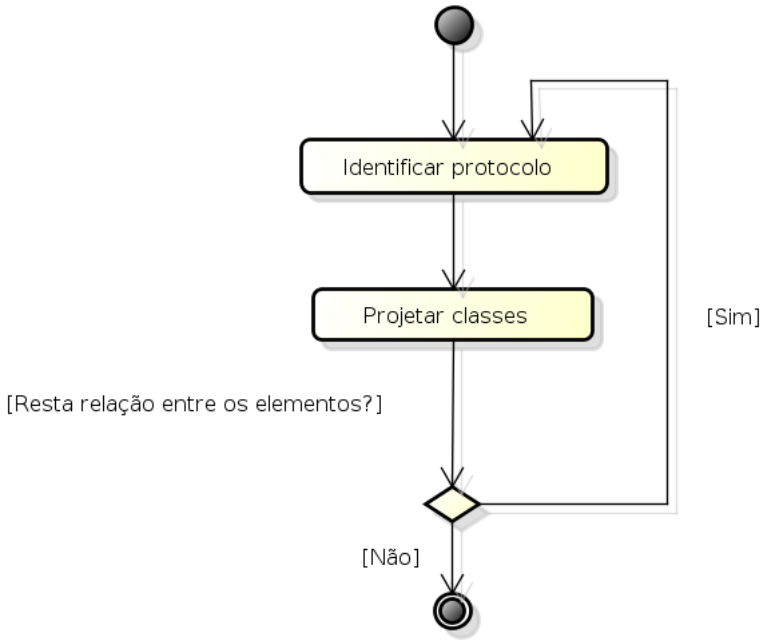

Figura 7 – Definir Interfaces - digrama de atividades.

[3] Para mais detalhes sobre esses padrões, veja o capítulo 6 do livro de Martin[25] e veja no livro de Fowler[3] na página 401 (*Data Transfer Object*) e 486 (*Value Object*).

3.4 Detalhar Arquitetura

Ao se projetar a arquitetura, surgiram elementos importantes do software e, posteriormente, quais informações são trocadas entre eles. Com isso, a etapa em questão tem o objetivo de detalhar cada um desses elementos, onde cada elemento se torna um módulo do sistema.

O detalhamento de um módulo consiste em projetar, em termos de orientação a objetos, o mesmo; ou seja, é nesta etapa que surgem as classes do sistema e a partir de então é possível começar a escrever as linhas de código do mesmo, já que as tarefas anteriores tinham um alto nível de abstração.

Conforme ilustrado na figura 8, a etapa em questão é subdividida em duas atividades:

Analisar requisitos do módulo: Consiste em compreender o que módulo deve fazer e analisar requisitos do módulo que podem influenciar no projeto do mesmo. Por exemplo, em um módulo que acessa banco de dados, foi identificado que o sistema de gerenciamento de banco de dados utilizado pode ser alterado a qualquer momento, o projeto então deve suportar essa mudança.

Projetar classes: Atividade cujo objetivo é, baseado na análise feita na atividade anterior, criar as classes do módulo. Como o projeto do módulo está no escopo de orientação a objetos, pode ser de grande ajuda fazer o uso de princípios de orientação a objetos (2.2.1) e de padrões de projetos (2.2.2) ao executar esta atividade.

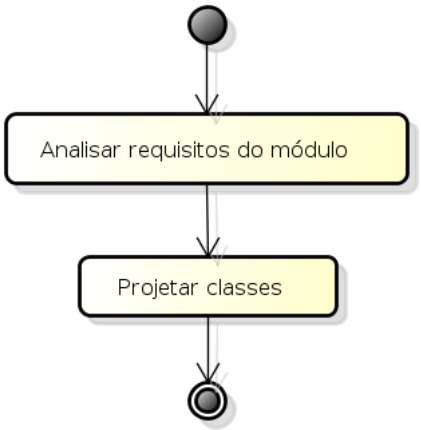

Figura 8 – Detalhar Arquitetura - digrama de atividades.

Uma interessante abordagem ao se projetar os módulos consiste em criar uma espécie de framework da aplicação. O framework seria um conjunto de classes gerais que forma uma estrutura que permite adicionar facilmente a regra de negócio específica. Como

uma dica para a criação dessa estrutura, o princípio *Dependency Inversion Principle* (2.2.1.5) e os padrões de projeto *Template Method* e *Factory Method* são muito úteis nesse escopo. Além do mais, para realizar a modelagem da regra de negócio, há dois livros interessantes, primeiro, um livro de Eric Evans, que mostra a abordagem do DDD (Domain-Driven Design) [26], e outro de Martin Fowler, que fala sobre DSL (Domain-specific language) [27].

Caso essa abordagem for utilizada, é necessário projetar primeiro as classes do framework e depois a extensão do mesmo. Desta forma, a etapa em questão passa a ter quatro atividades (exibidas na figura 9), onde as duas primeiras (em amarelo) são para projetar o framework e as outras duas (em azul) para estendê-lo.

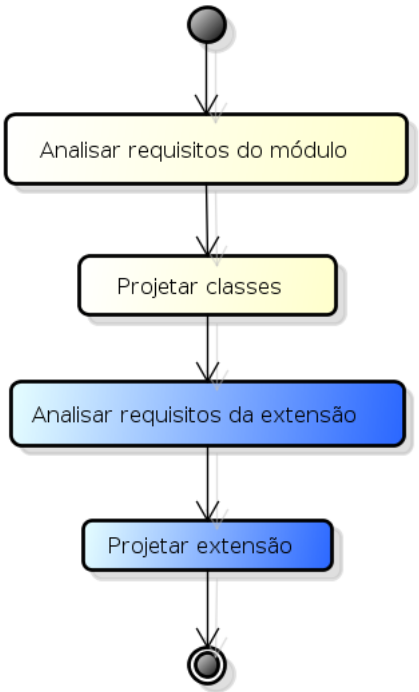

Figura 9 – Detalhar Arquitetura com framework - digrama de atividades.

A ideia dos pares de atividades é a mesma explicada anteriormente, sendo que na primeira os requisitos são analisados e a na segunda ocorre o projeto das classes; a diferença é o escopo do requisitos, onde um se concentra apenas no framework e o outro nos detalhes da extensão do mesmo.

4 ESTUDO DE CASO: CONSTRUÇÃO DA ARQUITETURA DE UM SITEMA CORPORATIVO

Este capítulo possui o estudo de caso deste trabalho, que consiste na construção da arquitetura de software de um sistema corporativo. Na seção 4.1, o sistema escolhido para ser utilizado no estudo de caso é descrito enquanto que a seção 4.2 contém os detalhes da construção.

4.1 Sistema Guenka MPI

O sistema utilizado no estudo de caso foi o Guenka MPI, um software legado do tipo MES (*Manufacturing Execution System*), que possui aproximadamente 195 mil linhas de código-fonte. De um modo geral, esse sistema ajuda a monitorar a produção da indústria, que consiste basicamente em exibir informações do chão de fábrica, sendo utilizado, pela gerência da produção, ao tomar decisões. Suas principais funcionalidades são:

- Coletar dados do chão de fábrica, através de apontamentos feito por operadores.
- Consultar o histórico de apontamentos de produção da indústria.
- Fornecer informações, à gerência da indústria, sobre cada setor da produção.
- Rastrear o andamento da produção, através de controle de lotes e roteiros.

O sistema MPI, conforme mostrado na figura 10, é composto por mais de um sistema, sendo eles:

Servidor MPI: Software responsável por receber dados referentes ao chão de fábrica, através dos apontamentos feito pelos operários, e armazená-los no banco de dados.

Apontador MPI: Sua funcionalidade consiste em fazer apontamentos manuais através do computador e não através de coletores. Esse software é geralmente utilizado para corrigir algum apontamento executado de maneira incorreta por algum operário.

Cliente MPI: É onde a gerência da indústria pode monitorar a produção. Este software fornece acesso as informações sobre o histórico de apontamentos dos operários, ao planejamento da produção, além de dados que permitem o rastreamento de toda produção.

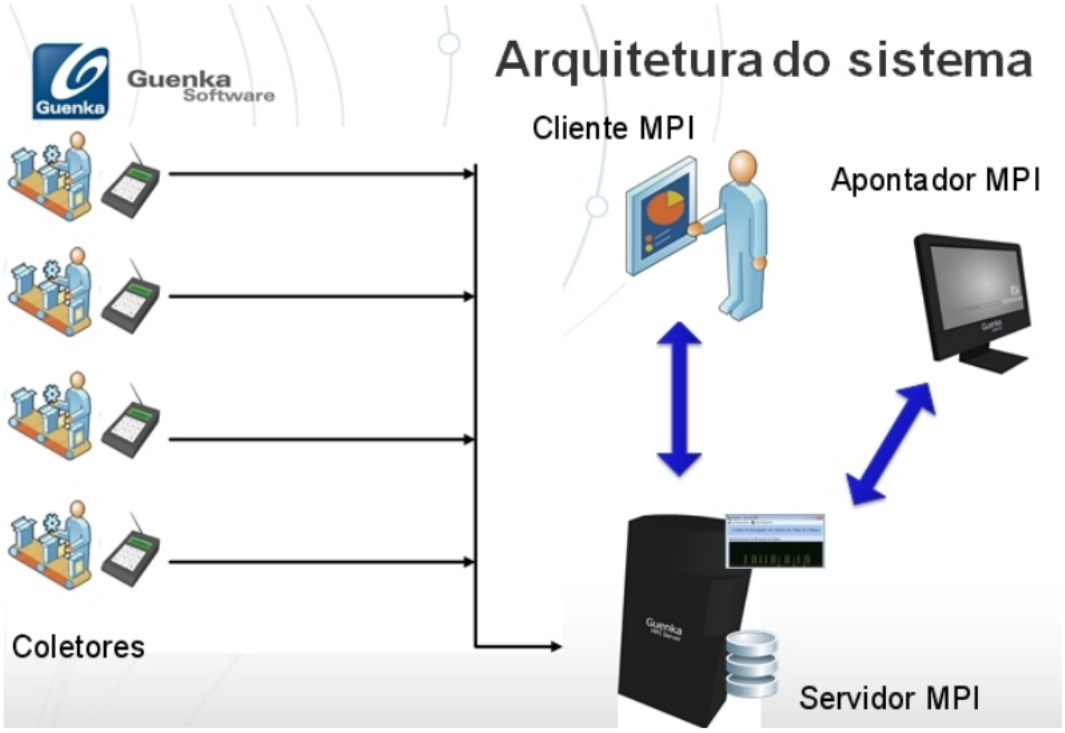

Figura 10 – Estrutura geral do MPI.

Em um aspecto geral, o sistema MPI possui um processo de manutenção muito doloroso, pois ele se tornou um sistema muito instável para mudanças. Como motivo para essa instabilidade, pode-se indicar os aspectos apontados no primeiro capítulo deste trabalho, como pouca e desatualizada documentação, alta rotatividade da equipe de desenvolvimento do sistema e uma arquitetura que não foi planejada. Porém, foi identificado que o Servidor MPI era o sistema com maior instabilidade e por isso esse estudo de caso consiste na construção de uma nova arquitetura para este sistema.

4.2 Construção da Arquitetura

4.2.1 Entender o Propósito do Software

4.2.1.1 Coletar

Por ser um sistema legado, para compreender mais sobre o funcionamento do software é inevitável o contato com o sistema em seu estado antigo. Esse contato foi feito através de correções de erros, tanto no Servidor MPI quanto no cliente MPI, que possibilitou entender o papel do Servidor MPI para com o sistema de uma maneira geral.

Além disso, foi de grande utilidade algumas documentações levantadas durante o processo de reengenharia, que o sistema passara no ano de 2012, e também o contato com membros mais antigos da empresa. Esses dois fatores foram de grande ajuda pois serviram como um guia para entender o sistema.

A execução dessa etapa foi simplificada pela experiência de 6 meses do autor com o sistema, não sendo, portanto, necessário toda a documentação precisa durante a fase de análise de requisitos e todas as etapas do processo de análise de requisitos.

4.2.1.2 Processar

Com o processamento das informações, identificou-se que a ideia geral do Servidor MPI é simples: ele recebe informações do chão de fábrica através de apontamentos feitos por operários; processa esses apontamentos, geralmente envolvendo banco de dados; e, se necessário, retorna o resultado desse processamento para o chão de fábrica.

Para receber informações dos operários, são utilizados os chamados coletores, que são dispositivos simples que facilitam o apontamento, como CLP e micro terminal. Além disso, foi identificado a possibilidade de outros dispositivos serem utilizados, como *smartphone*, *Thin Client*, entre outros.

Os apontamentos, de um modo geral, fornecem informações sobre o que está acontecendo no chão de fábrica, como por exemplo a entrada de um funcionário em um determinado turno de trabalho, ou a informação de peças produzidas em um determinado espaço de tempo por um operador.

O processamento desses apontamentos também é simples, consistindo em verificar se está tudo correto (por exemplo, funcionário existe) e, caso esteja correto, armazenar as informações no banco de dados. A resposta deste processamento, na maioria dos casos, é informada para o coletor.

Além dessas observações levantadas, durante a manutenção do Servidor MPI, foi verificado que, dependendo da indústria, o número de apontamentos pode ser alto ou baixo e, por isso, lógicas diferentes para o controle de carga poderiam ser utilizadas.

4.2.1.3 Validação

Como o Servidor MPI era um sistema legado, a única validação possível era com a equipe de desenvolvimento. Pelo fato de a equipe de desenvolvimento ter participado das atividades anteriores, a validação foi feito ao decorrer dessa etapa.

4.2.2 Projetar Arquitetura

4.2.2.1 Extrair Elementos do Software

Para projetar a arquitetura do Servidor MPI, as informações levantadas na subseção anterior foram utilizadas. Como dito, o fluxo básico do sistema foi identificado, que consiste em receber informações do chão de fábrica (através dos coletores), processá-las e, se necessário, responder ao coletores.

Observando essas informações, pode-se verificar que será necessário se comunicar com os coletores e realizar uma regra de negócio, que provavelmente irá trabalhar com algum banco de dados.

Além dessas observações, também foi levantada a necessidade de ter uma política de balanceamento de carga, que controlaria as informações a serem processadas, sendo uma etapa intermediária entre receber informação dos coletores e processar essas informações. Desta forma, já foram identificados 3 elementos da arquitetura a ser construída.

4.2.2.2 Relacionar Elementos

Lembre-se de que tanto na comunicação com os coletores, na aplicação da regra de negócio, como no balanceamento de carga, há a possibilidade de mudança de implementação; ou seja, é necessário projetar uma arquitetura que possibilite essas mudanças.

Desta forma, com essa possibilidade de mudança e tendo três funcionalidades distintas no software em questão, foi visto que o estilo de arquitetura em camadas (2.1.2.1) se encaixa com a realidade do novo Servidor MPI. Assim, esse estilo de arquitetura foi aplicado, onde cada elemento anteriormente citado deu origem a uma camada. Essas camadas, ilustradas na figura 11, são como segue:

IO: Camada responsável em se comunicar com os coletores, seu nome vem de *input* e *output*.

Thread Manager: Seu objetivo consiste em realizar a política de balanceamento de carga.

Processor: Possui a funcionalidade de processar as informações vindas dos coletores. É nessa camada que a lógica da regra de negócio é aplicada e a comunicação com o banco de dados é feita.

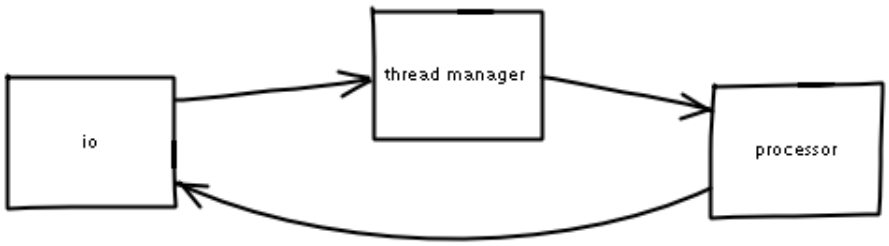

Figura 11 – Estilo de arquitetura do Servidor MPI.

4.2.3 Definir Interfaces

Para definir as interfaces entre as camadas da nova arquitetura do servidor MPI, foi necessário identificar os dados que são passados entre essas camadas.

No caso do sistema em questão, ocorre troca de informação: a partir da camada IO para a camada ThreadManager, da camada ThreadManager para a Processor e da Processor para a IO. Desta forma, cada uma dessas transições deve ser analisada, identificando o protocolo trocado e representando esse protocolo em termos de orientação a objetos. Cada análise é descrita nas subseções a seguir.

4.2.3.1 IO para ThreadManager

Identificar Protocolo: Ao receber dados de um determinado coletor, a camada IO deve repassar esses dados para a camada ThreadManager, para que possam ser processados quando a ThreadManager encaminhar para a camada Processor. Desta forma, da camada IO para a camada ThreadManager é necessária a transição dos dados que contém a informação do apontamento.

Os dados recebidos da camada IO precisam ser traduzidos, visto que os mesmos contém informações específicas da interface de comunicação utilizada pelos coletores. Porém, foi decidido que essa tradução não deve ser feita pela camada IO, para não atrasar o recebimento de novos apontamentos, mas sim feita na camada Processor. Desta forma, a camada IO também deve informar para o ThreadManager como essa tradução deve ser feita, enviando uma espécie de tradutor.

Projetar classes: Como pode ser observado, os dados a serem transitados são abstratos e dependem muito do tipo de coletor utilizado. Desta forma, a escolha da representação dessas informações foi a classe base da linguagem Java, chamada *Object*. Como não há a certeza do formato do dado a ser transitado, tentar adivinhá-lo poderá gerar muito trabalho futuramente, e como a classe *Object* é geral, podendo ser convertida para qualquer outra, essa foi a escolha tomada.

Para representar a tradução, é passado um objeto do tipo *InputTranslator*, que é uma interface com o método *input*, que recebe de parâmetro um objeto do tipo *Object* e o retorna traduzido, representado por um objeto *TranslatedInput*. O diagrama de figura 12 ilustra o que foi dito.

4.2.3.2 ThreadManager para Processor

Pelo fato da camada ThreadManager apenas decidir quando esses dados serão processados, os dados passados para o Processor são os mesmos dados recebidos da camada IO.

4.2.3.3 Processor para Sender

Identificar protocolo: Depois que os dados forem processados, é necessário uma resposta para o coletor que indique qual o próximo passo que operário pode executar. Além disso, é necessário informar para qual coletor a resposta deve ser enviada, visto que o Servidor MPI pode receber dados de um coletor e responder para outro diferente.

Desta forma, da camada Processor para a Sender é passada a mensagem que indica o próximo passo e também qual coletor irá receber a mensagem.

Projetar classes: Para representar a mensagem enviada, foi criada a classe *StateMachineOutput*, que possui o código que identifica o tipo de mensagem. Já para representar o coletor, surgiu a classe *Device*. Ambas as classes são representadas no diagrama de classes da figura 13.

4.2.4 Detalhar Arquitetura

Nesta etapa, que consiste em projetar cada elemento da arquitetura, foi decidido criar uma espécie de framework do Servidor MPI. A justificativa para tal escolha deu-se pelo fato de que o framework poderia facilitar futuras extensões no sistema, como por exemplo suportar um novo tipo de coletor, adicionar uma nova lógica de processamento ou até mesmo modificar a política de balanceamento de carga.

A ideia utilizada para construir o framework foi projetar os módulos da forma mais abstrata possível, sem se prender aos detalhes. Desta forma, cada módulo foi projetado para possuir classes básicas para seu funcionamento, onde as mesmas devem facilitar futuras extensões.

Nesta seção, apenas o projeto do framework em questão será mostrado, já que a extensão do mesmo ainda não foi concluída.

4.2.4.1 IO

Analisar requisitos do módulo: Conforme dito anteriormente, a camada IO é a camada responsável por se comunicar com os coletores, recebendo e enviando informações para eles. Assim, pelo fato de poder variar os tipos de coletores suportados pelo sistema, o módulo em questão precisa tornar fácil a tarefa de suportar novos tipos de dispositivos.

Projetar classes: O projeto desse módulo foi dividido em duas partes, sendo uma para receber os dados vindos dos coletores e a outra para enviar dados para eles, onde cada uma das partes serão explicas a seguir.

Na primeira parte, foi criada uma classe abstrata chamada *Listener*, responsável por receber os dados dos coletores e encaminhá-los ao ThreadManager. Assim sendo, para adicionar um tipo de coletor é necessário estender a classe Listerner, colocando, nessa classe estendida, os detalhes deste tipo de coletor. O diagrama de classes da classe em questão é ilustrado na figura 12.

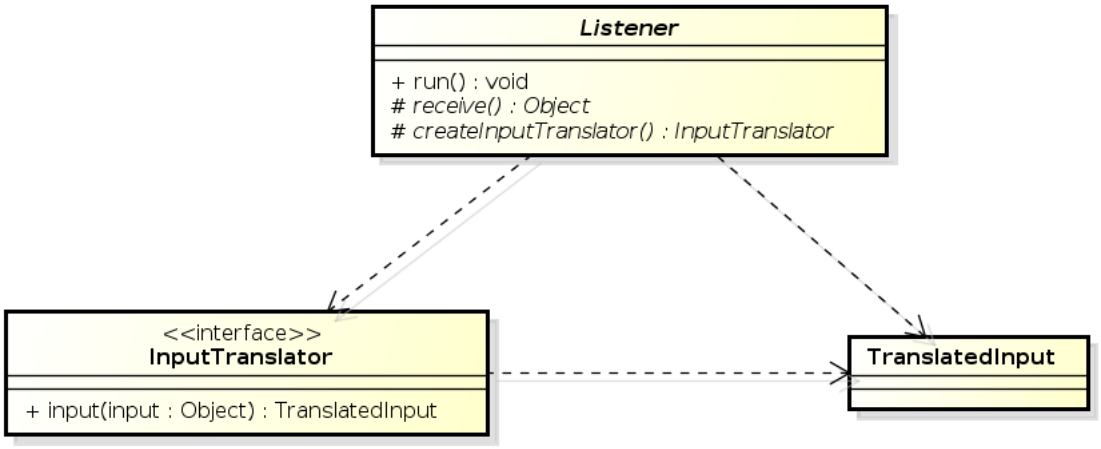

Figura 12 – Diagrama da classe Listener.

Para que a classe *Listener* possa, de forma independente do tipo de coletor, determinar os passos necessários para receber dados dos coletores, o padrão de projeto *Template Method* [4] foi utilizado, onde o método *run* é quem determina o algoritmo modelo e o método *receive* é o método abstrato que deverá ser implementado pela subclasse. O método *receive* deve conter a lógica para identificar a chegada de novos dados e então retorná-los. O algoritmo, contido no método *run*, consiste em receber os dados (chamando o método *receive*) e encaminhá-los ao ThreadManager.

O retorno do método *receive* contém as informações enviadas pelos coletores, mas esses dados estão no protocolo do tipo de coletor e precisam ser traduzidos para que o sistema possa interpretá-los. Dessa forma, viu-se a necessidade de utilizar um objeto para traduzir essa entrada, onde o mesmo deve implementar a interface *InputTranslator* (exibida na figura 12).

Como a classe *Listener* deve encaminhar as informações dos coletores o mais rápido possível, foi decidido que a tradução dos dados não fosse feita na classe *Listener*, mas sim no módulo Processor. Para que isso seja possível, é necessário encaminhar para o ThreadManager, juntamente com os dados, o objeto que irá fazer a tradução. Porém, com isso surge um problema: como fazer para instanciar o objeto tradutor, sem que a classe Listener saiba de nenhum detalhe? A solução foi utilizar o padrão de projeto *Factory Method* [4] através do método *createInputTranslator*, que obriga a criação do tradutor na subclasse de *Listener*.

Já na segunda parte, a classe abstrata *Sender*, cujo diagrama de classe é representado na figura 13, é quem possui o fluxo necessário para enviar dados para os coletores, através do método *send*. Primeiramente, este método traduz, para uma linguagem que o coletor entenda, os dados que irão ser enviados e depois esses dados traduzidos são enviados para o coletor destino. O passo de enviar os dados para o coletor destino é feito pelo método abstrato *sendMessage*, que deve ser implementado pela subclasse de *Sender*, contendo os detalhes de como conversar com o tipo de coletor. A classe em questão faz o uso do padrão de projeto *Template Method*, através do método *send*.

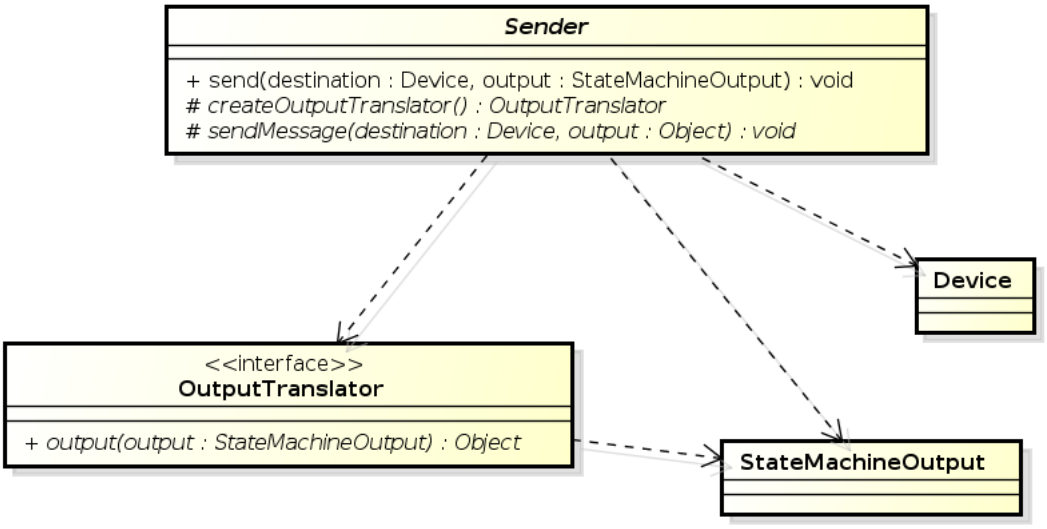

Figura 13 – Diagrama da classe Sender.

A tradução é feita por um objeto que implemente a interface *OutputTranslator*. Para a criação desse objeto também foi utilizado o padrão de projeto *Factory Method*, através do método *createOutputTranslator*.

Observe que o projeto das classes *Listener* e *Sender* respeita o princípio *Open-Closed* e, além disso, caso um módulo dependa dessas classes, o mesmo irá respeitar o princípio da Inversão de Dependência.

4.2.4.2 Thread Manager

Analisar requisitos do módulo: O módulo Thread Manager é responsável por fazer o balanceamento de carga, que, em outras palavras, consiste em escolher a thread que irá processar as informações vindas dos coletores. Pelo fato de que em cada empresa o fluxo de carga é diferente, o módulo em questão deve facilitar adicionar novas políticas de balanceamento de carga.

É importante lembrar que o ThreadManager recebe do módulo IO as informações vindas dos coletores e o tradutor das mesmas, devendo repassá-los para o módulo

Processor.

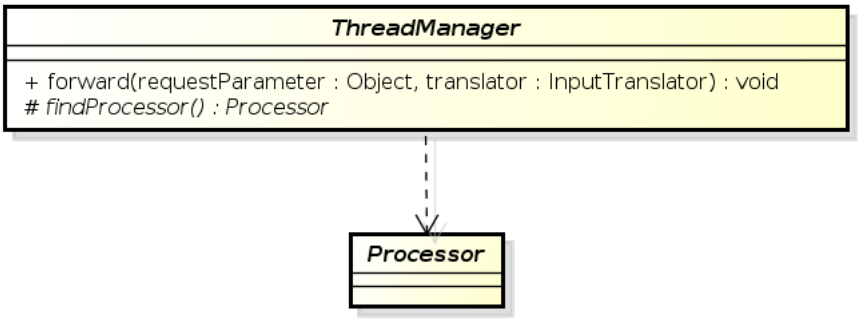

Figura 14 – Diagrama da classe ThreadManager.

Projetar classes: Os dados dos coletores são passados para esse módulo através da chamada do método *forward*, da classe *ThreadManager*, que é ilustrada pelo diagrama da figura 14. Este método escolhe a thread que irá processar os dados vindos do coletor, onde essa thread é do tipo *Processor*. Com o *Processor* escolhido, o mesmo é invocado, sendo passado como parâmetro os dados e o tradutor.

Para ser extensível e respeitar o princípio *Open-Closed*, o método em questão faz uso do padrão de projeto *Template Method*, com a escolha do objeto *Processor* sendo feita através do método abstrato *findProcessor*, que contém a lógica da política de balanceamento de carga. Desta forma, para adicionar uma nova política, é necessário estender a classe *ThreadManager* e sobrescrever o método em questão.

4.2.4.3 Processor

Analisar requisitos do módulo: O módulo em questão é responsável por processar as informações vindas dos coletores. Além desse processamento, o módulo deve tornar fácil as tarefas de adicionar novas lógicas de processamento e trocar as lógicas existentes.

Projetar classes: No módulo Processor, seus clientes tem contato com a classe abstrata *Processor*, como visto na descrição da classe *ThreadManager*.

Essa classe, conforme ilustrado na figura 15, possui um método abstrato chamado *process*, que recebe como parâmetro um objeto do tipo *TranslatedInput* e representa os dados vindos do coletor traduzidos, ou seja, não contendo nenhuma informação específica da interface de comunicação do coletor. O método *process* é chamado durante a execução do método *run*, que é um *Template Method*. Vale ressaltar que o método *run*, para traduzir os dados de entrada, faz o uso do tradutor passado pelo *ThreadManager*.

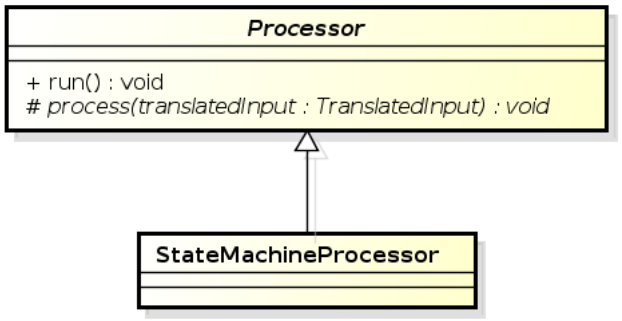

Figura 15 – Diagrama da classe Processor.

Assim, basta estender a classe *Processor* para adicionar uma lógica de processamento. Para fazer isso, no Servidor MPI foi criada a classe *StateMachineProcessor*.

O motivo de criar essa classe surgiu da ideia de fazer uma máquina de estados para cada coletor, onde essa máquina de estados representa a lógica de processamento. Na verdade, no antigo Servidor MPI já havia algo do tipo, mas cuja extensão e manutenção eram muito custosas. Desta forma, na classe *StateMachineProcessor*, a implementação do método *process* consiste em invocar a máquina de estados do coletor requisitante. Para facilitar alterações na máquina de estados, como adicionar ou remover estados e mudar a ordem dos estados, o padrão de projeto *State* [4] foi utilizado.

5 CONCLUSÃO

Durante o ciclo de vida de um software, empresas detentoras do mesmo concentram a maior parte dos recursos durante a fase de manutenção. Dentro dos fatores que influenciam o alto custo desta fase, está a arquitetura de software mal estruturada, pelo fato de que empresas durante o desenvolvimento e a manutenção não concentram a devida atenção para a arquitetura do sistema.

A engenharia de software permite que empresas construam software condizente com os requisitos do cliente, com devida documentação e processo de implantação, porém não há ênfase em como construir uma arquitetura de software que facilite o reuso e que seja simples de estender e modificar.

Desta forma, o trabalho em questão se concentra em como construir uma arquitetura de software que possua os fatores anteriormente citados através da exploração de temas como princípios de orientação a objetos e padrões de projeto, contribuindo para se projetar um software cuja arquitetura torne viável o reuso, extensão e modificação.

Ao decorrer do trabalho realizado foi possível verificar a dificuldade em compreender o tema arquitetura de software, pois o mesmo mostrou-se abstrato, não sendo abordado com frequência em meios acadêmicos, e que requer prática. Desta forma, por conta desses fatos citados, a dificuldade ao ingressar nos estudos sobre arquitetura de software é evidente e isso pode influenciar o não uso da mesma.

Com a construção da arquitetura de um sistema corporativo, além de trazer melhorias para o mesmo, foi visto que o termo arquitetura de software, a partir de agora, está fazendo parte da cultura da empresa, pois, além de externalizar os benefícios do assunto, o trabalho mostrou os meios que tornaram a construção da arquitetura mais aplicável.

REFERÊNCIAS

1 SOMMERVILLE, I. *Software engineering*. Boston: Pearson, 2011. ISBN 978-0137035151.

2 KAARTINEN, J.; PALVIAINEN, J.; KOSKIMIES, K. A pattern-driven process model for quality-centered software architecture design–a case study on usability-centered design. In: *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*. [S.l.: s.n.], 2007. p. 17–26. ISSN 1530-0803.

3 FOWLER, M. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321127420.

4 GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

5 CHANG, C.-H.; LU, C.-W.; HSIUNG, P.-A. Pattern-based framework for modularized software development and evolution robustness. *Information and Software Technology*, v. 53, n. 4, p. 307 – 316, 2011. ISSN 0950-5849.

6 PRESSMAN, R. *Software engineering : a practitioner's approach*. New York: McGraw-Hill Higher Education, 2010. ISBN 978-0073375977.

7 TIAKO, P. Maintenance in joint software development. In: *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*. [S.l.: s.n.], 2002. p. 1077–1080. ISSN 0730-3157.

8 REN, Y. et al. Research on software maintenance cost of influence factor analysis and estimation method. In: *Intelligent Systems and Applications (ISA), 2011 3rd International Workshop on*. [S.l.: s.n.], 2011. p. 1–4.

9 SU, C.-T.; YEH, D. Software architecture recovery and re-documentation tool of a hospital information system. In: *Computer and Communication Engineering (ICCCE), 2012 International Conference on*. [S.l.: s.n.], 2012. p. 143–146.

10 WU, B. Modeling software maturity: A software life cycle management approach. In: *Information Science and Technology (ICIST), 2012 International Conference on*. [S.l.: s.n.], 2012. p. 716–720.

11 FOWLER, M. Design - who needs an architect? *Software, IEEE*, v. 20, n. 5, p. 11–13, 2003. ISSN 0740-7459.

12 BOOCH, G. *Object-oriented analysis and design with applications*. Upper Saddle River, NJ: Addison-Wesley, 2007. ISBN 978-0201895513.

13 VALIPOUR, M. et al. A brief survey of software architecture concepts and service oriented architecture. In: *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*. [S.l.: s.n.], 2009. p. 34–38.

14 BOOCH, G. The professional architect. *Software, IEEE*, v. 29, n. 1, p. 12–13, 2012. ISSN 0740-7459.

15 GARLAN, D.; SHAW, M. *An Introduction to Software Architecture*. Pittsburgh, PA, USA, 1994.

16 TANENBAUM, A. S.; WETHERALL, D. J. *Computer Networks (5th Edition)*. [S.l.]: Prentice Hall, 2010. ISBN 0132126958.

17 DEMARCO, T. *Structured Analysis and System Specification*. [S.l.]: Prentice Hall, 1979. ISBN 0138543801.

18 PAGE-JONES, M. *Practical Guide to Structured Systems Design (2nd Edition)*. [S.l.]: Prentice Hall, 1988. ISBN 8120314824.

19 MARTIN, R. C. *Agile Software Development, Principles, Patterns, and Practices*. [S.l.]: Prentice Hall, 2002. ISBN 0135974445.

20 MEYER, B. *Object-Oriented Software Construction (Book/CD-ROM) (2nd Edition)*. [S.l.]: Prentice Hall, 2000. ISBN 0136291554.

21 LISKOV, B. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 23, n. 5, p. 17–34, jan. 1987. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/62139.62141>.

22 FOWLER, M. *Inversion of Control*. http://martinfowler.com/bliki/InversionOfControl.html. Acessado em 28/07/2013.

23 ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. *A Pattern Language: Towns, Buildings, Construction (Cess Center for Environmental)*. [S.l.]: Oxford University Press, 1977. Hardcover. ISBN 0195019199.

24 GOODRICH, M. T.; TAMASSIA, R.; MOUNT, D. M. *Data Structures and Algorithms in C++*. [S.l.]: Wiley, 2011. ISBN 0470383275.

25 MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. [S.l.]: Prentice Hall, 2008. ISBN 0132350882.

26 EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. [S.l.]: Addison-Wesley Professional, 2003. ISBN 0321125215.

27 FOWLER, M. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. [S.l.]: Addison-Wesley Professional, 2010. ISBN 0321712943.