CENTRO PAULA SOUZA
GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso Superior de Bacharelado em Análise de Sistemas e Tecnologia da Informação

ESTUDO DO PADRÃO DE PROJETO STATE NO DESENVOLVIMENTO DE SOFTWARES ORIENTADOS A OBJETOS

LUCAS LIMA DE OLIVEIRA

Americana, SP
2015

CENTRO PAULA SOUZA
GOVERNO DO ESTADO DE SÃO PAULO

Faculdade de Tecnologia de Americana
Curso Superior de Bacharelado em Análise de Sistemas e Tecnologia da Informação

ESTUDO DO PADRÃO DE PROJETO STATE NO DESENVOLVIMENTO DE SOFTWARES ORIENTADOS A OBJETOS

LUCAS LIMA DE OLIVEIRA
limma.lc@gmail.com

Trabalho monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Bacharelado em Análise de Sistemas e Tecnologia da Informação da Fatec-Americana, sob orientação da Prof.ª Dr.ª Maria Cristina Aranda.

Área: Desenvolvimento de software.

Americana, SP
2015

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

Oliveira, Lucas Lima de
O48e Estudo do padrão de projeto state no desenvolvimento de softwares orientados a objetos. / Lucas Lima de Oliveira. – Americana: 2015.
85f.

Monografia (Graduação em Tecnologia em Análise de Sistemas e Tecnologia da Informação). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.
Orientador: Prof. Dr. Maria Cristina Aranda

1. Desenvolvimento de software 2. Sistemas orientados a objetos I. Aranda, Maria Cristina II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.3.05
681.3.04

Lucas Lima de Oliveira

ESTUDO DO PADRÃO DE PROJETO STATE NO DESENVOLVIMENTO DE SOFTWARES ORIENTADOS A OBJETOS

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Bacharel em Análise de Sistemas e Tecnologia da Informação pelo CEETEPS/Faculdade de Tecnologia – Fatec/ Americana.
Área de concentração: Desenvolvimento de Software.

Americana, 08 de dezembro de 2015

Banca Examinadora:

Maria Cristina Aranda (Presidente)
Doutora
Fatec Americana

Rodrigo Brito Battilana (Membro)
Bacharel
Fatec Americana

Diógenes de Oliveira (Membro)
Mestre
Fatec Americana

AGRADECIMENTOS

A Deus, que me ilumina e me da saúde e força para continuar a trilhar o meu caminho.

A minha mãe, que me deu esta oportunidade, além de apoio necessário para finalizar o curso.

A minha orientadora Prof. Maria Cristina Aranda, por dispor do seu tempo para me auxiliar, por ser paciente, por compartilhar seus conhecimentos e por me apoiar no desenvolvimento deste trabalho.

A Faculdade de Tecnologia de Americana e a todo o seu corpo docente por prover um ambiente de ensino rico em conhecimento, possibilitando que eu me aprofundasse nos temas abordados durante as aulas.

A todos que direta, ou indiretamente, participaram da minha formação e viabilizaram que este sonho fosse realizado.

A cada momento feliz que tive a oportunidade de desfrutar nas dependências da faculdade.

E por último, mas não menos importante, agradeço a todos os colegas que tive a oportunidade de conhecer. Vocês mudaram a minha vida e sempre farão parte da minha memória nos tempos de graduação.

DEDICATÓRIA

Dedico este trabalho a minha Mãe e também a minha orientadora Prof.ª Dr.ª Maria Cristina Aranda, que sempre estiveram presentes ao meu lado. Seu apoio e força me fizeram o homem que eu sou, e, hoje, possibilitam a conclusão de mais um sonho que é a realização deste trabalho.

RESUMO

Durante a etapa de implementação no desenvolvimento de softwares orientados a objetos diversos problemas podem ser encontrados, dentre eles um dos mais comuns ocorre quando é necessário aplicar estados e comportamentos em um objeto, pois a implementação mais convencional nesses casos vem acompanhada do uso de diversas comandos condicionais para checar um atributo referente ao estado desse objeto, de modo a faze-lo executar o comportamento correto. Pensando na forma mais vantajosa para resolver este problema específico desenvolvedores e projetistas de software mais experientes fazem uso do padrão State, que é um padrão de projeto idealizado com o objetivo de implementar estados e comportamentos através do uso de herança e polimorfismo, facilitando assim a compreensão do código-fonte e, consequentemente, tornando-o menos complexo para as posteriores manutenções. Esse trabalho monográfico relata um estudo sobre os fundamentos do padrão State e busca evidenciar a importância da sua aplicação para obter a excelência e a qualidade desejadas para o software. Por meio de uma pesquisa bibliográfica e um estudo de caso, que fez uma análise comparativa entre dois protótipos, esse trabalho mostrou que a aplicação do padrão State na etapa de implementação pode contribuir de modo significativo nos futuros processos de manutenção.

Palavras Chave: Desenvolvimento de Software; Manutenção de Código-fonte; Padrões de Projeto; Padrão State.

ABSTRACT

During the implementation phase of object-oriented softwares sundry problems can be found, among them one of the most regular drawbacks occurs when it's necessary to apply states and behaviors to an object. In these cases the most common implementations comes with the use of a variety of conditions check an attribute regarding to the state of the object, in order to make it execute the correct behavior. Thinking on the best way to solve this specific problem experienced software developers and planners uses the State Pattern, which is a pattern of project idealized with the goal to implement status and behavior through the use of heritage and polymorphism, thus making it easier to understand the source code and making it less complex for future maintenances. This monograph describes the study of the basis of the State Pattern and seeks to enhance the awareness of the importance of its applications in order to attain the excellence and qualities desired for the software. Through a bibliographic research, and a case study that relates a comparative analysis between two prototypes, this work has shown that the application of the State Pattern in the implementation phase can contribute in a significant way for the future maintenance process.

Keywords: Software Development; Source code Maintenance; Design Patterns; State Pattern.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE DIAGRAMAS

LISTA DE TABELAS

1 INTRODUÇÃO

Neste trabalho de conclusão de curso foi abordado como tema principal o padrão State, contudo para que o entendimento seja claro, são necessárias breves explicações sobre padrões de projetos.

A aplicação de padrões de projeto teve sua origem a partir das documentações de padrões usadas em projetos arquitetônicos de edifícios e cidades.

Gamma *et al* (2000) adaptou o conceito arquitetônico ao desenvolvimento de software orientado a objetos, e, como resultado, escreveu o primeiro catálogo de padrões de projeto de software, que ainda hoje é tido como referência para se entender o conceito de padrões de projeto aplicados ao desenvolvimento de software orientado a objetos. Padrões de projeto são definidos por ele como uma solução a problemas específicos em um determinado contexto, de modo que se uma solução resolva um problema, esta mesma solução possa ser reaplicada milhares de vezes sem necessariamente seguir os mesmos passos.

As soluções propostas por esses padrões são, geralmente, respostas a problemas que são frequentemente encontrados durante o desenvolvimento de softwares orientados a objetos (MAGELA, 2006).

A utilização de padrões de projeto é considerada um avanço na área de engenharia de software, pois eles permitem a reutilização de soluções que já foram previamente testadas e aprovadas por desenvolvedores e projetistas de software experientes (THOMAZINI NETO, 2006).

Além disso, os padrões de projeto contribuem com a melhoria da legibilidade do código-fonte e facilitam o entendimento do software, pois a partir da aplicação de deles é possível promover reusabilidade, modularidade e melhorar a manutenabilidade do software. Entretanto, quando utilizados por desenvolvedores e projetistas de software inexperientes podem ocasionar uma complexidade desnecessária no software usando uma solução indevida (GAMMA *et al*, 2000).

O estudo se **justificou** com a prioridade de contribuir com desenvolvedores e projetistas de software, propondo a utilização de padrões de projeto para problemas específicos, e, sobretudo, do padrão State para problemas com objetos que possuem comportamentos e estados. Além dos desenvolvedores e projetistas de software, outros profissionais e estudantes da área também podem tirar alguma

contribuição deste trabalho, conhecendo o conceito de padrões de projeto de modo a utiliza-lo em outras áreas de conhecimento. Este trabalho foi realizado devido ao grande interesse na área, visando o aprendizado e contribuindo com futuros planos profissionais.

O **problema** acontece pelo fato da implementação para gerenciar diferentes comportamentos de um objeto, mais convencional, apresentar uma série de comandos condicionais que checam o estado desse objeto para executar algum comportamento. Em razão disso, tornam mais complexa a operação de manutenção do código-fonte.

A **pergunta** que se buscou responder foi como o padrão State pode solucionar a complexidade da operação de manuteção do código-fonte de softwares orientados a objetos que fazem uso de implementações de estados convencionais.

As **hipóteses** foram: a) Atualmente o padrão State é o mais recomendado para resolver a complexidade da operação de manutenção do código-fonte, então cabe aos desenvolvedores e projetistas de software mais experientes decidir quando é necessário aplica-lo no desenvolvimento do software. b) Dependendo do tamanho e das características do projeto de software é possível que o padrão State não seja adequado, dificultando a solução do problema. c) Apesar de alguns projetos de software conseguirem aplicar o padrão State, aqueles que não conseguem podem personaliza-lo.

O objetivo **geral** consistiu em elaborar um levantamento bibliográfico introduzindo o leitor sobre desenvolvimento de software orientado a objetos e reusabilidade através de padrões de projeto. O foco principal se encontra no padrão State, buscando explicar ao leitor como aplica-lo para acabar com códigos-fonte demasiadamente complexos.

Os **objetivos específicos** foram: a) Explicar o conceito geral dos padrões de projeto, deixando claro como efetuar a detecção, escolha e aplicação de um padrão específico; b) Desenvolver um estudo teórico aplicado ao padrão State, baseado em autores de renome, evidenciando sua aplicabilidade, sua estrutura e suas vantagens e desvantagens; c) Elaborar uma análise comparativa entre dois protótipos, um que aplica o padrão State e outro que não aplica, afim de evidenciar suas principais características e diferenças.

Como **metodologia** para o desenvolvimento desta monografia, foi utilizada, quanto à natureza, a pesquisa aplicada que de acordo com Andrade (2010), gera

conhecimento para aplicação prática, solucionando problemas específicos da área estudada. Com a abordagem do problema, foi feito o uso da pesquisa comparativa que, segundo Andrade (2010), tem o objetivo de analisar as informações dos conteúdos e através disso realizar uma comparação entre suas diferenças e similaridades.

Do ponto de vista dos objetivos, foi utilizado a pesquisa bibliográfica, que segundo Andrade (2010), consiste no levantamento de informações com o intuito de enriquecer a base da pesquisa e o estudo de caso, que coleta informações de um objeto visando o seu conhecimento e detalhamento.

Quanto aos procedimentos técnicos também foi utilizada a pesquisa bibliográfica.

O método utilizado neste trabalho foi o dedutivo, onde para Andrade (2010), significa explicar o conteúdo das premissas, utilizando a cadeia de raciocínio e de uma análise geral das informações, gerando assim uma conclusão.

Esta monografia foi estruturada em **seis** capítulos, onde o **primeiro** capítulo contém a introdução deste trabalho, possibilitando a entrada do **segundo**, que descreve conceitos sobre desenvolvimento de software orientado a objetos, seus processos e modelos de desenvolvimento, além de teorizar reuso de software, já o **terceiro** capítulo conceitua o que são padrões de projeto de uma forma geral, o **quarto** explica de forma mais elaborada os conceitos do padrão State e o **quinto** apresenta o estudo de caso, que consiste de uma análise comparativa entre dois protótipos, um que aplica o padrão State e outro que não aplica, evidenciando suas principais características e diferenças.

As informações conseguidas a partir de estudos realizados no capítulo anterior nos levam ao **sexto** capítulo, reservado às **Considerações Finais**.

2 DESENVOLVIMENTO DE SOFTWARE ORIENTADO A OBJETOS

Para Booch (1994), durante as etapas de desenvolvimento de um sistema de software orientado a objetos, uma sequência de atividades deve ser executada pelo projetista afim de se produzir inúmeros documentos que resultam na produção da aplicação especificada.

Essas atividades são definidas com auxílio de metodologias que atendam ao paradigma de desenvolvimento de software orientado a objetos. Ao fim deste processo, o projetista escolhe a metodologia que melhor se enquadra para a realização do trabalho (FREITAS, 2003).

A principal atividade do projetista de software é identificar aspectos relevantes do mundo real para fins de representação em computador. Isso implica abstrair elementos que de fato existem e representá-los de forma idêntica ou muito parecida no mundo computacional. Se essas abstrações não forem uma expressão direta entre os dois mundos, dificuldades de implementação e manutenção desta solução serão encontradas. Heldman (2005), complementa ao dizer que o processo de se desenvolver um software envolve uma série de fatores que são imprevisíveis, tornando algo difícil de coordenar.

O paradigma orientado a objetos trata o mundo real como se ele se constituisse de objetos autônomos e concorrentes que interagem entre sí, tendo seus próprios estados e comportamentos, semelhantes aos de seus correspondentes que são reais. Assim sendo, o desenvolvimento de sistemas orientados a objetos consituem uma abordagem que minimiza a distância entre os problemas do mundo real e do mundo computacional (BOOCH, 1994).

Com a utilização dos objetos, o projetista de software tem abertura para criar algorítimos que quando executados pelo computador serão capazes de efetuar um mapeamento físico para alguma ação do mundo real, além de possibilitarem que outras pessoas além do projetista de software também possam examiná-lo e interpretá-lo. Portanto, torna-se evidente que quanto mais próxima estiver a abstração dos elementos do mundo real para o computacional, mais fácil serão a compreensão, o desenvolvimento, a manutenção e a confiabilidade da aplicação (FREITAS, 2003).

Porém, é necessário levar em consideração que nem todos os projetistas de software possuem experiência com os processos de desenvolvimento, entendimento

e manutenção de um software orientado a objetos. A consequência disso pode ser uma aplicação cuja especificação não atenda ao solicitado pelo cliente, resultando em dificuldades de manutenção do código-fonte e baixo desempenho da aplicação.

Segundo Peters (2001), a fase de manutenção do software é muito delicada, pois são realizadas diversas alterações em elementos e componentes individuais como classes, métodos, estruturas, dentre outros. Essas alterações resultam em erros no projeto de software original, devido à não atualização das especificações dos trechos de código-fonte afetados. A manutenção deve, portanto, incluir toda a configuração do software, incluindo os documentos gerados durante a etapa de especificação.

Rumbaught (1994) completa que a modelagem orientada a objetos se utiliza de três dimensões para descrever um sistema que possui um grau de complexidade relativamente mais alto. São elas:

- Dimensão estrutural dos objetos.
- Dimensão funcional dos requisitos.
- Dimensão dinâmica do comportamento.

Destaca-se uma importância relevante para a dinâmica do comportamento, devido a sua descrição de comportamento dos objetos que compõe o sistema, que nem sempre retrata de forma clara o comportamento da aplicação em tempo de execução.

Freitas (2003, p. 17), ainda cita que:

> “A partir desta constatação, acredita-se que visualizar os estados dos objetos com a possibilidade de acompanhar as modificações no seu comportamento represente uma diminuição no esforço de entendimento do software avaliado. ”

2.1 PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE

Para Sommerville (2007), um processo de desenvolvimento de software, ou, processo de software, pode ser definido como um conjunto de atividades que tem como finalidade a produção de um produto de software, isso inclui o desenvolvimento do software propriamente dito, com a utilização de uma linguagem de programação.

Percebendo a sua relevância, organizações passaram a ter um espaço para o aprimoramento do processo de software, que tem também a finalidade de excluir técnicas obsoletas e fazer uso das melhores práticas na engenharia de software (SOMMERVILLE, 2007). Ele cita ainda que processos de software evoluem para que possam acompanhar e explorar a capacidade das pessoas e as características específicas de sistemas em desenvolvimento.

A escolha do melhor processo deve ser feita de acordo com o nível de criticidade de cada sistema, optando-se por um processo de desenvolvimento mais estruturado quando o sistema for mais crítico, e um processo de desenvolvimento mais ágil e flexível em sistemas de negócios com requisitos que mudam frequentemente.

Sommerville (2007, p. 43) diz que: “Não existe um processo ideal, e várias organizações desenvolveram abordagens inteiramente diferentes para o desenvolvimento de software. ”

E, apesar de existirem diversos processos de software diferentes, com diversos tipos de abordagens, certas atividades são fundamentais a todos eles (SOMMERVILLE, 2007).

Ainda segundo Sommerville (2007), são consideradas atividades fundamentais:

- Especificação de software: Tem como objetivo definir todas as funcionalidades e restrições sobre a operação do software.

- Projeto e implementação de software: O software que atenda as especificações levantadas na etapa anterior deve ser projetado e seu desenvolvimento é executado.

- Validação de software: Testes realizados com a finalidade de se garantir que o software faça o que foi planejado. É nessa etapa que o cliente faz uma análise do software para saber se ele está de acordo.

- Evolução de software: De extrema importância para o ciclo de vida do software, a evolução de software possibilita que ele atenda às necessidades mutáveis do cliente.

Cada uma das atividades acima listadas compõe também subatividades que possuem funções específicas e, portanto, devem ser desempenhadas por profissionais especializados, tais como:

- Programador;
- Analista de Sistemas;
- Arquiteto de Sistemas;
- Gerente de Projetos;
- Dentre outros.

Os modelos de processos de software auxiliam as equipes de desenvolvimento facilitando a maneira com que as atividades serão desencadeadas no processo de software. Documentos de requisitos são gerados com mais agilidade, viabilizando o planejamento e a codificação e, consequentemente, o cliente fica satisfeito em razão de não haver alterações no orçamento e prazo.

2.2 MODELOS DE PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE

Segundo Sommerville (2007), com o surgimento de projetos de maior porte e complexidade e conforme a informática ocupava seu lugar no cotidiano, as empresas que desenvolviam software foram obrigadas a adotarem metodologias de desenvolvimento. Os problemas mais comuns eram relacionados ao financeiro ou prazos, já que um planejamento adequado não existia, levando à necessidade de organizar o desenvolvimento, ou seja, utilizar metodologias e a partir daí uma série delas foram criadas.

Os três modelos genéricos abordados por Sommerville (2007) serão explicados de maneira breve a seguir.

2.2.1 MODELO EM CASCATA

O modelo em cascata, também conhecido como ciclo de vida do software, foi o primeiro modelo de processo de software publicado.

Este modelo possui basicamente cinco estágios que devem ser desempenhados linearmente, isto é, eles devem respeitar a seguinte ordem: análise e definição de requisitos, projeto de sistema e software, implementação e teste de unidade, integração e teste de sistema, operação e manutenção (SOMMERVILLE, 2007).

Pressman (2002), trata os estágios como se fossem processos, e fornece a seguinte descrição para cada um deles:

- Análise e definição de requisitos: Este processo consiste na compreensão da natureza do sistema que será construído. Para executa-lo são necessários alguns conhecimentos, dentre eles: entender o domínio da informação do software, pensar na função necessária para executar o comportamento desejado e entender a melhor forma de obter este comportamento sem comprometer o desempenho. Ao fim deste processo os documentos de requisitos do sistema são criados e revistos com o cliente.

- Projeto de sistema e software: Um processo de múltiplos passos que enfoca quatro atributos distintos do programa, sendo eles: representações da interface, estrutura de dados, detalhes algoritimicos e arquitetura do software. Ou seja, este processo tem por finalidade traduzir os requisitos para uma representação do software de modo a permitir sua avaliação quanto à qualidade antes que a codificação tenha início. Este processo também cria uma documentação que se torna parte do projeto do software.

- Implementação e teste de unidade: O processo de implementação e teste de unidade consiste na geração de código, que pode ser realizada de forma manual, ou, mecânica, através dos documentos provenientes dos processos anteriores. Este processo traduz o projeto para uma linguagem de máquina, que pode ser testada ao fim do desenvolvimento de cada nova funcionalidade.

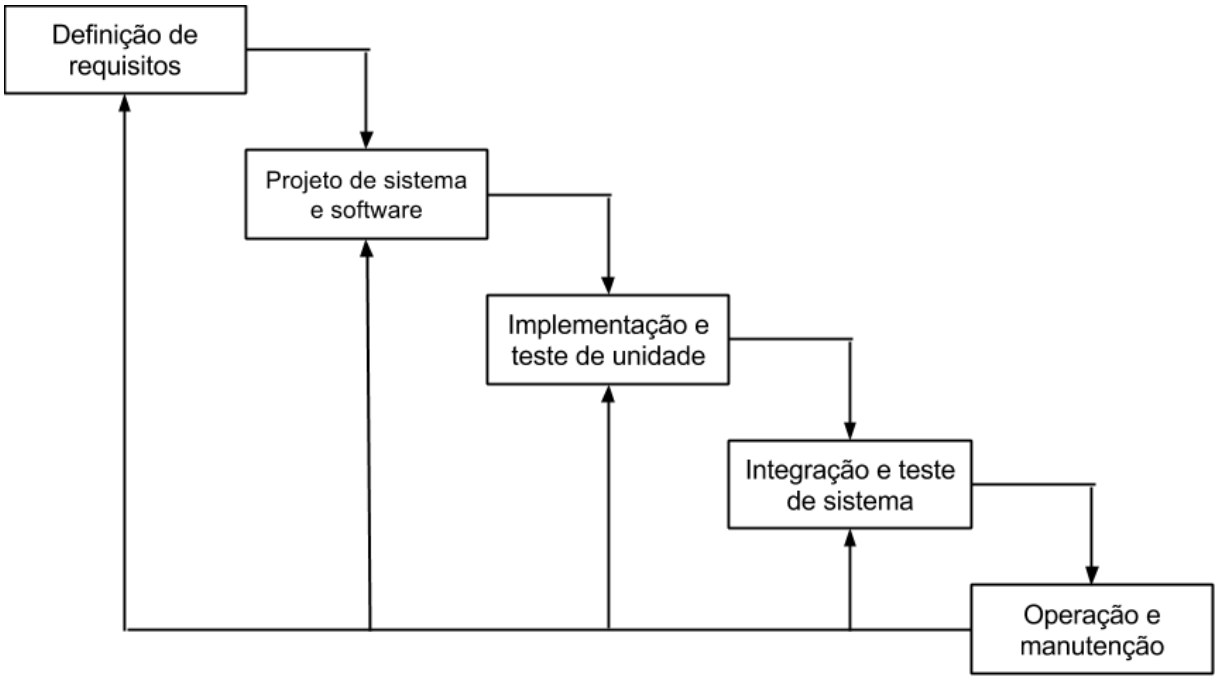
- Integração e teste de sistema: Neste processo ocorre a integração de todo o código, que é então depurado através do teste de sistema. A finalidade deste processo é garantir que todas as intruções sejam testadas para descobrir erros e garantir que as entradas definidas produzirão os resultados exigidos pelo cliente.

- Operação e manutenção: Após ser liberado para o cliente existe a possibilidade de o software necessitar de modificações inevitáveis. Essas modificações podem ocorrer quando são encontrados erros, quando o software precisa ser adaptado para acomodar inovações tecnológicas, ou quando o cliente deseja melhoramentos funcionais ou de desempenho. O processo de operação e manutenção reaplica cada um dos processos precedentes a um sistema já existente ao invés de construir um novo sistema.

Ao fim de cada estágio o resultado obtido é um ou mais documentos aprovados. Além disso, vale ressaltar que é necessário seguir a premissa de que um estágio não pode seguir adiante antes que o anterior tenha sido aprovado. Esse é o motivo do porquê do modelo se chamar cascata (SOMMERVILLE, 2007).

A figura 01 ilustra o modelo em cascata:

Figura 01: Ciclo de vida de software.



Fonte: Sommerville (2007)

Ao observar a figura acima, entende-se que o modelo em cascata se aplica melhor em projetos nos quais há boa especificação do sistema e pouca ou nenhuma alteração radical durante o desenvolvimento (SOMMERVILLE, 2007).

Seu grande trunfo é a rica documentação que será obtida ao fim de cada estágio do processo.

Contudo, conforme Sommerville (2007), outros problemas também podem ser identificados:

- Suspensão de estágios do processo.
- Retrabalho causado pela produção de documentos a cada estágio.
- Reprogramação causada pela omissão de problemas levados adiante para posterior correção.

- Alto custo de produção causado pelos motivos apresentados anteriormente.

- Sistemas mal estruturados tendo em vista que os problemas de projeto foram contornados através do uso de artifícios de implementação.

2.2.2 DESENVOLVIMENTO EVOLUCIONÁRIO

O desenvolvimento evolucionário, também conhecido como modelo de prototipagem, segue uma abordagem diferente e um pouco mais iterativa com relação ao cliente.

Segundo Sommerville (2007), este modelo divide o projeto em 3 estágios principais que podem ser intercalados, sendo eles: especificação, desenvolvimento e validação.

A idéia é que uma implementação inicial seja desenvolvida com a utilização de especificações abstratas, gerando um protótipo que tem como finalidade a confirmação do que foi proposto inicialmente. Esse protótipo é exposto ao cliente, que faz uma avaliação para validar se o software atende as suas necessidades. Por meio do refinamento do resultado dessa avaliação, novos requisitos são descobertos e especificados acarretando no desenvolvimento de várias versões até que se obtenha o resultado desejado (SOMMERVILLE, 2007).

Sommerville (2007, p. 45) acrescenta que:

> “Uma abordagem evolucionária para o desenvolvimento de software é frequentemente mais eficaz do que a abordagem em cascata na produção de sistemas que atendam às necessidades imediatas dos clientes. ”

Mas o desenvolvimento evolucionário também apresenta problemas, a lista a seguir evidencia os que mais se destacam, também segundo Sommerville (2007):

- Não visibilidade do processo devido a velocidade com que cada versão é gerada.

- Sistemas frequentemente mal estruturados devido incorporação de mudanças que corrompem sua estrutura inicial.

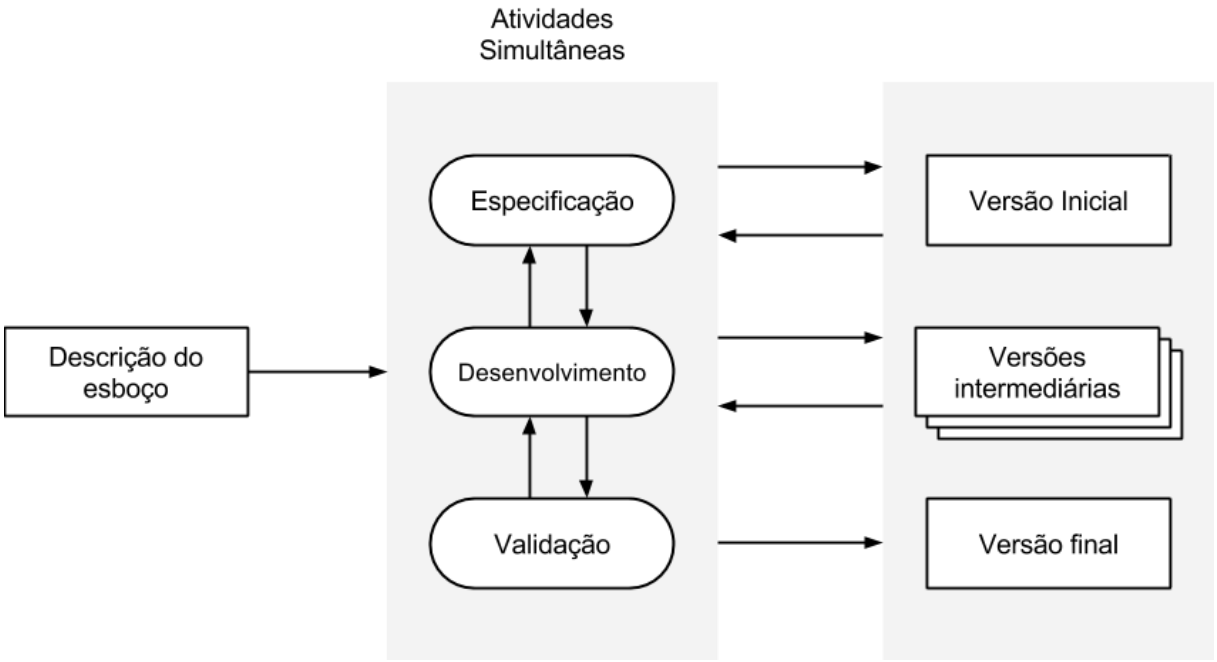
- Sistemas de grande porte que tem seu desenvolvimento segmentado em partes divididas entre várias equipes, dificultando a integração das contribuições de cada equipe.

Outro problema proveniente da aplicação deste modelo também foi identificado por Pressman (2002), sendo ele:

- O cliente se engana e acredita que vê uma versão executável do software, desconsiderando que o protótipo apenas funciona de maneira precária quando é levado em consideração a sua qualidade global e manutenabilidade a longo prazo. Então, o cliente reclama e exige que o protótipo seja “arrumado” com a finalidade de transforma-lo na versão final do produto.

A figura 02 ilustra o desenvolvimento evolucionário:

Figura 02: Desenvolvimento evolucionário.



Fonte: Sommerville (2007)

Sommerville (2007, p. 45), cita que:

> “A vantagem de um processo de software baseado na abordagem evolucionária é que a especificação pode ser desenvolvida de forma incremental. À medida que os usuários compreendem melhor seu problema, isso pode ser refletido no sistema de software. ”

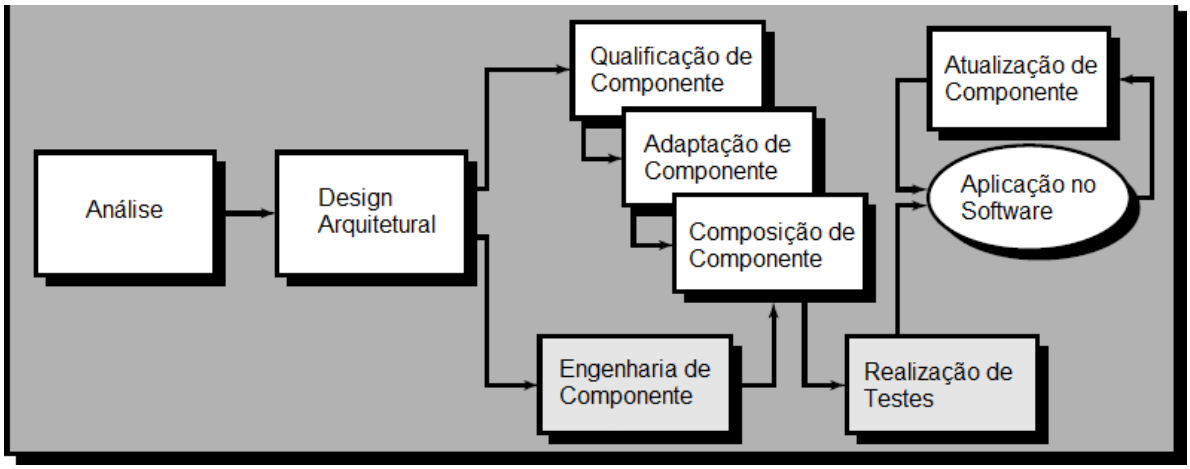
2.2.3 ENGENHARIA DE SOFTWARE BASEADA EM COMPONENTES

Segundo Sommerville (2007), o principal foco da engenharia de software baseada em componentes, ou, *component-based software engineering* (CBSE) é o reaproveitamento de componentes que já foram desenvolvidos, onde a maior parte do trabalho fica na integração dos mesmos, ao invés de desenvolver tudo a partir do zero.

Ainda segundo Sommerville (2007), a vantagem mais significativa neste modelo é a economia de tempo e recursos, resultando menos riscos. Porém, ao mesmo tempo, caso a empresa não controle os componentes é provável que a evolução do software seja mais complexa e alguns componentes não atinjam as necessidades do cliente.

A figura 03 ilustra os componentes da engenharia de software baseada em componentes; sendo eles: Análise, Design Arquitetural, Engenharia de Componente, Qualificação de Componente, Adaptação de Componente, Composição de Componente, Realização de Testes, Aplicação no Software e Atualização de Componente.

Figura 03: Caracterização do modelo CBSE.



Fonte: Adaptado de Semwal (2012)

Sommerville (2007) por fim constatou que já que cada projeto tem suas próprias características, não existe um modelo ideal de desenvolvimento, mas o ideal é que um aprimoramento seja feito através de processos onde a equipe de desenvolvimento personaliza as técnicas e métodos, sempre aliando com as boas práticas, permitindo que um software de boa qualidade seja feito no prazo ideal.

2.3 REUSO DE SOFTWARE

O reuso de software se basea na reutilização de componentes que já foram desenvolvidos, depurados, e que apresentam menor possibilidade de erros. Segundo Freitas (2003), essas qualidades relacionadas a reutilização de software influenciam em um aumento de produtividade de atividades ligadas ao desenvolvimento, levando em consideração que componentes prontos, quando usados por desenvolvedores experientes, economizam tempo.

Deutsch (1989) completa que pode haver reutilização direta do código-fonte e outros elementos do software de projetos já desenvolvidos, tais como concepções arquitetônicas de uma aplicação, contudo refletidos em outra aplicação e com uma implementação diferente.

Segundo Gamma *et al* (2000), pode-se dizer que o reuso de softwares é um meio de usar outras vezes o mesmo software. Os softwares são compostos por artefatos e ativos.

Os artefatos são quaisquer *workproducts* do ciclo de vida de desenvolvimento de software, tais como documentos de requisitos, modelos, fonte, arquivos de código, descritores de implementação, casos ou scripts de teste, e assim por diante. Em geral, o termo "artefato" está associado com um documento ou arquivo (GAMMA *et al*, 2000).

Existem vários tipos de ativos. Um tipo específico de ativo pode especificar os artefatos que devem estar no ativo e pode declarar um contexto específico, como um contexto de desenvolvimento ou um contexto de tempo de execução para que o ativo é relevante. Há três dimensões-chave que descrevem ativos reutilizáveis: granularidade, variabilidade e articulação (GAMMA *et al*, 2000).

Granularidade: Determina o número de problemas endereçados por um ativo. Pode ser pequena, quando o ativo trata apenas um problema específico. Mas pode

ser grande quando apresentação solução para uma grande diversidade de problemas, um bom exemplo seria um *framework*[1].

Visibilidade: Indica quanto um ativo pode ser visualizado e manipulado.

Articulação: Descreve o grau de completitude de um determinado ativo. Ou seja, o quão pronto um ativo está para a solução de um dado problema. Um conjunto de requisitos, por exemplo, está longe de solucionar efetivamente o problema. Diz-se que seu grau de articulação é baixo. Já um componente em sua forma executável apresenta um alto grau de articulação.

Sommerville (2007) aborda o reuso de componentes de uma forma pouco diferente, pois as partes reutilizáveis são vistas como unidades que apresentam características diferentes:

- Reuso de funções: caracterizado pelo reuso de componentes que implementam uma única função.

- Reuso de sistemas em aplicações: caracterizado pelo reuso de todo o sistema de aplicações incorporado ao novo sistema, sem modificações.

- Reuso de componentes: caracterizado pelo reuso de componentes que variam em tamanho, podendo ser um objeto isolado ou até mesmo um subsistema.

[1] Um framework é um conjunto de classes que, de acordo com Gamma *et al.* (2000), colaboram de forma a compor um projeto reutilizável, servindo de base para classes específicas do sistema.

3 PADRÕES DE PROJETO

Uma outra área muito importante para o reuso de software são os padrões de projeto (*design patterns*), uma boa descrição é dada por Christopher *et al* (1977, p. 15):

> "Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para esse problema, de tal forma que você pode usar esta solução um milhão de vezes, sem nunca o fazer da mesma maneira duas vezes. "

Padrões de projeto são considerados técnicas avançadas de orientação a objetos, para utilizá-los o desenvolvedor deve estar familiarizado com conceitos básicos de orientação a objetos tais como: herança, encapsulamento, classes, objetos, interfaces, dentre outros (DOFACTORY, 2013).

Eles são classificados de acordo com seu propósito, que especifica a qual tipo de problema que ele se aplica. Em sua obra, Gamma *et al* (2000), sugere a classificação, a qual divide os padrões em 3 categorias diferentes:

- Criacionais: Tratam especificamente do processo de criação de objetos no sistema.

- Comportamentais: Tratam especificamente a forma com a qual as classes e objetos realizam suas interações e distribuem suas tarefas do sistema.

- Estruturais: Tratam especificamente a forma com a qual as classes e objetos são compostos e usados, cabendo a eles propor uma solução para problemas de projeto muito complexos.

A Tabela 01 apresenta essa classificação de acordo com o **escopo**, podendo ser uma **classe** ou um **objeto** e se refere a qual tipo de componente que este padrão se aplica. Padrões cujo escopo é uma **classe** tendem a ser mais estáticos e lidam com a forma com que as classes e subclasses do sistema se relacionam. Por outro lado, padrões cujo escopo é um **objeto** tendem a ser mais dinâmicos, pois seus relacionamentos podem mudar em tempo de execução.

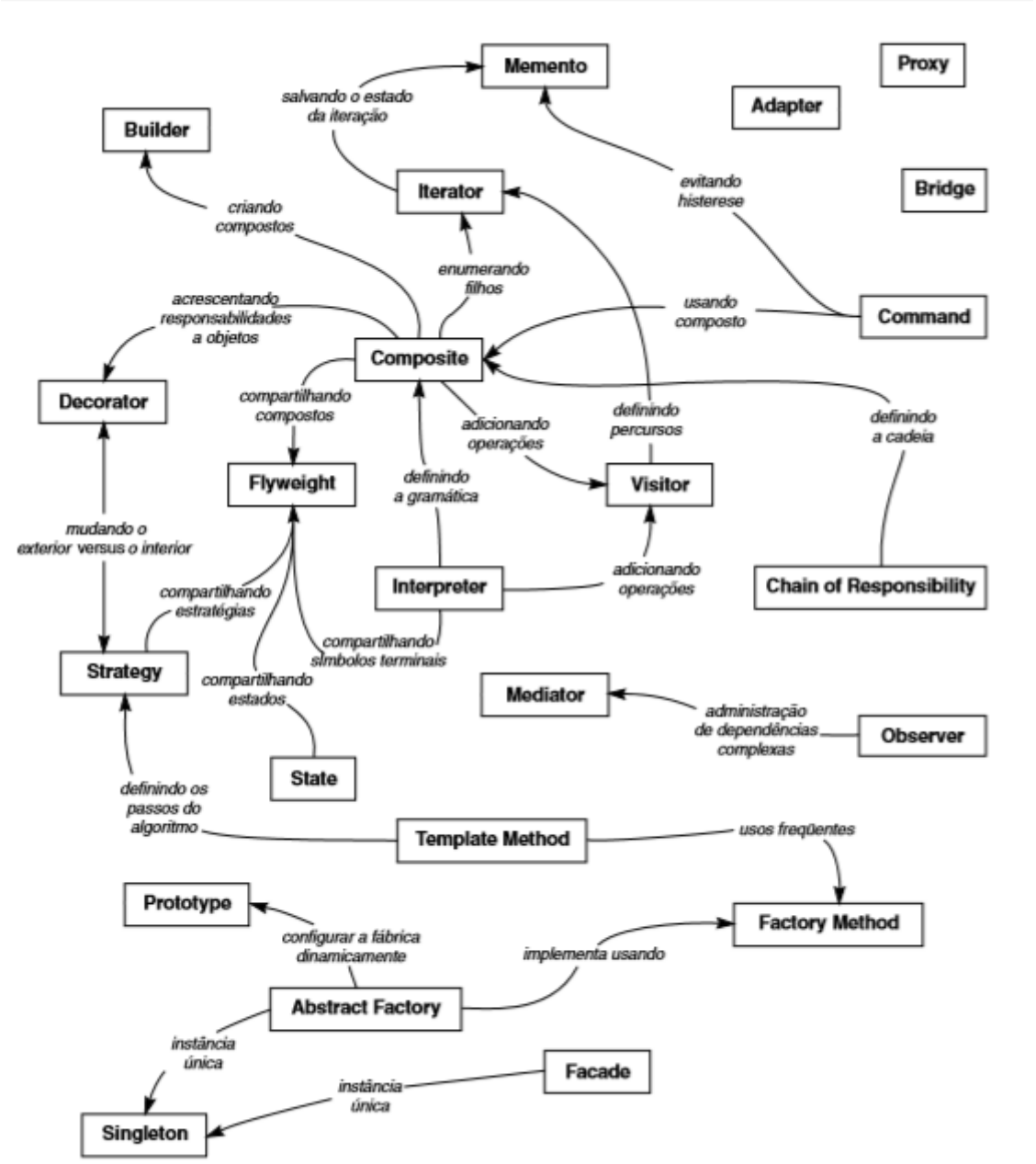
Tabela 01: O espaço dos padrões de projeto.

		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: Adaptado de Gamma *et al* (2000)

Ainda sob a ótica de Gamma *et al* (2000), pode ser observado na Figura 04 como os padrões se relacionam, tornando viável que um ou mais padrões sejam escolhidos e aplicados simultaneamente para solucionar um problema.

Figura 04: Relacionamentos entre padrões de projeto.



Fonte: Gamma *et al* (2000)

O principal objetivo desses padrões é encontrar soluções reutilizáveis para problemas encontrados com muita frequência nas etapas de construção de um sistema orientado a objetos.

Segundo Chu *et al* (2001) a aplicação de padrões de projeto não apenas diminu os custos de manutenção dos sistemas, mas também aumenta sua

reusabilidade. Tanto a engenharia avante[2] quanto a reengenharia[3] têm sua produtividade melhorada através do uso de padrões. Além disso, os engenheiros de software são beneficiados por terem uma base comum para melhor se comunicarem.

Segundo Gamma *et al* (2000) os padrões de projeto integram práticas de projeto bem-sucedidas e experiências de especialistas em um conjunto de componentes que apresentam comportamentos conhecidos com estruturas melhores. Sua aplicação na reengenharia permite a obtenção de sistemas com projeto e arquitetura consistentes, aumentando de forma significativa sua compreensibilidade e manutenabilidade.

A utilização de padrões possibilita o entendimento do projeto em um nível de abstração maior que o de segmentos de código. Essa facilidade no entendimento e na padronização torna a evolução do sistema muito mais efetiva, uma vez que eles são capazes de representar abstrações de alto nível, que tiveram suas soluções bem-sucedidas e documentadas, tornando-se fundamentais para o reuso no desenvolvimento de sistemas orientados a objetos (FREITAS, 2003).

3.1 DETECÇÃO DE PADRÕES DE PROJETO

O foco da detecção de padrões de projetos consiste em oferecer subsídios para a reestruturação de sistemas orientados a objetos, bem como é uma forma de avaliar sistemas que ainda não estão finalizados (GAMMA *et al*, 2000).

Identificar os problemas existentes (anti-padrões) possibilita que algumas das estruturas que podem comprometer uma futura expansão ou modificação possam ser identificadas e trocadas por outras mais apropriadas. A identificação de padrões possibilita que o projeto do sistema seja entendido em um nível maior de abstração, além de permitir uma avaliação sobre a utilização destes padrões no projeto (GAMMA *et al*, 2000).

Ainda segundo Gamma *et al* (2000), detectar padrões e anti-padrões em um projeto destinado a objetos é tarefa árdua e de difícil realização devido a vários fatores, dentre os quais:

---

[2] Utiliza-se de informações recuperadas a partir do código-fonte já existente, com a finalidade de reconstruir sistemas melhorando sua qualidade e desempenho (ZADROZNY, 2015).

Sistemas com interesse e necessidade de reestruturação, normalmente, correspondem a sistemas de médio/grande porte.

Geralmente, a única fonte segura de informação sobre o projeto está no código-fonte. Modelos, quando existem, estão desatualizados ou são superficiais demais para a detecção da maior parte dos problemas. A análise manual do código-fonte limita o escopo dos problemas que podem ser detectados de forma economicamente viável (GAMMA *et al*, 2000).

Desenvolvedores muitas vezes não sabem que tipo de construção procurar. A existência de uma base de construções de projeto, englobando tanto padrões como anti-padrões, pode fornecer um suporte valioso nesta busca.

O mecanismo de detecção atua sobre um modelo orientado a objetos extraído pela ferramenta de engenharia reversa[4] ou sobre um modelo previamente construído no ambiente de modelagem.

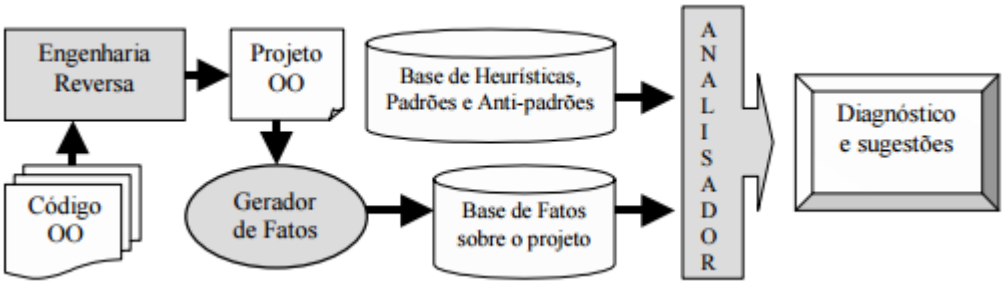
Uma visão geral da estrutura do mecanismo é ilustrada na Figura 05.

---

[3] Consiste em um processo de reconstrução de um sistema já existente com o uso dos mesmos passos e produtos já encontrados em um processo de software qualquer (ZADROZNY, 2015).

[4] Ferramentas de engenharia reversa são softwares capazes de extrair informações do sistema a partir do código-fonte, além de também extrairem informações de sistemas que já estão em produção (ZADROZNY, 2015).

Figura 05: Visão geral da estrutura do mecanismo de detecção.



Fonte: Correa (1999).

O mecanismo de detecção possibilita o relacionamento de padrões, anti-padrões e heurísticas de projeto. Um padrão pode ser solução para um determinado anti-padrão e/ou pode reforçar uma determinada heurística. Um anti-padrão pode violar uma determinada heurística. Desta forma, é importante que o usuário registre estes relacionamentos. (GAMMA *et al*, 2000).

3.2 ESCOLHA DE PADRÕES DE PROJETO

O catálogo de padrões criado por Gamma *et al* (2000) ajuda na escolha do padrão a ser adotado no projeto, contudo, é importante ressaltar que existem alguns critérios que segundo o autor da obra, são importantes para a escolha do padrão que virá a ser aplicado no projeto.

Quando o projetista não está familiarizado com o catálogo, fazer a escolha de um padrão dentre tantos existentes, torna-se um problema; pensando em formas de viabilizar esta escolha, Gamma *et al* (2000) apresenta diversas abordagens para auxiliar na escolha do padrão correto para solucionar o problema em questão, dentre elas:

- Considerar como padrões de projeto solucionam problemas de projeto: Os padrões ajudam a selecionar e encontrar objetos apropriados, determinando suas granularidades e especificando suas interfaces.

- Examinar quais as intenções dos padrões: A intenção refere-se ao propósito de criação do padrão em questão, que é a solução de algum problema relevante.

- Estudar como os padrões se inter-relacionam: Conforme já apresentado na Figura 04 a compreensão dos relacionamentos entre estes padrões pode ajudar na escolha do padrão ou grupo de padrões mais adequado.

- Estudar padrões de finalidades semelhantes: Diferentes padrões podem resolver o mesmo problema devendo-se considerar o padrão que é capaz de proporcionar a melhor solução.

- Examinar uma causa de reformulação do projeto: Alguns dos benefícios da utilização dos padrões são: o baixo acoplamento entre as classes, melhoria da estrutura de camadas, facilidade de extensão de hierarquias de classes, dentre outras encontradas especificamente em seus respectivos padrões. Dessa forma, uma delas pode servir como causa para uma possível reformulação do projeto.

- Considerar o que deveria ser viável no seu projeto: Esta abordagem estuda quais padrões podem ser escolhidos sem que ocorra grande reformulação do projeto, ao contrário do item anterior.

Levando em consideração algumas ou grupos dessas abordagens no desenvolvimento de software, a tarefa da escolha do padrão a ser adotado no projeto torna-se menos passível de erros, viabilizando sua aplicação de maneira mais consistente.

3.3 APLICAÇÃO DE PADRÕES DE PROJETO

Estudos sobre aplicabilidade, consequências, estrutura, participantes e colaborações do padrão escolhido são considerados, por Gamma *et al* (2000), suficientes para que sua aplicação ocorra de forma benéfica.

Contudo, mesmo que cuidados sejam tomados durante a etapa de escolha do padrão, projetistas de software inexperientes estão mais propensos a erros. Quando não há completa percepção dos impactos da aplicação do padrão, a tentativa da

solução a um problema que inicialmente era mais específico, acaba por gerar diversos problemas (GAMMA, 2000).

Um dos erros mais comuns dos projetistas de software inexperientes é a aplicação indiscriminada de padrões por motivos de flexibilidade e variabilidade. Essa prática além de diminuir o desempenho da aplicação, complica a evolução do projeto (GAMMA, 2000).

Padrões de projeto devem ser aplicados apenas quando são necessários, ou seja, sua flexibilidade é um dos fatores que indica que a solução escolhida é a correta para ser aplicada a problemas mais específicos.

4. PADRÃO STATE

Um problema específico vivenciado pelas equipes de desenvolvimento incide quando um objeto possui mais de um estado (*state*), e cada estado implica em um comportamento diferente. Isso resulta em uma dificuldade de controle desse comportamento tendo em vista as mudanças no estado interno do objeto (GUERRA 2013).

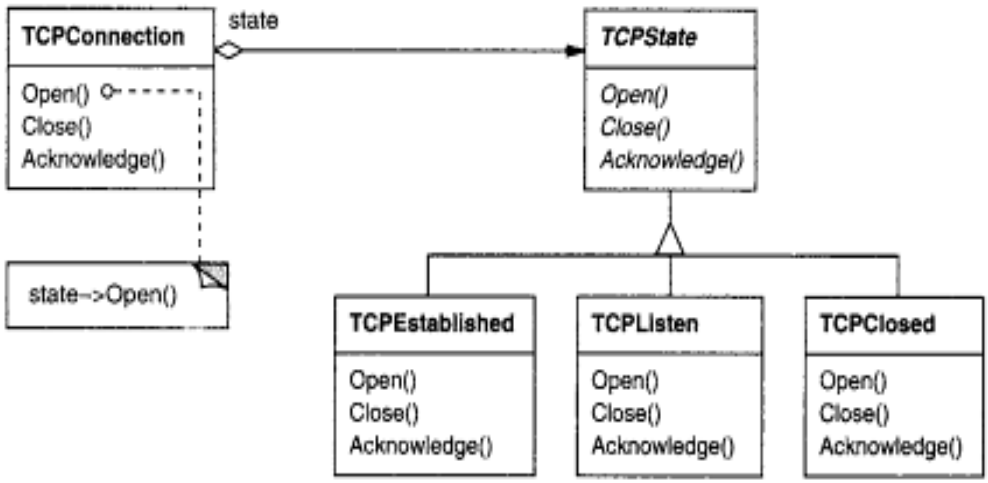
Guerra (2013, p. 57) acrescenta que:

> "A implementação comum nesses casos é criar uma série de condicionais em que, em diversos pontos da classe, verifica-se o estado do objeto e executa-se a lógica mais adequada. Seguindo essa idéia alguns estados são sufientes para deixar o código bastante confuso. É comum que a mesma sequência de condicionais seja encontrada em diversos pontos do código onde o estado do objeto possui influência. Além disso, essa estrutura torna complicada a adição de mais estados, pois isso demandará alterações no código da própria classe e irá aumentar mais a quantidade de condicionais. "

O padrão State permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe (GAMMA *et al*, 2010).

De acordo com a Figura 06, a classe TCPConnection permite abrir e fechar uma conexão de rede. A idéia chave deste padrão é introduzir uma classe abstrata chamada TCPState para representar os estados de conexão de rede. A classe TCPState declara uma interface comum para todas as classes que representam diferentes estados operacionais. As subclasses de TCPState representam estados concretos e implementam comportamentos específicos ao estabelecido (GAMMA *et al*, 2000).

Figura 06: Exemplo de comportamento TCPState.



Fonte – Gamma (2000)

A classe TCPConnection mantém um objeto de estado que representa o estado corrente na conexão TCP (GAMMA *et al*, 2000).

Connection delega todas as solicitações específicas de estados para este objeto de estado. TCPConnection usa sua instancia da subclasse de TCPState para executar operações específicas ao estado da conexão (GAMMA *et al*, 2000).

Sempre que a conexão muda de estado, o objeto TCPConnection muda o objeto de estado que ele utiliza. Por exemplo, quando a conexão passa do estado Established para o estado Closed, TCPConnection substituirá sua instancia de TCPEstablished por uma instancia de TCPClosed (GAMMA *et al*, 2000).

4.1 APLICABILIDADE DO STATE

O comportamento de um objeto depende de seu estado e ele pode mudar seu comportamento em tempo de execução. Dependendo desse estado, operações têm comandos condicionais grandes, com várias alternativas, que dependem do estado do objeto. Esse estado é normalmente representado por uma ou mais constantes enumeradas (GAMMA *et al*, 2000).

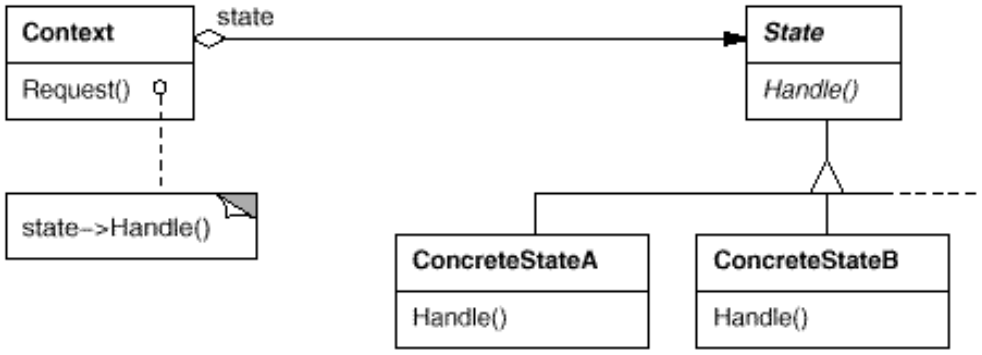
Frequentemente, várias operações conterão essa mesma estrutura condicional. O padrão State coloca cada bloco de código-fonte do comando adicional em uma classe separada. Isto lhe permite tratar o estado do objeto como

um objeto propriamente dito, que pode variar independentemente de outros objetos (GAMMA *et al*, 2000).

4.2 ESTRUTURA DO STATE

Conforme a Figura 07, o objeto do tipo Context delega solicitações específicas de estados para o objeto ConcreteState corrente. Um objeto do tipo Context pode passar uma referência para si, como um argumento para o objeto State acessar o seu contexto, se necessário. Context é a interface primária para os clientes. Clientes não necessitam tratar os objetos State diretamente. Tanto Context como as subclasses ConcreteState podem decidir a sequência de estados (GAMMA *et al*, 2000).

Figura 07: Estrutura State.



Fonte: GAMMA, 2000.

Nesta estrutura Gamma *et al* (2000) apresenta as seguintes definições:

Context (TCPConnection)

Define a interface de interesse dos clientes.

Mantém uma instancia de uma subclasse ConcreteState que define o estado corrente.

State (TCPState)

Define uma interface para encapsulamento associado comum determinado estado de Context.

ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)

Cada subclasse implementa um comportamento associado com um estado do Context.

4.3 VANTAGENS E DESVANTAGENS DO STATE

Segundo Gamma *et al* (2000) e Guerra (2014), diversas vantagens podem ser obtidas com a aplicação do padrão State, dentre elas:

- É fácil de localizar as responsabilidades de estados específicos, devido as classes que correspondem a cada estado. Isto proporciona uma maior clareza no desenvolvimento e nas subsequentes manutenções. Esta facilidade é fornecida pelo fato de que diferentes estados são representados por um único atributo (estado) e não envolvidos em diferentes variáveis e grandes condicionais.

- Faz as mudanças de estado explícitas, posto que em outro tipo de implantação os estados são alterados, modificando os valores em variáveis, enquanto aqui fazer-se representar cada estado.

- Os objetos State podem ser compartilhados se eles não contêm variáveis de instância, isto pode ser alcançado se o estado está totalmente codificado representando seu tipo. Quando isso é feito, os estados são flyweights sem estado intrínseco.

- Facilita a expansão de estados.

- Permite a um objeto alterar de classe em tempo de execução dado que ao modificar suas responsabilidades pela de outro objeto de outra classe, a herança e responsabilidades do primeiro mudaram pelas do segundo.

Porém, o padrão State também apresenta algumas desvantagens, dentre elas Gamma *et al* (2000) destaca a seguinte:

- O padrão State distribui o comportamento para diversos estados entre várias subclasses de State, e isso aumenta o número de classes e se torna menos compacto do que ter uma única classe. Então a principal desvantagem do padrão State é que este aumenta o número de subclasses.

5 ESTUDO DE CASO

5.1 APRESENTAÇÃO DO CASO

Segundo a Forbes (2015), as instituições que lideram o ranking mundial das mais lucrativas no ano de 2014 são os grandes bancos e seguradoras chineses e norte-americanos. Antigamente os bancos tinham apenas uma finalidade, guardar o dinheiro de seus clientes com segurança. Entretanto, com o avanço da tecnologia e das necessidades humanas, certas mudanças na rotina dessas instituições foram percebidas e resolvidas com o auxílio da infraestrutura proporcionada pela Tecnologia da Informação. Uma das mudanças mais significantes foi a introdução de um dispositivo chamado caixa eletrônico. Isso se justifica pelo número de operações que eles são capazes de realizar em benefício de seus clientes, incluindo ainda clientes de outros Bancos.

Segundo Pimentel (2015), o primeiro caixa eletrônico foi instalado em uma agência do Barclays Bank em 1967. Desde então, sua evolução foi considerável e eles foram se tornando mais tecnológicos, oferecendo, por exemplo, o suporte para a execução de algumas tarefas até então desempenhadas apenas por funcionários dessas instituições, através do uso de softwares específicos. Algumas dessas tarefas podem ser observadas a seguir: depósitos, saques, transferências, saldos, pagamentos, dentre outros.

A proposta desse estudo de caso é a construção de dois protótipos, um com o padrão State outro sem o padrão State. O objetivo deles é simular o funcionamento de um caixa eletrônico contendo apenas as operações consideradas básicas para o gerenciamento de uma conta, sendo elas: depósito, saque, recebimento de juros, acúmulo de juros, bloqueio de conta e desbloqueio de conta.

Os protótipos fazem algumas interações entre dois usuários, sendo eles: um cliente e um gerente. Caso alguma dessas interações implique em diferenças no saldo da conta, trocas de estados serão realizadas tendo em vista os seguintes tipos de conta:

- Conta Especial: Uma conta é definida como especial quando o saldo presente nela é superior ou igual a 1000. A vantagem de contas especiais reside no recebimento de juros referente a 0,05 do saldo, além da isenção das tarifas de saque, depósito e acúmulo de juros.

- Conta Normal: Uma conta é definida como normal quando o saldo presente nela é superior ou igual a 0 e inferior a 1000. Não possuem nenhuma vantagem específica e pagam tarifa nas operações de saque.

- Conta Negativa: Uma conta é definida como negativa quando o saldo presente é inferior a 0. Também não possuem nenhuma vantagem específica e pagam tarifa nas operações de depósito, além de acumularem juros referente a 0,02 do saldo já negativo.

- Conta Bloqueada: Uma conta é definida como bloqueada quando o gerente solicita o seu bloqueio. Durante este estado nenhuma operação pode ser efetuada na conta até que o gerente efetue seu desbloqueio.

Mais detalhes sobre o funcionamento dos protótipos são apresentados a seguir:

5.2 FUNCIONAMENTO

Basicamente, o funcionamento dos protótipos se resume a um cliente que possui uma conta bancária e realiza operações de saque, depósito, recebimento e acúmulo de juros. A conta, por sua vez, é bloqueada por um gerente para que uma análise possa ser efetuada. Uma vez terminada esta análise ela é desbloqueada e suas transações financeiras voltam a surtir efeito.

A figura 08 mostra o código-fonte da classe Main, responsável por criar os objetos apropriados e invocar os métodos que executam operações bancárias realizadas por um objeto do tipo cliente e um objeto do tipo gerente.

Figura 08: Código-fonte da classe Main de ambos os protótipos.

```java
package Common;

import Banco.Conta;
import Pessoa.Cliente;
import Pessoa.Gerente;

/**
 *
 * @author lucas.oliveira
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Cliente cliente = new Cliente("Lucas", "47.131.616-7", "Programador", new Conta());
        Gerente gerente = new Gerente("Oliveira", "395.588.618-20", "Projetista");

        cliente.getConta().Depositar(500.0);
        cliente.getConta().Depositar(300.0);
        cliente.getConta().Depositar(550.0);

        cliente.getConta().ReceberJuros();

        cliente.getConta().Retirar(2000.0);
        cliente.getConta().Retirar(1100.0);
        cliente.getConta().Retirar(50.0);

        cliente.getConta().AcumularJuros();

        cliente.getConta().Depositar(800.0);

        gerente.BloquearConta(cliente.getConta());

        cliente.getConta().Retirar(220.85);

        cliente.getConta().Depositar(5000.0);

        gerente.DesbloquearConta(cliente.getConta());

        cliente.getConta().Retirar(220.85);

        cliente.getConta().Depositar(5000.0);
    }
}
```

Fonte: gerada pelo autor.

Este código-fonte foi desenvolvido na linguagem de programação Java e teve o auxílio da ferramenta de desenvolvimento Netbeans IDE para executar testes unitários. Ambas as tecnologias são gratuitas e estão disponíveis para download na Internet.

As figuras 09 e 10 mostram quais foram as saídas obtidas com a execução dos protótipos. Cada conta possui uma identificação única, além de outras informações tais como: operação, valor, saldo e estado.

Figura 09: Primeira parte da saída obtida após execução dos protótipos.

```
run:
Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Depósito
Valor: R$ 500.0
Saldo: R$ 500.00
Estado: Conta Normal

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Depósito
Valor: R$ 300.0
Saldo: R$ 800.00
Estado: Conta Normal

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Depósito
Valor: R$ 550.0
Saldo: R$ 1350.00
Estado: Conta Especial

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Receber Juros
Saldo: R$ 1417.50
Estado: Conta Especial

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Saque
Valor: R$ 2000.0
Saldo: R$ -582.50
Estado: Conta Negativa

Impossível realizar o saque no valor de R$ 1100.0
O saldo está negativo em R$ -582.50

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Saque
Valor: R$ 1100.0
Saldo: R$ -582.50
Estado: Conta Negativa

Impossível realizar o saque no valor de R$ 50.0
O saldo está negativo em R$ -582.50
```

Fonte: gerada pelo sistema desenvolvido.

Um dos detalhes da figura 10 são as mensagens de bloqueio e desbloqueio exibidas a cada realização de uma operação enquanto a conta encontra-se bloqueada.

Outro detalhe também da figura 10 é a necessidade de interação com o usuário, que requer que uma tecla seja pressionada quando o gerente realiza o bloqueio ou desbloqueio de uma conta.

Figura 10: Segunda parte da saída obtida após execução dos protótipos.

```
Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Saque
Valor: R$ 50.0
Saldo: R$ -582.50
Estado: Conta Negativa

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Acumular Juros
Saldo: R$ -594.15
Estado: Conta Negativa

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Depósito
Valor: R$ 800.0
Saldo: R$ 190.85
Estado: Conta Normal

A conta d6ce5406-519c-499b-bfed-669897ea96a0 foi bloqueada pela Gerencia.
Pressione alguma tecla para continuar...

As operações relacionadas a esta conta encontram-se bloqueadas pela Gerência.
Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Saque
Valor: R$ 220.85
Saldo: R$ 190.85
Estado: Conta Bloqueada

As operações relacionadas a esta conta encontram-se bloqueadas pela Gerência.
Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Depósito
Valor: R$ 5000.0
Saldo: R$ 190.85
Estado: Conta Bloqueada

A conta d6ce5406-519c-499b-bfed-669897ea96a0 foi desbloqueada pela Gerencia.
Pressione alguma tecla para continuar...

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Saque
Valor: R$ 220.85
Saldo: R$ -30.60
Estado: Conta Negativa

Conta: d6ce5406-519c-499b-bfed-669897ea96a0
Operação: Depósito
Valor: R$ 5000.0
Saldo: R$ 4954.40
Estado: Conta Especial

BUILD SUCCESSFUL (total time: 7 seconds)
```

Fonte: gerada pelo sistema desenvolvido.

5.3 UML

Segundo Sampaio (2015), a UML, ou, *Unified Modeling Language*, é uma linguagem de modelagem unificada. Isto é, ela provê aos projetistas de software um conjunto de técnicas que permitem modelar elementos, mecanismos de extensibilidade, diagramas e relacionamentos existentes num projeto de software.

Ainda segundo Sampaio (2015), ela destina-se, basicamente, a permitir que desenvolvedores possam visualizar, especificar, construir e documentar artefatos de um software em desenvolvimento, pois dessa forma o software passa a ter uma representação visual antes de entrar na etapa de implantação.

Contudo, não pode ser considerada uma metodologia de desenvolvimento, pois não propõe passos a serem seguidos, mas sim uma visão geral do desenho e da comunicação entre objetos do projeto.

Uma citação importante de Sampaio (2015) registra que:

> "Da mesma forma que é impossível construir uma casa sem primeiramente definir sua planta, também é impossível construir um software sem inicialmente definir sua arquitetura. "

A UML proporciona uma maneira visual de representação de estrutura, funcionamento e operação de um sistema (BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. 2005). Os diagramas devem ser elaborados de maneira a exibir visões diferentes das diversas funcionalidades do sistema.

Sampaio (2015) argumenta que diagramas são uma parte fundamental da UML, pois provem uma representação parcial do software, ajudando a compreender a arquitetura a ser desenvolvida.

Logo, conclui-se que a UML possui uma relevância significativa para a documentação do projeto e consequentemente para o sucesso e qualidade do software que será produzido. Desta forma, foi a linguagem escolhida para a criação dos diagramas que contemplam a documentação dos protótipos utilizados no modelo em cascata. Alguns destes diagramas, considerados mais relevantes para a elaboração dos protótipos, foram criados através da ferramenta Astah Professional com o propósito de auxiliar no desenvolvimento e manutenção dos protótipos.

Os diagramas elaborados têm o objetivo de mostrar visões diferentes das diversas funcionalidades do sistema.

5.3.1 DIAGRAMAS

5.3.1.1 CASO DE USO

Diagramas de caso de uso tem como objetivo descrever um cenário mostrando as funcionalidades do software do ponto de vista dos usuários. Além disso, auxiliam na comunicação entre as partes envolvidas no projeto.

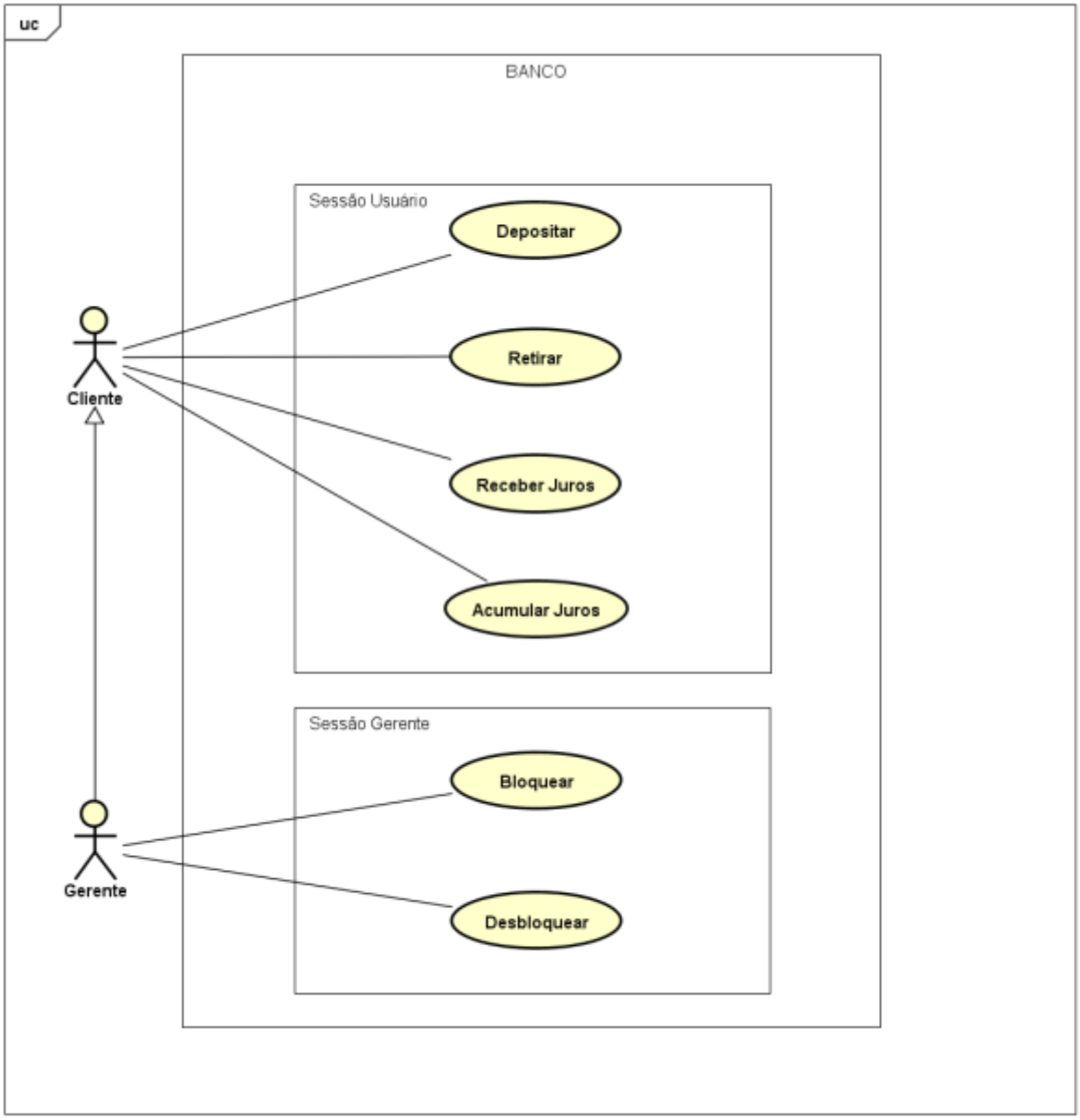
É um diagrama muito útil para passar as funcionalidades básicas de um sistema para pessoas que não possuem conhecimento técnico na área (BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. 2005).

Contam com 3 elementos para representações, sendo eles: Atores, Casos de Uso e Relacionamentos entre Atores e Casos de Uso.

No diagrama 01 existe um relacionamento de generalização entre os atores, além de associações entre eles e seus respectivos casos de uso.

O conceito de generalização, segundo a UML, consiste numa representação em que o ator que foi generalizado, neste caso o gerente, também utiliza dos casos de uso do ator principal, neste caso o cliente.

Diagrama 01: Diagrama de Caso de Uso de ambos os protótipos.



Fonte: gerado pelo autor.

5.3.1.2 CLASSES

O diagrama de classes exibe os blocos básicos de um sistema orientado a objetos. Ao invés de mostrar detalhes de funcionamento dos métodos e operações, ele é responsável pela exibição dos atributos e métodos existentes. Este diagrama é útil principalmente por ilustrar de maneira simples a relação entre as classes (BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. 2005).

Segundo Sampaio (2015), estes diagramas descrevem como os vários tipos de objetos presentes num software se relacionam, e, eles podem ser obserados sob 3 perspectivas diferentes, sendo elas:

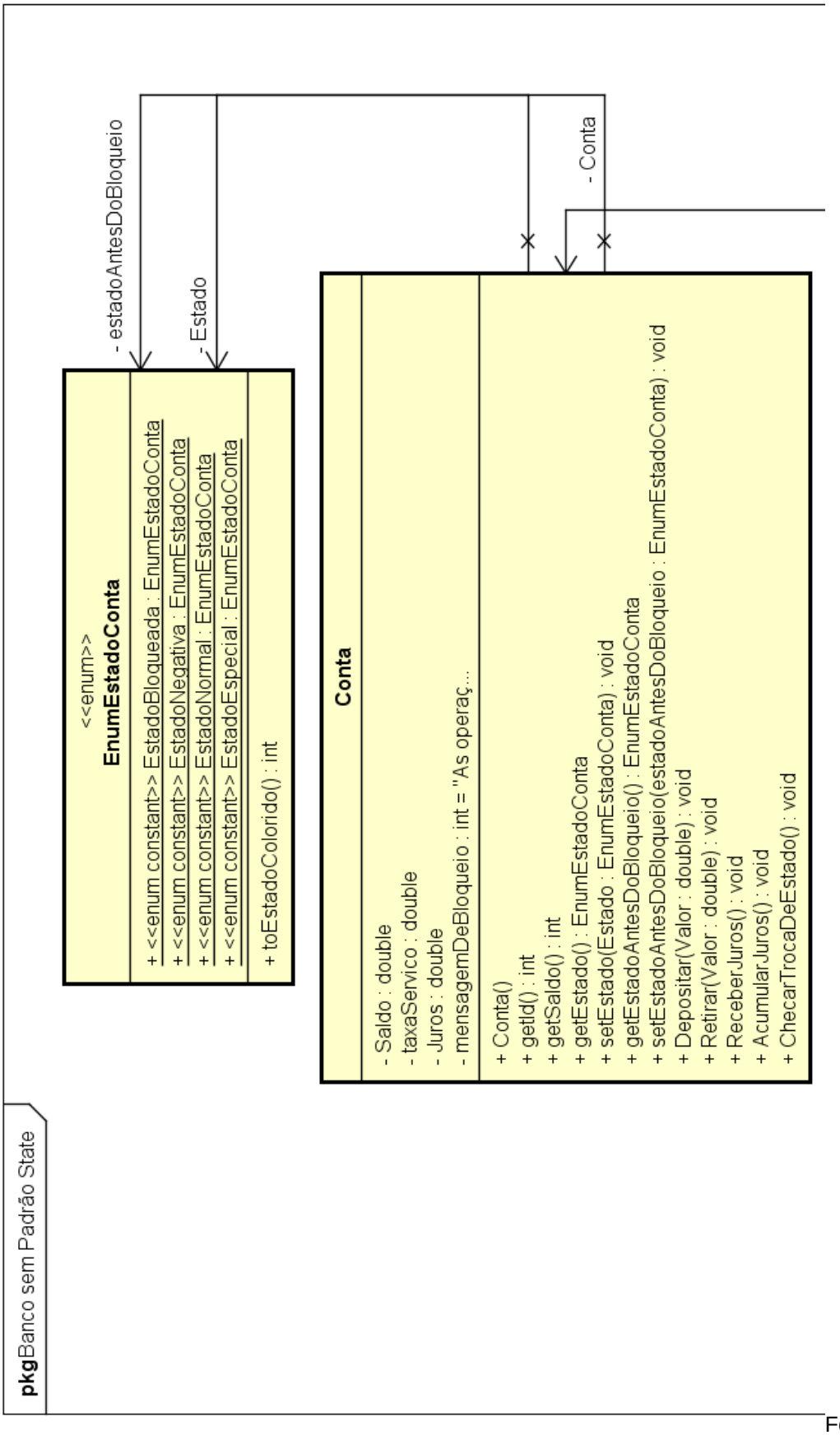
- Conceitual: Esta perspectiva destina-se ao cliente e representa os conceitos do domínio em estudo.

- Especificação: Essa perspectiva destina-se a pessoas que não necessitam de detalhes de desenvolvimento. Seu foco é nas interfaces da arquitetura e nos principais métodos, deixando de lado como eles serão implementados.

- Implementação: Essa perspectiva destina-se ao time de desenvolvimento e é a mais utilizada. Nela são detalhados vários tipos de informações úteis a equipe de desenvolvimento, tais como tipo de atributos, retorno de métodos, navegabilidade, relacionamentos de herança, composição, dentre outros.

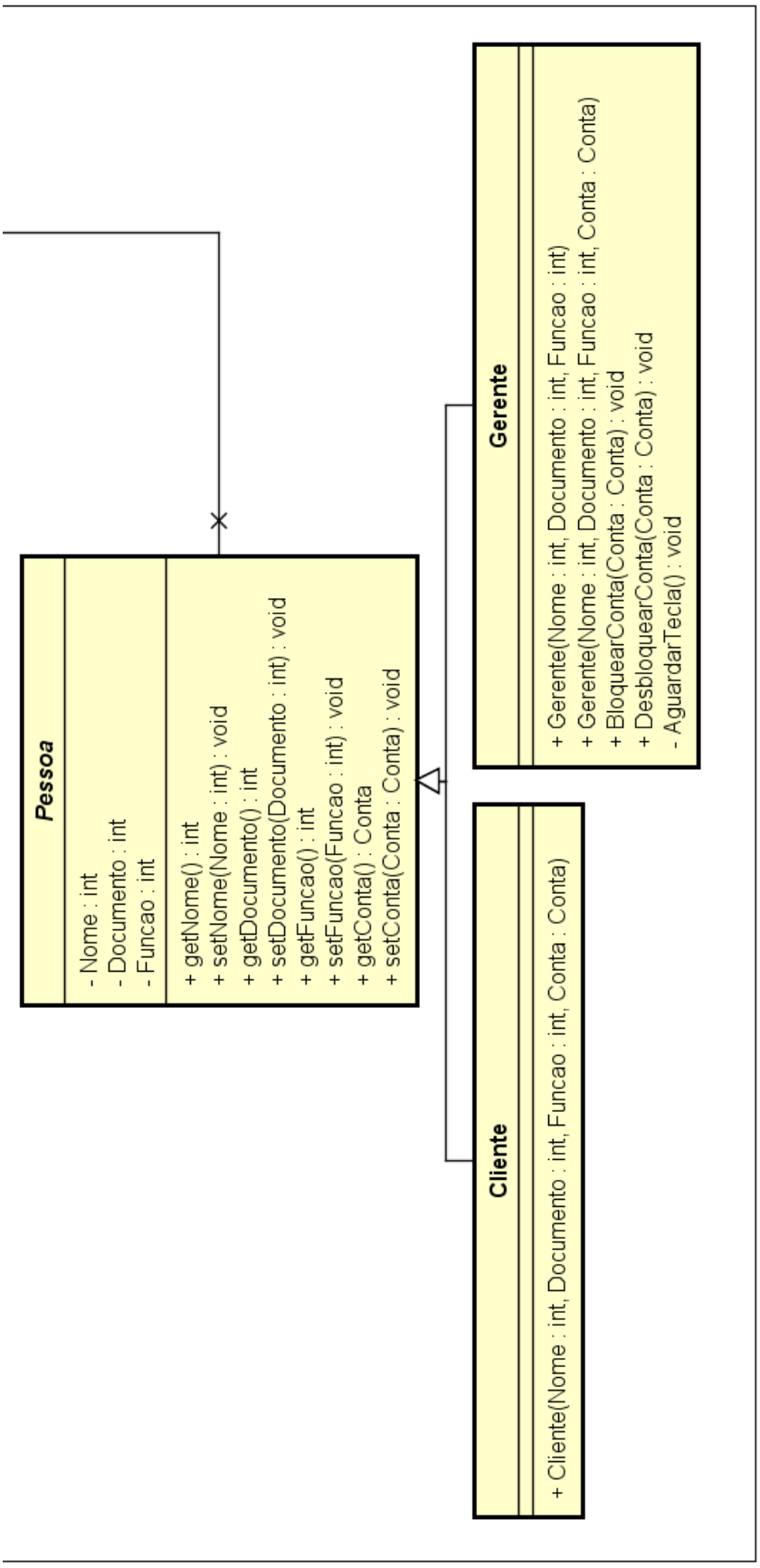
Para ilustrar os 02 protótipos, foram criados os diagramas de classe 02, 03, 04 e 05 que demonstram, respectivamente, o protótipo sem e com o padrão State aplicado.

Diagrama 02: Diagrama de Classes do protótipo sem padrão State – Parte 1.



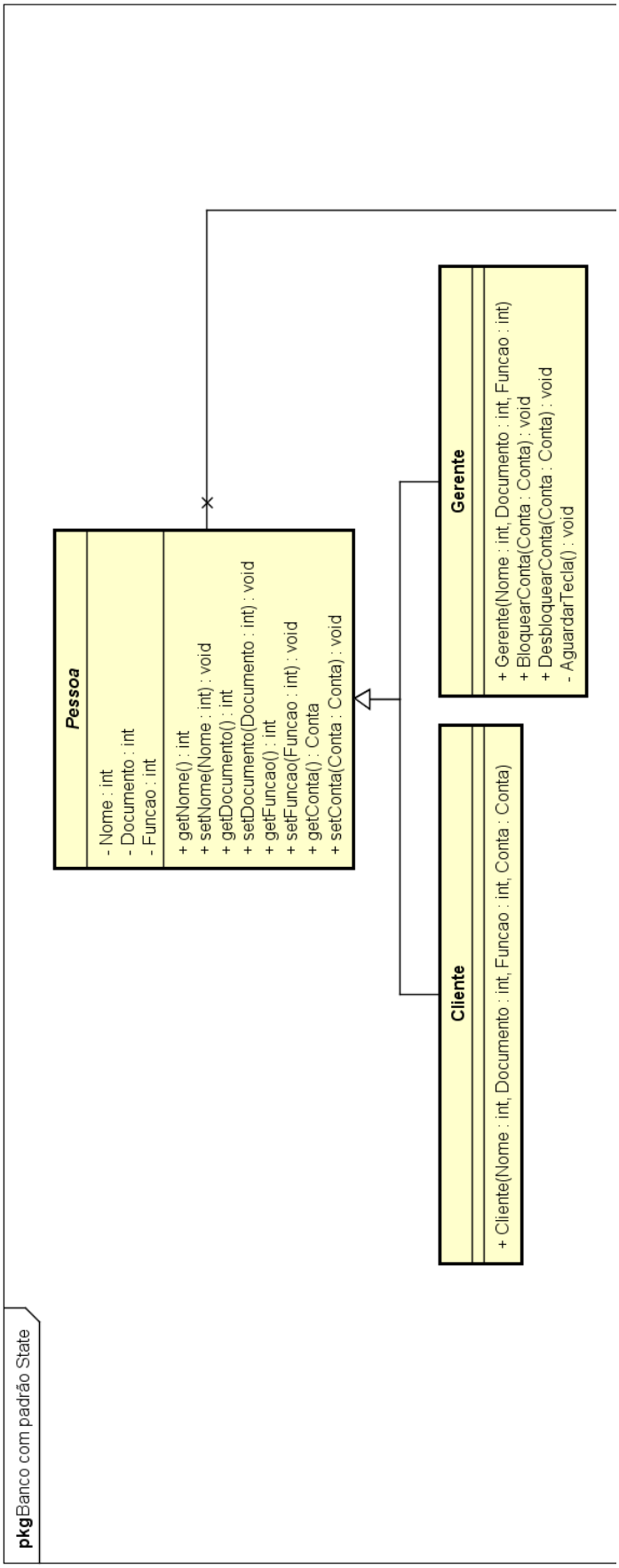
Fonte: gerado pelo autor.

Diagrama 03: Diagrama de Classes do protótipo sem padrão State – Parte 2.



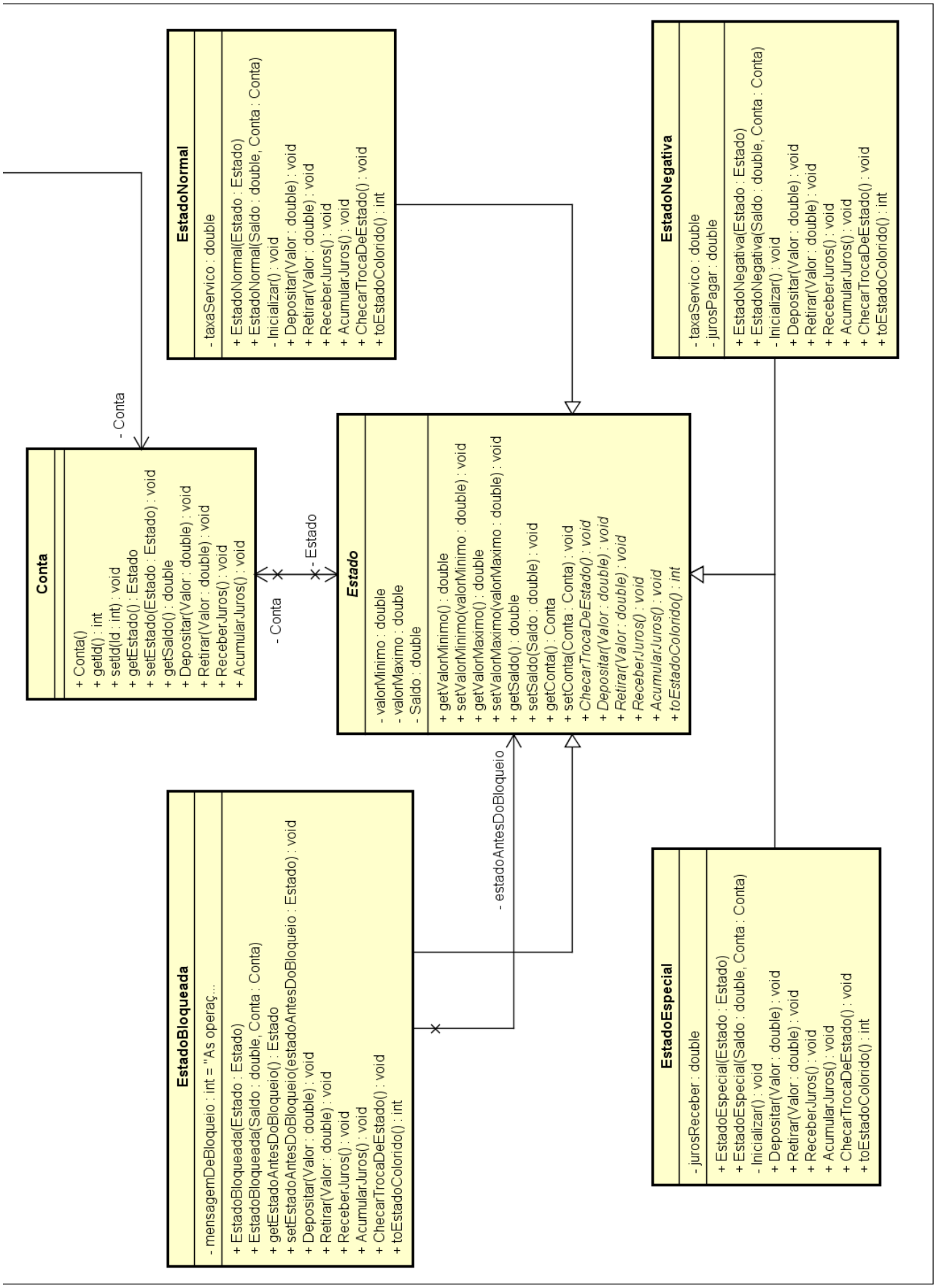
Fonte: gerado pelo autor.

Diagrama 04: Diagrama de Classes do protótipo com padrão State – Parte 1.



Fonte: gerado pelo autor.

Diagrama 05: Diagrama de Classes do protótipo com padrão State – Parte 2.



Fonte: gerado pelo autor.

Através dos diagramas de classe apresentados é possível notar que as classes Pessoa, Cliente e Gerente possuem exatamente a mesma estrutura. Isso porque segundo o contexto e especificações, tanto um usuário do tipo cliente quanto um usuário do tipo gerente são considerados uma pessoa, e, desta forma, têm o direito de possuir uma conta.

As classes de maior destaque no protótipo sem o padrão State são: Conta e EnumEstadoConta. Elas são responsáveis por gerenciar as transações de estado do objeto e os possíveis comportamentos atrelados a cada estado, além de prover também métodos comuns para depósito, saque, recebimento de juros e acúmulo de juros.

Já no protótipo com o padrão State, as classes de maior destaque são: Conta, Estado, Estado Negativa, Estado Normal, Estado Especial e Estado Bloqueada. Isso porque a classe Conta mantém o seu propósito inicial, porém tem sua implementação modificada para a aplicação do padrão State. Dessa forma, cabe a classe abstrata Estado o dever de delegar a implementação de todas as operações a todas as classes de estados concretos (Estado Negativa, Estado Normal, Estado Esepecial e Estado Bloqueada) que devem implementar as operações com suas respectivas particularidades, tais como: tarifas de serviço e cobrança de juros.

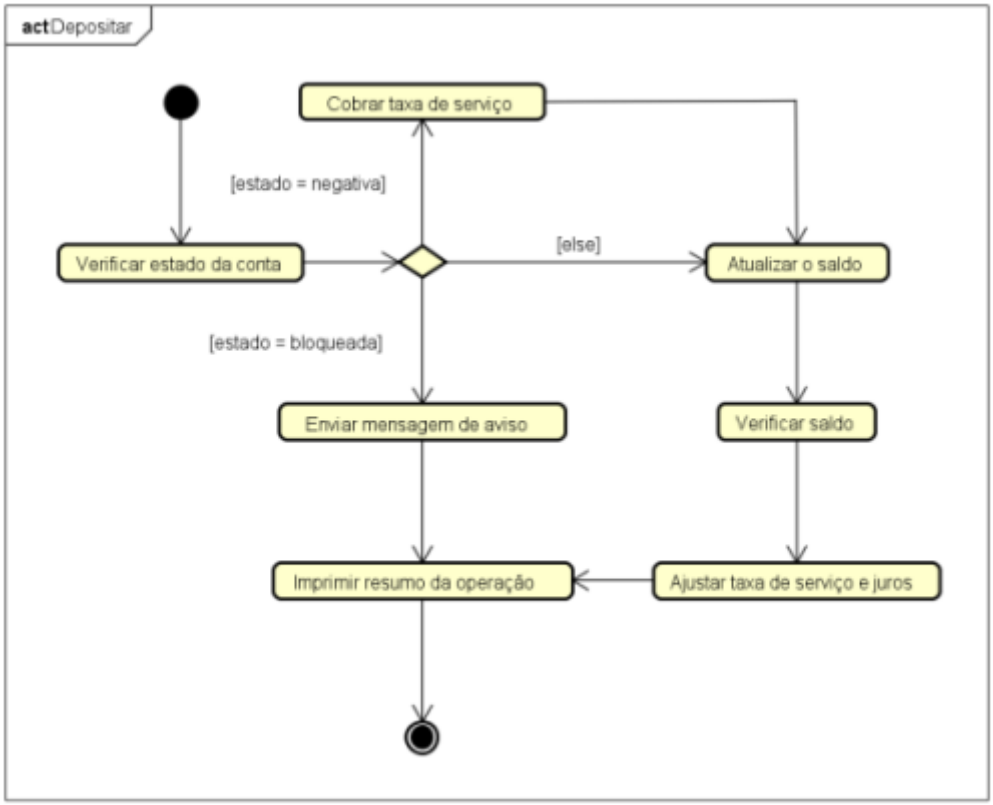
5.3.1.3 ATIVIDADES

O diagrama de atividades é responsável pela exibição de uma determinada sequência de atividades. Este diagrama mostra o fluxo das atividades de uma parte do sistema, do ponto inicial ao ponto final, passando por tomadas de decisões e todos os eventos contidos nesta área do sistema (BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. 2005).

Sampaio (2015) acrescenta que eles podem evidenciar a relação de dependência existente entre as atividades envolvidas em um unico processo, destacando quando e porque elas dependem umas das outras.

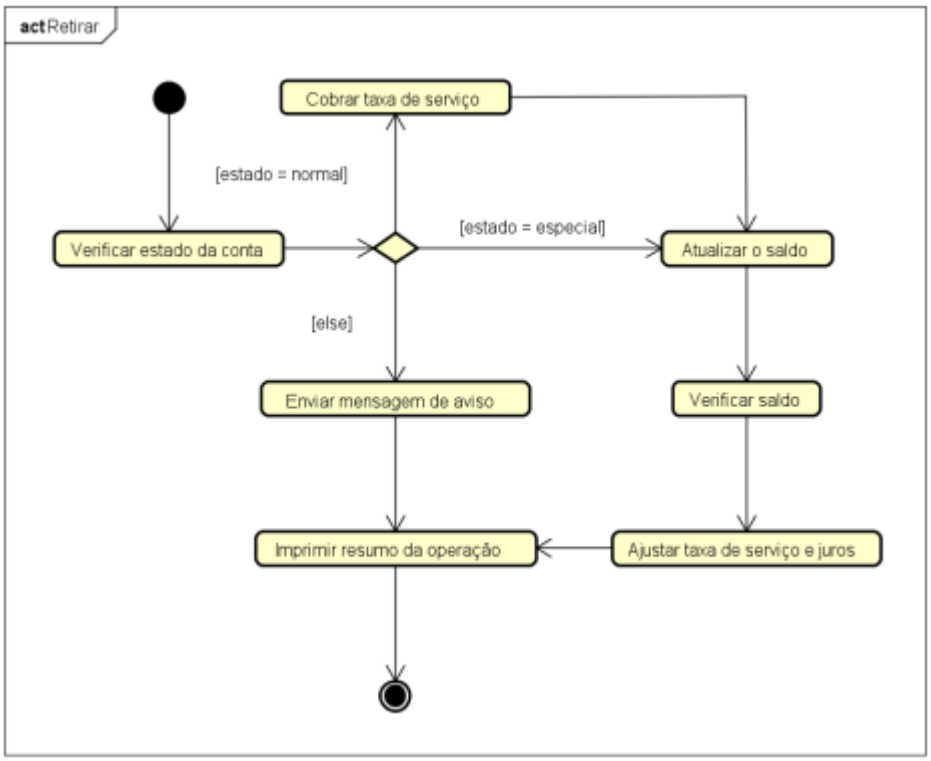
Os diagramas a seguir mostram, de maneira simples, todas as atividades envolvidas nas principais operações realizadas pelos usuários do protótipo.

Diagrama 06: Diagrama de Atividades da operação de depositar valor na conta.



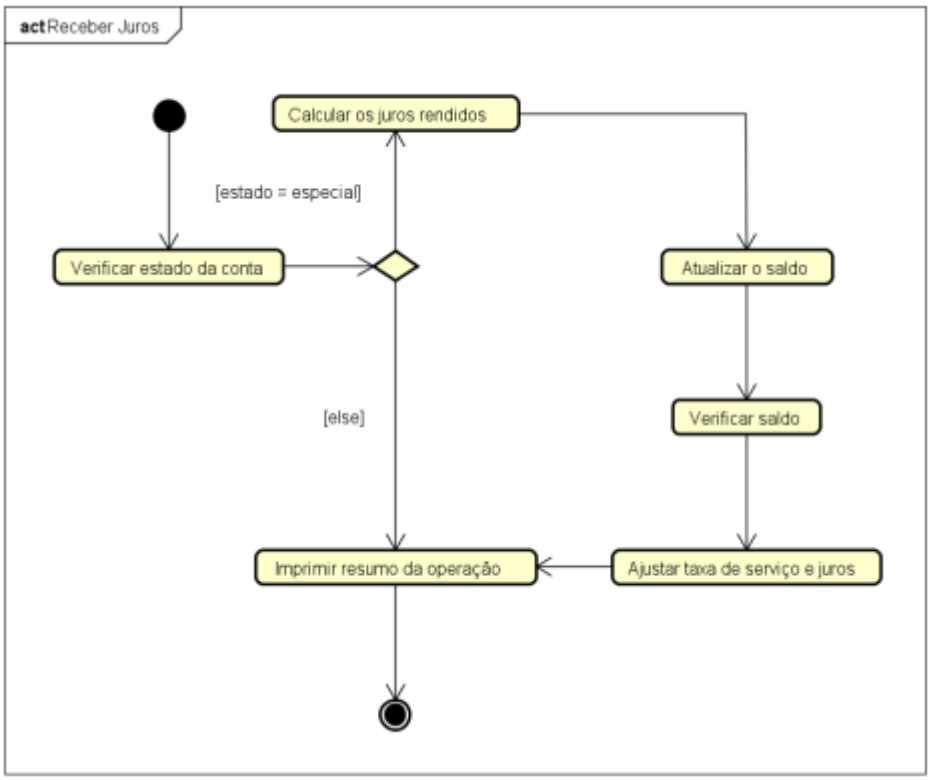
Fonte: gerado pelo autor.

Diagrama 07: Diagrama de Atividades da operação de retirar valor da conta.



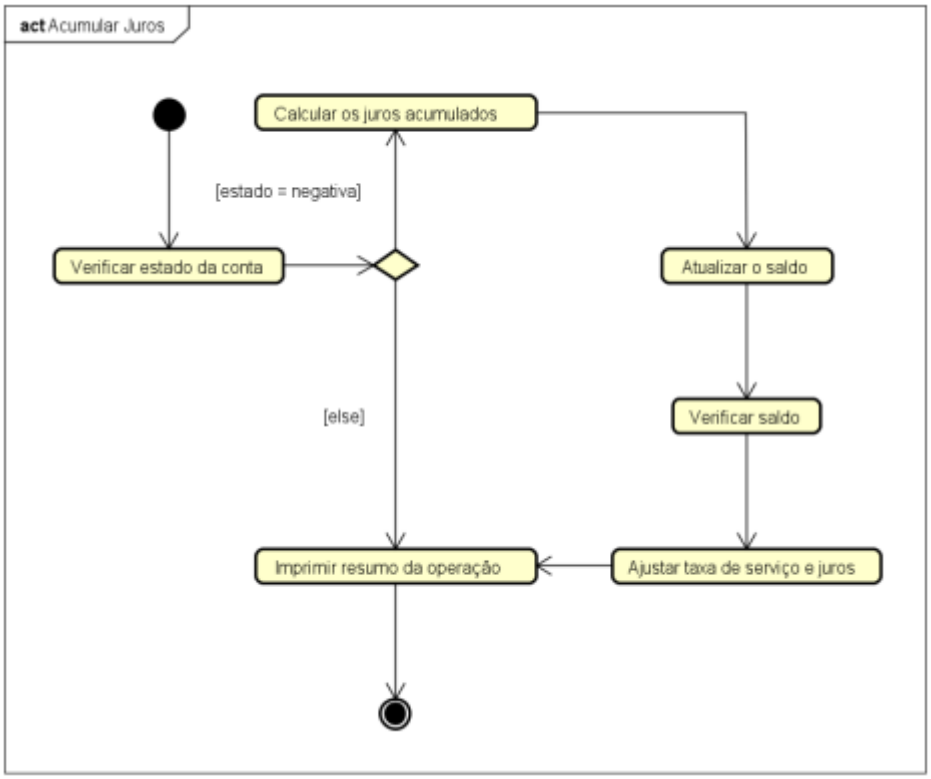
Fonte: gerado pelo autor.

Diagrama 08: Diagrama de Atividades da operação de receber juros na conta especial.



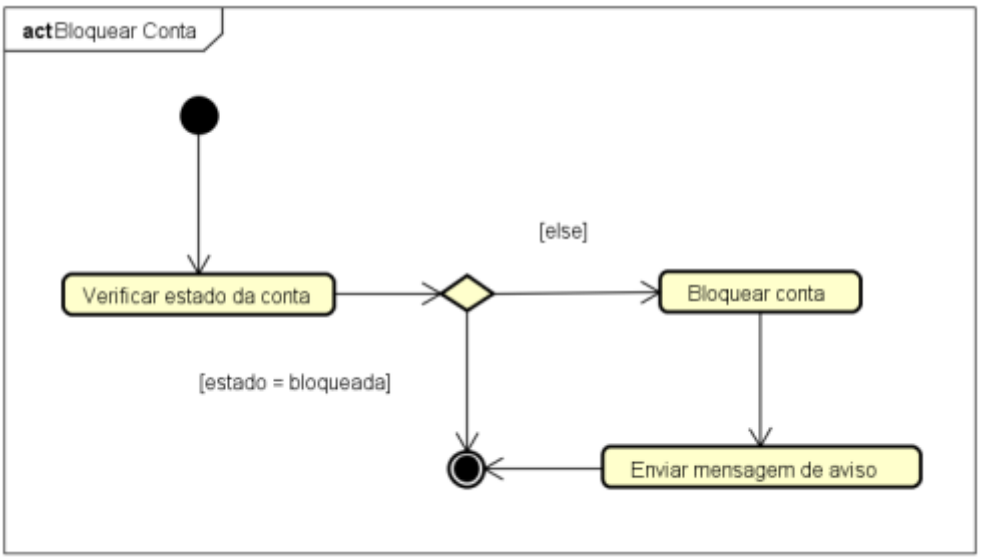
Fonte: gerado pelo autor.

Diagrama 09: Diagrama de Atividades da operação de acumular juros na conta negativa.



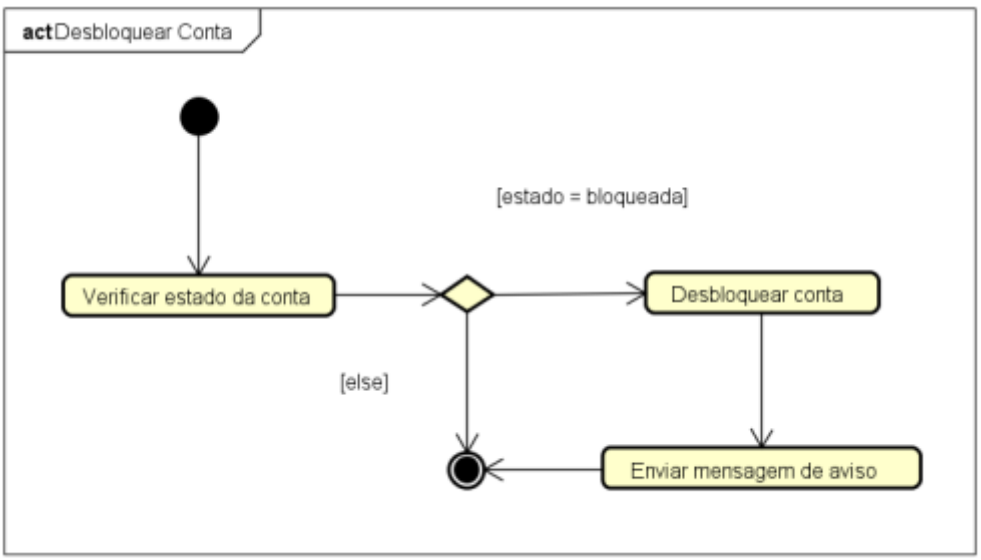
Fonte: gerado pelo autor.

Diagrama 10: Diagrama de Atividades da operação de boquear conta não bloqueada.



Fonte: gerado pelo autor.

Diagrama 11: Diagrama de Atividades da operação de debloquear conta bloqueada.



Fonte: gerado pelo autor.

5.3.1.4 SEQUÊNCIA

O diagrama de sequência mostra o funcionamento de objetos do sistema e a interação entre eles em relação ao tempo. Os objetos são representados por uma linha retratando a passagem do tempo durante a execução do sistema (BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. 2005).

Sampaio (2015) acrescenta que estes diagramas mostram, a nível de implementação, como mensagens são trocadas no decorrer da execução de uma operação, criando uma espécie de linha do tempo. Estas mensagens são enviadas através de métodos que podem, ou não, enviar mensagens de resposta dependendo da operação.

Os elementos mais comumente encontrados nestes diagramas são, ainda segundo Sampaio (2015):

- Linhas verticais representando o tempo de vida de um objeto (lifeline);

- Barras verticais que indicam exatamente quando um objeto passou a existir.

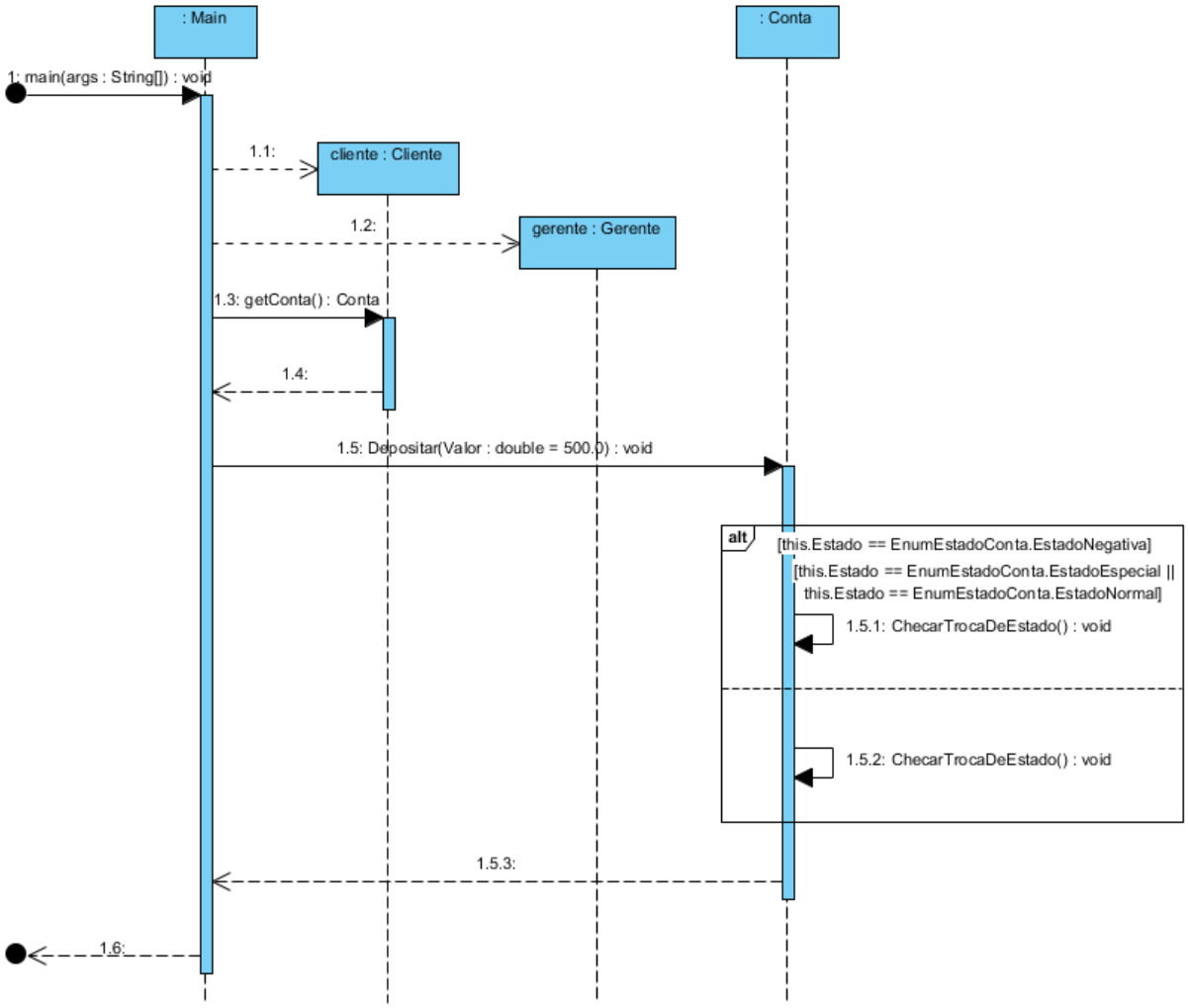
- Linhas horizontais ou diagonais representando mensagens trocadas entre objetos. Estas linhas são acompanhadas de um rótulo que contém o nome da mensagem e, opcionalmente, os parâmetros da mesma. Também podem existir mensagens enviadas para o mesmo objeto, representando uma iteração;

- Condições são representadas por uma mensagem cujo rótulo é envolvido por colchetes;

- Mesagens de retorno são representadas por linhas horizontais tracejadas. Este tipo de mensagem só deve ser mostrado quando forfundamental para a clareza do diagrama.

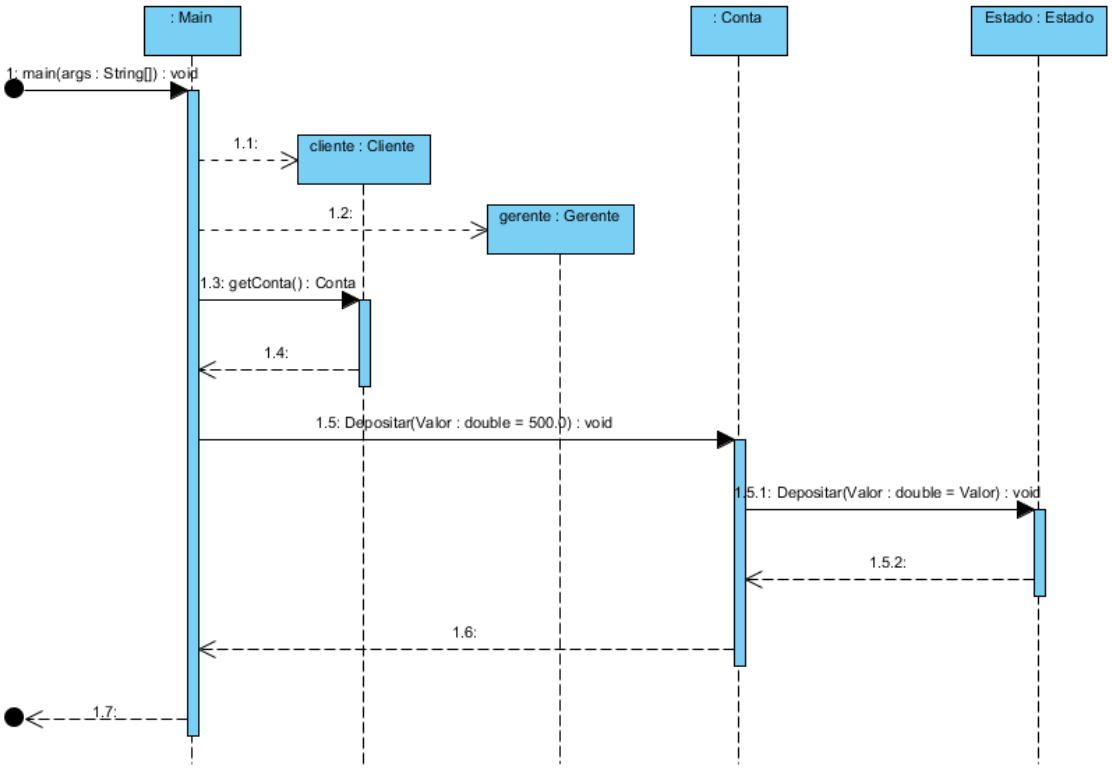
Os diagramas 12 e 13 apresentam a sequência de métodos envolvidos na operação de depositar valor na conta para o protótipo sem o padrão State, e, com o padrão State, respectivamente.

Diagrama 12: Diagrama de Sequência da operação de depositar valor na conta do protótipo sem padrão State.



Fonte: gerado pelo autor.

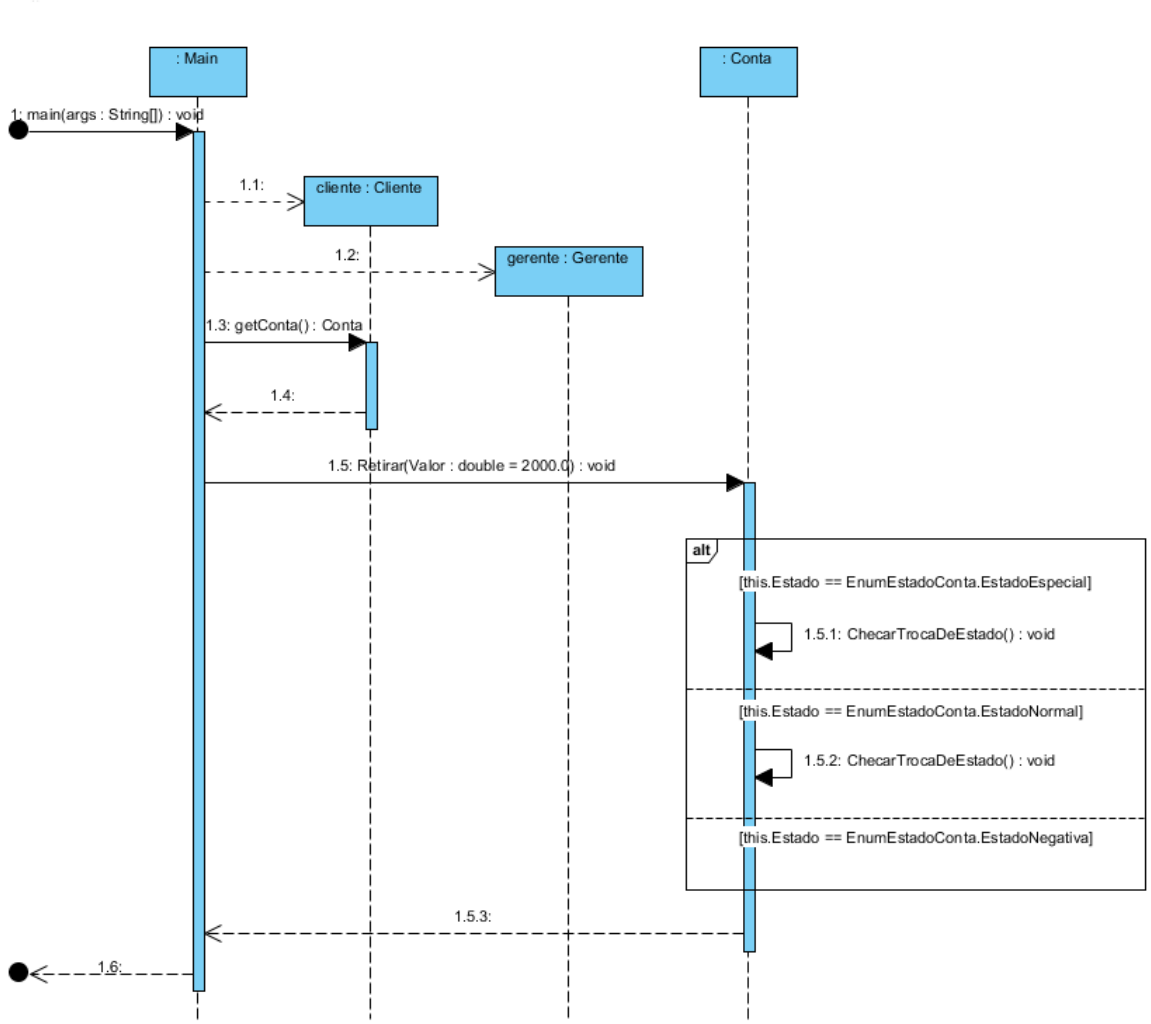
Diagrama 13: Diagrama de Sequência da operação de depositar valor na conta do protótipo com padrão State.



Fonte: gerado pelo autor.

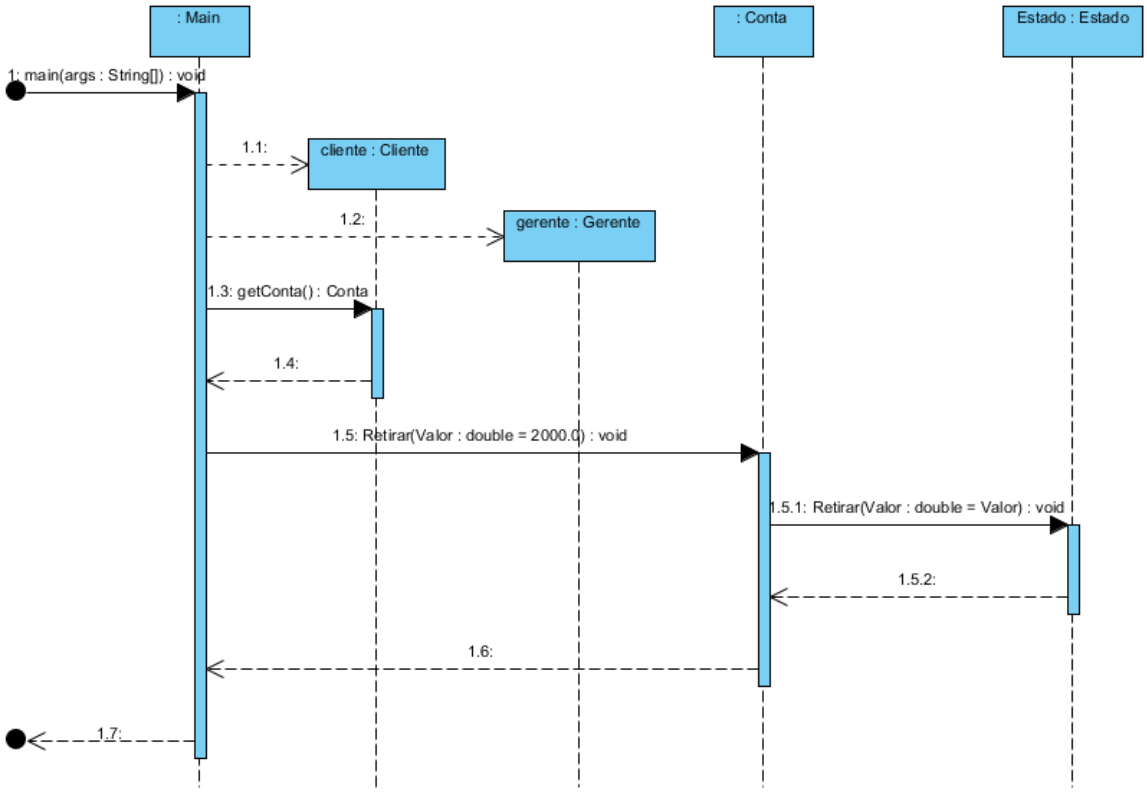
Os diagramas 14 e 15 apresentam a sequência de métodos envolvidos na operação de retirar valor da conta para o protótipo sem o padrão State, e, com o padrão State, respectivamente.

Diagrama 14: Diagrama de Sequência da operação de retirar valor da conta do protótipo sem padrão State.



Fonte: gerado pelo autor.

Diagrama 15: Diagrama de Sequência da operação de retirar valor da conta do protótipo com padrão State.



Fonte: gerado pelo autor.

5.4 ANÁLISE COMPARATIVA ENTRE OS PROTÓTIPOS

A Tabela 02 apresenta os critérios que foram levados em consideração para a realização da comparação entre os dois protótipos. Seu intuito é o de mostrar como as desvantagens do padrão State são facilmente superadas por todas as vantagens que ele tem a oferecer.

Tabela 02: Critérios de comparação dos protótipos.

Critérios de comparação	**Protótipo sem o padrão State**	**Protótipo com o padrão State**
Número de classes	7	11
Número de condicionais relacionadas ao estado do objeto	20	2
Há classes muito extensas?	Sim	Não
Há divisão de responsabilidades entre as classes?	Não	Sim
Há dificuldade para a implementação de um novo estado e/ou comportamento?	Sim	Não
Há dificuldade para a manutenção de estado e/ou comportamento já existente?	Sim	Não

Fonte: gerado pelo autor.

Os dados apresentados na tabela acima são capazes de provar que o protótipo sem o padrão State apresenta um numero elevado de desvantagens em relação ao com o padrão State, sendo superior apenas na inferioridade do número de classes geradas que é apontado logo na primeira linha.

Também é possível notar o alto número de condicionais existentes no protótipo sem o padrão State. Isso ocorre devido a necessidade de checar qual é o estado do objeto a cada vez que uma funcionalidade que depende de um estado específico é acessada.

A figura 11 ilustra o grande número de condicionais relacionados ao estado do objeto quando os métodos **Depositar** e **Retirar** são executados no protótipo que não aplica o padrão State.

Figura 11: Condicionais nos métodos do protótipo sem o padrao State.

```
public void Depositar(double Valor) {
    if (this.Estado == EnumEstadoConta.EstadoEspecial ||
        this.Estado == EnumEstadoConta.EstadoNormal) {
        this.Saldo += Valor;
        ChecarTrocaDeEstado();
    } else if (this.Estado == EnumEstadoConta.EstadoNegativa) {
        this.Saldo += (Valor - tarifaServico);
        ChecarTrocaDeEstado();
    } else {
        System.out.println(
            PrintlnColors.ANSI_RED +
            mensagemDeBloqueio +
            PrintlnColors.ANSI_RESET);
    }

    System.out.println("Conta: " + this.Id);
    System.out.println("Operação: Depósito");
    System.out.println("Valor: R$ " + Valor);
    System.out.println("Saldo: R$ " + this.getSaldo());
    System.out.println("Estado: " + this.Estado.toEstadoColorido());
    System.out.println("");
}

public void Retirar(double Valor) {
    if (this.Estado == EnumEstadoConta.EstadoEspecial) {
        this.Saldo -= Valor;
        ChecarTrocaDeEstado();
    } else if (this.Estado == EnumEstadoConta.EstadoNormal) {
        this.Saldo -= (Valor + this.tarifaServico);
        ChecarTrocaDeEstado();
    } else if (this.Estado == EnumEstadoConta.EstadoNegativa) {
        System.out.println("Impossível realizar o saque no valor de R$ " + Valor);
        System.out.println("O saldo está negativo em R$ " + this.getSaldo());
        System.out.println("");
    } else {
        System.out.println(
            PrintlnColors.ANSI_RED +
            mensagemDeBloqueio +
            PrintlnColors.ANSI_RESET);
    }

    System.out.println("Conta: " + this.Id);
    System.out.println("Operação: Saque");
    System.out.println("Valor: R$ " + Valor);
    System.out.println("Saldo: R$ " + this.getSaldo());
    System.out.println("Estado: " + this.Estado.toEstadoColorido());
    System.out.println("");
}
```

Fonte: gerada pelo autor.

Se houvesse melhor planejamento da etapa de codificação, no que diz respeito ao prévio conhecimento da existência do padrão State, o código-fonte não estaria tão estruturado e com diversas condicionais de estado. Isso significa que um padrão de projeto não necessariamente deve ser aplicado somente quando um problema é descoberto. Neste caso, como as especificações já demonstravam que o objeto Conta teria estados concretos com suas respectivas responsabilidades, o padrão State, que é um padrão caracterizado como comportamental por Gamma *et al* (2000), poderia ser aplicado ainda na etapa de implementação e testes de unidade evitando que algum problema oriundo dos estados desse objeto fosse descoberto apenas no final do projeto, ou seja, próximo da liberação do software.

Essa é outra evidência de que os padrões de projeto podem ser aplicados antes mesmo que um problema específico seja identificado, resultando em um software mais estável, com arquitetura bem definida e com baixa probabilidade de erros referentes ao contexto ao qual o padrão foi aplicado.

5.4.1 PLANEJAMENTO DA ETAPA DE CODIFICAÇÃO

O principal objetivo deste planejamento foi gerenciar o prazo da construção dos protótipos e organizar a estrutura do código-fonte dos dois protótipos desenvolvidos.

Antes de dar início à etapa de codificação foi necessário realizar o planejamento de acordo com as especificações levantadas para o funcionamento dos protótipos. Esse planejamento contou com a escolha de um dos modelos de processo de desenvolvimento de software, e, posteriormente, com a adaptação desse modelo de modo a aproveitá-lo da melhor forma possível no projeto dos protótipos.

Dessa forma, o modelo em cascata foi adaptado e utilizado no projeto dos protótipos, pois duas das características que indicam a sua utilização são encontradas, sendo elas: clara especificação do sistema e baixa probabilidade de mudanças radicais durante o desenvolvimento. Os cinco estágios do modelo foram executados e a adaptação se restringiu ao fato de não haver documentos gerados para todos os estágios do processo, tendo em vista que os protótipos são apenas para demonstrar como é o padrão State aplicado a um sistema de software orientado a objetos que possui entidades com estados e comportamentos específicos.

Além disso, para aplicar o padrão State de forma correta e segura foi necessário realizar uma série de procedimentos que são abordados ainda neste capítulo.

5.4.2 IDENTIFICAÇÃO DOS ESTADOS DAS ENTIDADES

Após o planejamento da etapa de codificação houve a necessidade de identificar todos os estados que uma entidade pode apresentar, pois a correta identificação desses estados é crucial para separar a implementação das suas responsabilidades em suas respectivas classes.

Alguns dos diagramas UML elaborados durante a etapa de especificação foram utilizados para auxiliar nesse processo de identificação, sendo eles os diagramas de classe, atividades e sequência. Em conjunto com os diagramas, os capítulos 5.1 e 5.2 também puderam contribuir com algumas especificações gerais.

Através dos diagramas de classe foi possível identificar que os estados que a entidade Conta poderia assumir eram: bloqueada, negativa, normal ou especial. Através dos diagramas de atividades e sequência foi possível identificar quando uma atividade dependia de um estado específico para ser executada, e, qual deveria ser esse estado.

Após a identificação desses estados, os protótipos puderam passar para a identificação da necessidade de refatoração de algumas classes.

5.4.3 IDENTIFICAÇÃO DA NECESSIDADE DE REFATORAÇÃO DE ALGUMAS CLASSES

Como o padrão State possui uma estrutura específica, previamente abordada no capítulo 4.2, houve a necessidade de refatoração da classe Conta para que ela pudesse atender a essa mesma estrutura.

O tipo Enum, que armazena os possíveis estados da entidade no protótipo sem o padrão State foi substituído por um tipo Estado, que representa uma nova entidade no protótipo com padrão State. Essa nova entidade possui características gerais compartilhadas por todos os estados que são seus herdeiros, além de servir de base para a criação de novas classes que possuem características específicas

como é o caso das classes: EstadoBloqueada, EstadoNegativa, EstadoNormal e EstadoEspecial.

Dessa forma essa refatoração limitou-se a alterações na classe Conta do protótipo sem o padrão State, transformando um atributo do tipo Enum em um atributo do tipo Estado. Houve a necessidade de criação de cinco novas classes, sendo quatro baseadas em cada estado identificado anteriormente e uma o estado abstrato que contém as características gerais.

Após a refatoração da classe Conta os protótipos puderam passar para a avaliação de consistência do código.

5.4.4 AVALIAÇÃO DE CONSISTÊNCIA DE CÓDIGO

A avaliação de consistência de código teve como objetivo encontrar erros referentes à lógica ou estrutura do código-fonte após a aplicação do padrão State. Este procedimento foi realizado para assegurar que o protótipo está implementado de maneira correta, segundo a utilização dos conceitos de orientação a objetos e do padrão State.

Tanto o interpretador quanto o compilador de código-fonte da ferramenta de desenvolvimento Netbeans IDE deram auxílio na identificação e análise de todos os erros capturados nas saídas das execuções dos protótipos, até que eles atendessem as especificações.

Ambos os protótipos obtiveram sucesso na avaliação da consistência de código, pois apresentaram códigos-fonte coerentes e coesos em relação ao que era esperado para o seu funcionamento.

Após o sucesso na avaliação de consistência de código os protótipos puderam passar para a avaliação de futuras manutenções.

5.4.5 AVALIAÇÃO DE FUTURAS MANUTENÇÕES

A última etapa da análise comparativa se concentra em avaliar o resultado do impacto de futuras manutenções. Seguindo uma linha de raciocínio simples, três situações foram avaliadas, sendo elas: a implementação de um novo estado, a

remoção de algum estado já implementado e a alteração de comportamento de algum estado já existente.

A primeira situação avaliada foi a implementação de um novo estado.

No protótipo sem o padrão State houve a necessidade de alterar o Enum para incluir uma entrada designada especificamente para o novo estado. Além disso, todas as condicionais que dependem de algum estado para desempanhar um comportamento também foram alteradas permitindo que o objeto tivesse o comportamento correto. Já no protótipo com o padrão State houve a necessidade de extender a classe Estado de forma a criar uma nova classe EstadoConcreto. As condicionais com estados do objeto dão lugar a condicionais com atributos desse objeto. Quando o objeto realizar alguma ação e seus atributos forem modificados a responsabilidade de trocar o estado desse objeto, através da solicitação de instância de um novo estado, fica codificada exclusivamente dentro das classes de EstadoConcreto deixando a classe Conta livre de alterações.

Isso significa que no protótipo com o padrão State não existem alterações em fora das classes de Estado, diferente do protótipo sem o padrão State que requer que alterações tanto no Enum quando nas condicionais implementadas na classe Conta.

A segunda situação avaliada foi a remoção de algum estado já implementado.

Ela é muito similar a situação de implementação de um novo estado. O processo necessário para adicionar um novo estado deve ser feito de forma reversa, ou seja, no protótipo sem o padrão State deve-se alterar o Enum e as condificionais até eliminar o estado indesejado e no protótipo com o padrão State deve-se remover a classe extendida criada e as solicitações de instâncias desse estado.

Dessa forma, o impacto de uma remoção de estado é basicamente o mesmo de uma adição de estado.

E a terceira e última situação avaliada foi a alteração de comportamento de algum estado já existente.

Neste cenário um estado já implementado teve o seu comportamento modificado de acordo com alguma especificação solicitada pelo cliente. Por exemplo, o estado ContaBloqueada deve passar a permitir que a operação de Depósito fique disponível mesmo quando o objeto assumir este estado.

No protótipo sem o padrão State foi necessário alterar apenas as condicionais da classe Conta que envolvem o estado ContaBloqueada até que o comportamento

ficasse de acordo com o que foi especificado. Já no protótipo com o padrão State foi necessário alterar a lógica imposta dentro da classe ContaBloqueada e das classes que criam instância desse estado.

Por fim, fica evidente que no protótipo sem o padrão State não existe uma centralização de código no que se refere aos estados do objeto, deixando as responsabilidades de trocas de estado para a classe Conta, que deveria apenas gerenciar informações pertinentes ao seu contexto. Além disso, são necessárias alterações consideráveis no código-fonte para implementar qualquer uma das três situações acima descritas. O protótipo com o padrão State apresentou baixo acomplamento, resultante da maneira com que ele é implementado, facilitando a implementação de qualquer uma das três situações acima descritas. Além disso, apresentou um código-fonte bem centralizado, repassando responsabilidades da classe Conta para a classe Estado e os demais EstadosConcretos.

6 CONSIDERAÇÕES FINAIS

A partir da apresentação e análise dos dados, observa-se que, com o decorrer do tempo os padrões de projeto se tornaram soluções ideais para os problemas que ocorrem com frequência no processo de desenvolvimento de um sistema de software orientado a objetos.

As pesquisas bibliográficas revelam que os padrões de projeto fornecem alternativas de reuso de soluções que já foram projetadas, testadas e se mostraram eficientes, assim, evitando alternativas que comprometam a eficiência do sistema de software.

Os padrões de projeto são classificados, basicamente, em três grupos principais: padrões de criação, que descrevem técnicas para instanciar objetos; padrões estruturais, que descrevem métodos de organização de classes e objetos em estruturas maiores; e padrões comportamentais, que descrevem métodos de atribuição de responsabilidades a classes e objetos.

O desenvolvimento deste trabalho teve como objetivo realizar um estudo sobre o padrão State no desenvolvimento de softwares orientados a objetos através de uma análise comparativa. Para tal, dois protótipos de softwares de caixas eletrônicos foram criados sob mesmas especificações, diferindo apenas na implementação do código-fonte de um deles no que diz respeito a aplicação do padrão State.

Os diagramas de UML elaborados auxiliaram de forma significativa no estágio de implementação do ciclo de vida dos protótipos. Isso porque eles demonstram uma visão geral do sistema e das suas funcionalidades de uma maneira totalmente visual.

Os resultados dessa análise foram satisfatórios e provaram que a utilização do padrão State soluciona o principal problema das implementações mais comuns de estados. Este problema refere-se a uma complexidade de código oriundo das condicionais que verificam o estado do objeto para executar a lógica correta.

Muitos critérios foram levados em consideração, sendo eles: o planejamento da etapa de codificação, a correta identificação dos estados das entidades, a identificação da necessidade de refatoração de algumas classes para que fosse possível aplicar o padrão State, a avaliação de consistência do código-fonte pós aplicação do padrão State e, por fim, o impacto de futuras manutenções.

O padrão State pode ser utilizado na construção de qualquer sistema de software orientado a objetos que possua objetos que dependam de estados para desempenhar algum comportamento. Por apresentar uma estrutura simples, pode ser aplicado sem grande dificuldade por projetistas de software e desenvolvedores que compreendam sua estrutura, que tenham conhecimento mediano sobre orientação a objetos e amplo sobre o contexto do problema.

Um padrão de projeto, como definição, é uma solução reutilizavel para algum problema específico. Logo, conclui-se que padrões de projeto podem surgir a todo momento; podendo ser uma variação de algum padrão inicialmente proposto para solucionar algo ainda mais específico ou algo totalmente novo para corrigir um novo problema.

Este trabalho tem também o objetivo de promover o padrão State propondo que ele seja utilizado pela comunidade de desenvolvedores e projetistas de software que tenham problemas para realizar manutenção em objetos que possuem estados implementados da maneira mais convencional.

REFERÊNCIAS BIBLIOGRÁFICAS

ABNT. ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **NBR 10520**: informação e documentação - citações em documentos - apresentação. Rio de Janeiro: ABNT, 2002. 7 p.

ANDRADE, Maria Margarida de, **Introdução à Metodologia do Trabalho Científico**. 10. ed. São Paulo: Atlas, 2010. p. 112-121.

BOOCH, G. et al. **Object-Oriented analysis and design with applications**. Redwood City: Benjamin Cummings, 1994. 589 p.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML**: guia do usuário. Tradução Fábio Freitas da Silva e Cristina de Amorim Machado. Rio de Janeiro: Elsevier, 2005.

CHRISTOPHER, A. et al. **A pattern language**. Oxford University Press, New York, 1977.

CHU, W. C. et al. Pattern-based software reengineering. In: **Handbook of software engineering and knowledge engineering**. Skokie, IL, EUA: Knowledge Systems Institute, 2001. v. 1, p. 767-786.

CORREA, Alexandre Luís. **Uma arquitetura de apoio para análise de modelos orientados a objetos**. Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil. Julho, 1999.

DOFACTORY, Data & Object Factory, LLC. **Getting started with the design pattern framework 4.5**. Austin, TX, EUA, DoFactory, 2013. v. 1.

_____. **Gang of four design patterns 4.5**. Austin, TX, EUA, DoFactory, 2013. v. 1, p. 67-70.

_____. **Head first design patterns for .NET**. Austin, TX, EUA, DoFactory, 2013. v. 1, 16 p.

DEUTSCH, L. P. **Frameworks and reuse in smalltalk 80 system**. In: Software Reusability: applications and experience. New York: ACM Press, 1989. v. 1, p. 57-71.

FORBES. Forbes.com Inc. **The world's biggest public companies**. Disponível em: <http://www.forbes.com/global2000/list/>. Acesso em: 25 set. 2015.

FREITAS, André Luis Castro de. **Inspeção de aplicações java através da identificação de padrões de projeto**. Dissertação de Doutorado, UFRS, Porto Alegre, RS, Brasil. Junho, 2003.

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

PIMENTEL, João Paulo. **Caixa eletrônico faz 40 anos**: curitiba foi primeira cidade brasileira a ter Banco 24 Horas, no início da década de 80. Disponível em: <www.gazetadopovo.com.br/tecnologia/caixa-eletronico-faz-40-anos-aiwkw248ujv2421mvdzen9xzi>. Acesso em: 09 ago. 2015.

GUERRA, Eduardo. **Design patterns com Java**. 1. ed. São Paulo: Casa do Código, 2013. p. 56-57.

HELDMAN, Kim. **Gerência de projetos**: fundamentos. Tradução Project Management: Jump Start. 5. ed. Rio de Janeiro: Elsevier Brasil, 2005. 344 p.

MAGELA, R. **Engenharia de software aplicada**: princípios. Rio de Janeiro: Alta Books, 2006.

PETERS, J. **Engenharia de software**: teoria e prática. Rio de Janeiro: Campus, 2001. 602 p.

PRESSMAN, Roger S. **Engenharia de software**. 5. ed. Rio de Janeiro: McGraw-Hill, 2002. p. 26-29.

THOMAZINI NETO, Luis Francisco. **Padrões de projeto**. 2006. 47f. Trabalho de conclusão de curso (Graduação em Ciência da Computação), Faculdade de Jaguariúna, Jaguariúna, 2006.

SAMPAIO, Marcus Costa. **Unified modeling language**. Disponível em: < http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/uml/>. Acesso em: 23 ago. 2015.

SEMWAL, Ashwin. **Caracterização do modelo CBSE**. Disponível em: <http://tech-kido.blogspot.com.br/2012/11/component-based-software-engineering.htm>. Acesso em: 28 jun. 2015.

SOMMERVILLE, Ian. **Engenharia de software**. 8. ed. Pearson, 2007. 569 p, 592 p.

ZADROZNY, Bianca. **Engenharia de software II.** Aula 27. Disponível em: <http://www2.ic.uff.br/~bianca/engsoft2/index_arquivos/Aula27-EngSoft2.pdf>. Acesso em: 31 jul. 2015.

APÊNDICE A – CÓDIGO-FONTE DO PADRÃO STATE IMPLEMENTADO NO PROTÓTIPO

As figuras a seguir representam o código-fonte das classes que estão relacionadas ao padrão State.

Estrutura da classe Conta.

```
package Banco;

import StatePattern.Estado;
import StatePattern.EstadoNormal;
import java.util.UUID;

/**
 *
 * @author lucas.oliveira
 */
public class Conta {

    // Tipos Comuns
    private UUID Id;

    // Tipos Específicos
    private Estado Estado;

    // Construtor
    public Conta() {
        this.Id = UUID.randomUUID();
        this.Estado = new EstadoNormal(0.0, this);
    }
```

Fonte: gerada pelo autor.

Estrutura da classe Estado – Parte 1.

```java
public abstract class Estado {

    // Tipos Comuns
    private double valorMinimo;
    private double valorMaximo;
    private double Saldo;

    // Tipos Específicos
    private Conta Conta;

    // Getters & Setters
    public double getValorMinimo() {
        return valorMinimo;
    }

    public void setValorMinimo(double valorMinimo) {
        this.valorMinimo = valorMinimo;
    }

    public double getValorMaximo() {
        return valorMaximo;
    }

    public void setValorMaximo(double valorMaximo) {
        this.valorMaximo = valorMaximo;
    }

    public double getSaldo() {
        return Saldo;
    }

    public void setSaldo(double Saldo) {
        this.Saldo = Saldo;
    }

    public Conta getConta() {
        return Conta;
    }

    public void setConta(Conta Conta) {
        this.Conta = Conta;
    }
```

Fonte: gerada pelo autor.

Estrutura da classe Estado – Parte 2.

```
    // Métodos
    public abstract void ChecarTrocaDeEstado();

    public abstract void Depositar(double Valor);

    public abstract void Retirar(double Valor);

    public abstract void ReceberJuros();

    public abstract void AcumularJuros();

    public abstract String toEstadoColorido();
}
```

Fonte: gerada pelo autor.

Estrutura da classe EstadoBloqueada – Parte 1.

```java
package StatePattern;

import Banco.Conta;
import Common.PrintlnColors;

/**
 *
 * @author lucas.oliveira
 */
public class EstadoBloqueada extends Estado {

    // Tipos Comuns
    private final String mensagemDeBloqueio = "As operações relacionadas"
            + " a esta conta encontram-se bloqueadas pela Gerência.";

    // Tipos Específicos
    private Estado estadoAntesDoBloqueio;

    // Construtores
    public EstadoBloqueada(Estado Estado) {
        this(Estado.getSaldo(), Estado.getConta());
        this.estadoAntesDoBloqueio = Estado;
    }

    public EstadoBloqueada(double Saldo, Conta Conta) {
        super.setSaldo(Saldo);
        super.setConta(Conta);
    }

    // Getters & Setters
    public Estado getEstadoAntesDoBloqueio() {
        return estadoAntesDoBloqueio;
    }

    public void setEstadoAntesDoBloqueio(Estado estadoAntesDoBloqueio) {
        this.estadoAntesDoBloqueio = estadoAntesDoBloqueio;
    }

    // Métodos Sobreescritos
    @Override
    public void Depositar(double Valor) {
        System.out.println(mensagemDeBloqueio);
    }

    @Override
    public void Retirar(double Valor) {
        System.out.println(mensagemDeBloqueio);
    }
```

Fonte: gerada pelo autor.

Estrutura da classe EstadoBloqueada – Parte 2.

```
    @Override
    public void ReceberJuros() {
        System.out.println(mensagemDeBloqueio);
    }

    @Override
    public void AcumularJuros() {
        System.out.println(mensagemDeBloqueio);
    }

    @Override
    public void ChecarTrocaDeEstado() {
        super.getConta().setEstado(estadoAntesDoBloqueio);
    }

    @Override
    public String toEstadoColorido() {
        return this.getClass().getSimpleName().replace("Estado",
                PrintlnColors.ANSI_YELLOW + "Conta ").
                concat(PrintlnColors.ANSI_RESET);
    }
}
```

Fonte: gerada pelo autor.

Estrutura da classe EstadoNormal – Parte 1.

```
package StatePattern;

import Banco.Conta;
import Common.PrintlnColors;

/**
 *
 * @author lucas.oliveira
 */
public class EstadoNormal extends Estado {

    // Tipos Comuns
    private double taxaServico;

    // Construtores
    public EstadoNormal(Estado Estado) {
        this(Estado.getSaldo(), Estado.getConta());
    }

    public EstadoNormal(double Saldo, Conta Conta) {
        super.setSaldo(Saldo);
        super.setConta(Conta);
        Inicializar();
    }

    // Métodos
    private void Inicializar() {
        this.taxaServico = 0.6;
        super.setValorMinimo(0.0);
        super.setValorMaximo(1000.0);
    }

    // Métodos Sobreescritos
    @Override
    public void Depositar(double Valor) {
        super.setSaldo(super.getSaldo() + Valor);
        ChecarTrocaDeEstado();
    }

    @Override
    public void Retirar(double Valor) {
        super.setSaldo(super.getSaldo() - (Valor + this.taxaServico));
        ChecarTrocaDeEstado();
    }

    @Override
    public void ReceberJuros() {
        // Este tipo de conta não possui juros a receber.
```

Fonte: gerada pelo autor.

Estrutura da classe EstadoNormal – Parte 2.

```
    @Override
    public void AcumularJuros() {
        // Este tipo de conta não possui juros a pagar.
    }

    @Override
    public void ChecarTrocaDeEstado() {
        if (super.getSaldo() < super.getValorMinimo()) {
            super.getConta().setEstado(new EstadoNegativa(this));
        } else if (super.getSaldo() > super.getValorMaximo()) {
            super.getConta().setEstado(new EstadoEspecial(this));
        }
    }

    @Override
    public String toEstadoColorido() {
        return this.getClass().getSimpleName().replace("Estado",
                PrintlnColors.ANSI_CYAN + "Conta ").
                concat(PrintlnColors.ANSI_RESET);
    }
}
```

Fonte: gerada pelo autor.

Estrutura da classe EstadoNegativa – Parte 1.

```
package StatePattern;

import Banco.Conta;
import Common.PrintlnColors;

/**
 *
 * @author lucas.oliveira
 */
public class EstadoNegativa extends Estado {

    // Tipos Comuns
    private double taxaServico;
    private double jurosPagar;

    // Construtores
    public EstadoNegativa(Estado Estado) {
        this(Estado.getSaldo(), Estado.getConta());
    }

    public EstadoNegativa(double Saldo, Conta Conta) {
        super.setSaldo(Saldo);
        super.setConta(Conta);
        Inicializar();
    }

    // Métodos
    private void Inicializar() {
        this.jurosPagar = 0.02;
        this.taxaServico = 15.0;
        super.setValorMinimo(-100.0);
        super.setValorMaximo(0.0);
    }

    // Métodos Sobreescritos
    @Override
    public void Depositar(double Valor) {
        super.setSaldo(super.getSaldo() + (Valor - taxaServico));
        ChecarTrocaDeEstado();
    }

    @Override
    public void Retirar(double Valor) {
        System.out.println("Impossível realizar o saque no valor de R$ " +
                Valor);
        System.out.println("O saldo está negativo em R$ " + this.getSaldo());
        System.out.println("");
    }
```

Fonte: gerada pelo autor.

Estrutura da classe EstadoNegativa – Parte 2.

```
    @Override
    public void ReceberJuros() {
        // Este tipo de conta não possui juros a receber.
    }

    @Override
    public void AcumularJuros() {
        super.setSaldo(super.getSaldo() + (super.getSaldo() * this.jurosPagar)
        ChecarTrocaDeEstado();
    }

    @Override
    public void ChecarTrocaDeEstado() {
        if (super.getSaldo() > super.getValorMaximo()) {
            super.getConta().setEstado(new EstadoNormal(this));
        }
    }

    @Override
    public String toEstadoColorido() {
        return this.getClass().getSimpleName().replace("Estado",
                PrintlnColors.ANSI_RED + "Conta ").
                concat(PrintlnColors.ANSI_RESET);
    }
}
```

Fonte: gerada pelo autor.

Estrutura da classe EstadoEspecial – Parte 1.

```
package StatePattern;

import Banco.Conta;
import Common.PrintlnColors;

/**
 *
 * @author lucas.oliveira
 */
public class EstadoEspecial extends Estado {

    // Tipos Comuns
    private double jurosReceber;

    // Construtores
    public EstadoEspecial(Estado Estado) {
        this(Estado.getSaldo(), Estado.getConta());
    }

    public EstadoEspecial(double Saldo, Conta Conta) {
        super.setSaldo(Saldo);
        super.setConta(Conta);
        Inicializar();
    }

    // Métodos
    private void Inicializar() {
        this.jurosReceber = 0.05;
        super.setValorMinimo(1000.0);
        super.setValorMaximo(1000000.0);
    }

    // Métodos Sobreescritos
    @Override
    public void Depositar(double Valor) {
        super.setSaldo(super.getSaldo() + Valor);
        ChecarTrocaDeEstado();
    }

    @Override
    public void Retirar(double Valor) {
        super.setSaldo(super.getSaldo() - Valor);
        ChecarTrocaDeEstado();
    }
```

Fonte: gerada pelo autor.

Estrutura da classe EstadoEspecial – Parte 2.

```
    @Override
    public void ReceberJuros() {
        super.setSaldo(super.getSaldo() + (super.getSaldo() *
                this.jurosReceber));
        ChecarTrocaDeEstado();
    }

    @Override
    public void AcumularJuros() {
        // Este tipo de conta não possui juros a pagar.
    }

    @Override
    public void ChecarTrocaDeEstado() {
        if (super.getSaldo() < 0.0) {
            super.getConta().setEstado(new EstadoNegativa(this));
        } else if (super.getSaldo() < super.getValorMinimo()) {
            super.getConta().setEstado(new EstadoNormal(this));
        }
    }

    @Override
    public String toEstadoColorido() {
        return this.getClass().getSimpleName().replace("Estado",
                PrintlnColors.ANSI_BLUE + "Conta ").
                concat(PrintlnColors.ANSI_RESET);
    }
}
```

Fonte: gerada pelo autor.