Aplicação de Domain-Driven Design no gerenciamento de GRU de Cronotacógrafo no Inmetro/RS

Tiago O. de Farias[1]

[1]UniRitter Laureate International Universities
Rua Orfanatrófio, 555 - Alto Teresópolis - 90840-440 - Porto Alegre - RS - Brasil

tiago.farias.poa@gmail.com

Abstract. *Since 1997, when it was instituted the Brazilian Traffic Code, cargo vehicles with a gross weight exceeding 4536 kilograms and passengers with more than 10 seats must have tachograph. From 2009, the instruments should also be checked periodically by Inmetro (National Institute of Metrology, Quality and Technology), which increases the reliability of the measurements. All the tachograph verification process is managed through a web system. This paper presents a proposal for improving the current web system using some techniques Domain-driven design. The main objective is to provide an organized, highly reusable and production structure; where the application life cycle can be extended. Throughout this work is done a study on the concept of Domain-Driven Design, in addition to the current application improvement case study.*

Resumo. *Desde 1997, quando foi instituído o Código de Trânsito Brasileiro, veículos de carga com peso bruto superior a 4.536 kg e de passageiros com mais de 10 lugares devem possuir cronotacógrafo. A partir de 2009, os instrumentos também devem ser verificados periodicamente pelo Inmetro (Instituto Nacional de Metrologia, Qualidade e Tecnologia), o que aumenta a confiabilidade das medições. Todo o processo de verificação do cronotacógrafo é gerenciado através de um sistema web. Este trabalho apresenta uma proposta de melhoria do atual sistema web utilizando algumas técnicas de Domain-Driven Design. O principal objetivo é fornecer uma estrutura organizada, altamente reutilizável e produtiva; onde o ciclo de vida da aplicação pode ser prolongado. Ao longo deste trabalho é feito um estudo sobre o conceito de Domain-Driven Design, além do estudo de caso de melhoria da atual aplicação.*

1. Introdução

O Brasil está entre os países que tornaram o uso do cronotacógrafo obrigatório em ônibus e caminhões, o instrumento inibe os excessos e ajuda a reduzir os acidentes, uma vez que registra o histórico das velocidades desenvolvidas, distâncias percorridas e tempos de movimento e paradas do veículo.

O processo de verificação tem basicamente três grandes etapas: emissão e pagamento da GRU (Guia de Recolhimento da União), realização da selagem e realização do ensaio metrológico. O sistema de gerenciamento de verificação do cronotacógrafo existe há pelo menos oito anos e atende a todos os estados da federação. Inicialmente foi concebido para simples emissão de GRU onde o proprietário do veículo acessava o site do cronotaógrafo preenchia seus dados, emitia e pagava GRU, após se dirigia a um posto

de selagem para dar inicio ao processo, ao finalizar a selagem e dentro de um período determinado deveria procurar um posto de ensaio para realizar o ensaio metrológico e assim recebia o certificado do Inmetro que garante que o instrumento atende aos padrões vigentes sob penalização de ser autuado pela policia federal.

No dia 01 de janeiro de 2016 houve uma grande mudança no processo de emissão de GRU, hoje os postos de selagem e de ensaio emitem a GRU, e semelhante a um plano pré-pago de celular, o posto compra créditos para cada GRU emitida, cada crédito permite ao posto acessar o site do Inmetro e registrar os dados do serviço realizado, seja serviço de selagem ou de ensaio metrológico.

2. Referencial Teórico

2.1. Domain-Driven Design

Domain-Driven Design é uma abordagem de desenvolvimento de software desenhado para gerir a complexa e grande escala de produtos de software [Evans 2003].

A melhor maneira de justificar uma tecnologia ou técnica é fornecer valor ao negócio [Carlos Buenosvinos and Akbary 2014].

Segundo [Evans 2003] é um processo que alinha o código do desenvolvedor com o problema real, um conjunto de técnicas de desenvolvimento de software, usadas principalmente em projetos complexos que provê conceitos e regras para ajudar no cilco do desenvolvimento de software, essas técnicas também tem o objetivo de ajudar clientes, gestores e todas as pessoas envolvidas no processo de desenvolvimento. Seguir a filosofia DDD dará aos desenvolvedores o conhecimento e as habilidades que eles precisam para enfrentar sistemas de negócios grandes e complexos de maneira eficaz [Millett and Tune 2015].

DDD utiliza o princípio da separação de conceitos. Este princípio é usado para separar a aplicação do modelo em um modelo de domínio, que consiste de domínio relacionado com a funcionalidade, e domínio independente de funcionalidade, que é representado por diferentes serviços que facilitam a usabilidade do modelo de domínio [Uithol 2008].

A solução está em resolver o problema do domínio, focado no domínio do problema, falar a mesma linguagem dos especialistas do domínio. Pode ser aplicado em qualquer projeto desde que não torne complicado o desenvolvimento, algumas vezes o desenvolvimento não é tão complexo que necessite do DDD. O foco está na criação de uma linguagem comum conhecida como a linguagem ubíqua para descrever de forma eficiente e eficaz um domínio de problemas [Millett and Tune 2015]. De acordo com o conceito do DDD o mais importante em um software não é somente o código, não é somente a arquitetura, tampouco a tecnologia sobre o qual foi desenvolvido, mas sim o problema que o mesmo se propõe a resolver, a regra de negócio. Ela é a razão do software existir. DDD é sobre o desenvolvimento de conhecimento em torno do negócio e de utilizar a tecnologia para fornecer valor [Carlos Buenosvinos and Akbary 2014].

Ainda segundo [Carlos Buenosvinos and Akbary 2014] Domain-Driven Design não é uma bala de prata, como tudo em software, depende do contexto. Como regra geral, deve ser utilizado para simplificar o domínio, nunca para adicionar mais complexidade. DDD é mais do que tecnologia ou metodologia, ou até mesmo um *framework*. É uma

maneira de pensar, é um conjunto de prioridades que visa acelerar projetos de software que têm de lidar com domínios complicados [Vlahovic].

Pensar nos problemas de forma técnica não é ruim, o único problema é que, às vezes, pensar menos tecnicamente é melhor. A fim pensar em comportamentos de objetos precisa-se pensar na Linguagem universal em primeiro lugar [Carlos Buenosvinos and Akbary 2014].

De forma geral [Carlos Buenosvinos and Akbary 2014] afirma que os benefícios com a utilização do DDD são:

- Alinhamento com o modelo do domínio
- Especialistas do domínio contribuem para o design do software
- Melhor experiência do usuário
- Limites claros
- Melhor organização da arquitetura
- Modelagem contínua de forma ágil

2.2. Linguagem Ubiqua

Que está ou pode estar em toda parte ao mesmo tempo; onipresente [Michaelis 2011].

A linguagem ubíqua é uma linguagem de equipe compartilhada. Ela é compartilhada por especialistas em domínio e desenvolvedores. Na verdade, ela é compartilhada por todos na equipe do projeto. Não importa o seu papel na equipe, uma vez que você está na equipe que usa a linguagem ubíqua do projeto [Vernon 2013]. Esta linguagem é composta de documentos, diagramas de modelo, e mesmo código. Se um termo está ausente no projeto, é uma oportunidade para melhorar o modelo incluindo-a [Evans 2003]. A linguagem ubíqua exige que os desenvolvedores trabalhem duro para entender o domínio do problema, mas também exige que a empresa trabalhe duro para ser precisa em suas nomenclaturas e descrição desses conceitos [Haywood 2009]. Se uma ideia não pode ser expressa usando este grupo de conceitos, o modelo deve ser extendido para procurar e remover as ambiguidades e as inconsistências [Haywood 2009].

Especialistas do domínio podem se comunicar com os times de desenvolvedores de sistema sobre as regras do domínio através da linguagem ubiqua que também representa a especificação formal do sistema [Vlahovic].

Para [Millett and Tune 2015] se a equipe de desenvolvimento não se envolver com especialistas de domínio para compreender plenamente a linguagem e usá-la no âmbito da implementação de código, muito do seu benefício é perdido. Para alcançar uma melhor compreensão, as equipes precisam se comunicar de forma eficaz. É a criação da linguagem onipresente que permite uma compreensão mais profunda do que vai permanecer após o código ser reescrito e substituído [Millett and Tune 2015]. Ainda segundo [Millett and Tune 2015] a utilidade da criação de uma linguagem ubiqua tem um impacto que vai além da aplicação para o produto em desenvolvimento. Ela ajuda a definir explicitamente o que a empresa faz, revela uma compreensão mais profunda do processo e a lógica do negócio, e melhora a comunicação empresarial. Enquanto as equipes estão implementando o modelo em código, novos conceitos podem aparecer. Estes termos descobertos precisam ser levados de volta para os especialistas de domínio para validação e esclarecimentos. [Millett and Tune 2015]

Um projeto enfrenta sérios problemas quando os membros da equipe não compartilham uma linguagem comum para discutir o domínio [Avram 2007]. Ele ainda defende usar um modelo como espinha dorsal de uma línguagem. Solicitando que a equipe deve usar a linguagem de forma consistente em todas as comunicações e também no código.

2.3. Mapa de Contexto

Descrever os conceitos...

2.4. Diagrama de Casos de Uso

É a especificação de uma seqûencia de interações entre um sistema e os agentes externos que utilizam esse sistema [Bezerra 2006].

O diagrama de casos de uso procura, por meio de uma linguagem simples possibilitar a compreensão do comportamento externo do sistema por qualquer pessoa, tentanto apresentar o sistema através da perspectiva do usuário [Guedes 2009].

Ainda segundo [Guedes 2009] o diagrama de casos de uso sendo uma linguagem informal, apresenta uma visão geral do comportamento do sistema a ser desenvolvido, que pode e deve ser apresentado durante as reuniões inicias com os clientes com uma forma de ilustrar o comportamento do sistema, facilitando a compreenção dos usuários e auxiliando na identificação de possíveis falhas de especificação, verificando se os requisitos do sistema foram bem compreendidos.

Sendo um cenário uma descrição de umas das maneiras pelas quais um caso de uso pode ser realizado, então um cenário é chamado também de instância de caso de uso [Bezerra 2006]. E segundo [Guedes 2009] atores e casos de uso são itens principais do diagrama de casos de uso. Atores represenam os papéis desempenhados pelos diversos usuários que poderão utilizar os serviços do sistema. Um ator também pode representar um hardware ou ouro software que interaja com o sistema, pode ser qualquer elemento externo que interaja com o software. [Guedes 2009] Segundo [Bezerra 2006] Um ator pode estar envolvido em vários casos de uso, e para cada caso de uso pode ter responsabilidades diferentes, o mais importante é que o nome dado a esse ator deve lembrar o seu papel em vez de lembrar o que o representa.

Para [Guedes 2009] um caso de uso normalmente é documentado de maneira informal, mas o engenheiro de software se considerar necessário, pode inserir detalhes de implementação em uma linguagem mais técnica e que a UML (Linguagem de Modelagem Unificada) não define um formato específico de documentação para casos de uso.

2.5. Blocos de Construção

A finalidade desses padrões é apresentar alguns dos principais elementos da modelagem de objeto e design de software do ponto de vista do design orientado por domínio [Avram 2007].

Segundo [Avram 2007] existe uma necessidade de se ter um conhecimento aprimorado do domínio e que a camada modelo, tem como principal objetivo organizar todas as informações obtidas através dos especialistas. Um desafio para os desenvolvedores é de criar um modelo que reflita os conceitos do domínio de forma clara que também seja útil para o desenvolvimento.

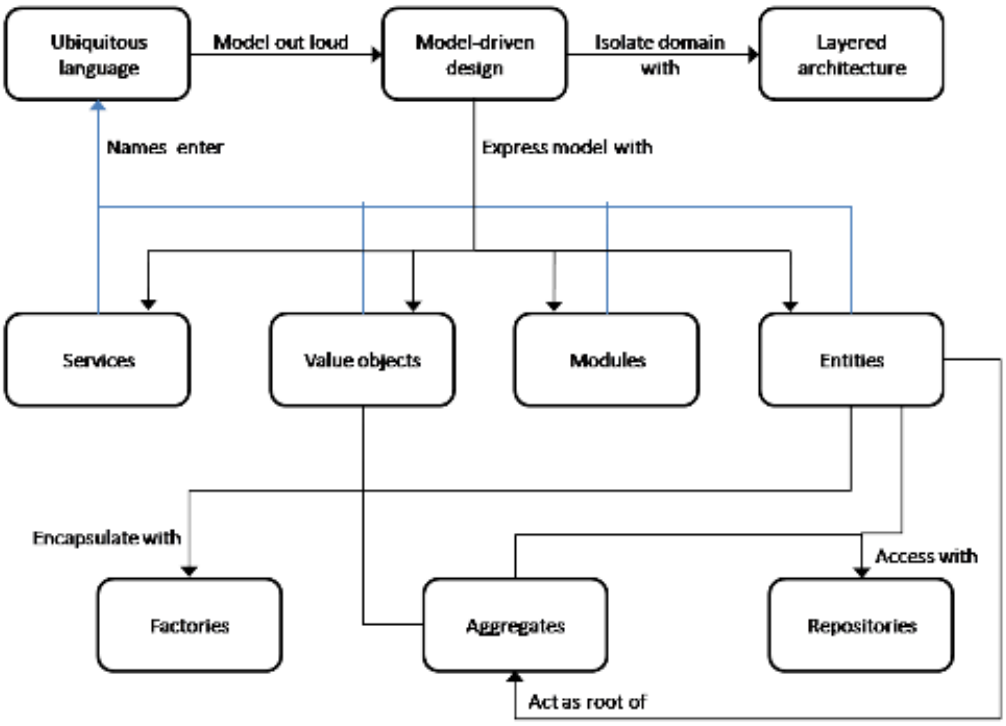

Figure 1. Padrões para auxiliar a modelagem de objetos

Os efeitos destes padrões é apresentar alguns dos elementos-chave para modelagem de objetos e software do ponto de vista no DDD. Neste caso, o domínio sendo modelado é o próprio DDD [Evans 2003].

A criação de programas que podem lidar com chamadas de tarefas muito complexas para a separação de responsabilidades, permitindo a concentração em diferentes partes do projeto isoladamente. Ao mesmo tempo, o complexo de interações dentro do sistema deve ser mantido apesar da separação [Evans 2003].

Desenvolver um projeto dentro de cada camada que seja coesa e que depende apenas de níveis abaixo. Siga os padrões de padrões de arquitetura para fornecer acoplamento frouxo para as camadas acima. Concentrar todo o código relacionadas com o modelo de domínio em uma amada e isolar a partir da interface do usuário, aplicativo e código de infra-estrutura [Avram 2007]. Normalmente encontra-se no desenvolvimento códigos de lógica comercial incorporado no comportamento de elementos na interface do usuário juntando e adicionando os scripts de acesso dados e até de retorno de informações do objeto. Isso acontece porque é a maneira mais fácil que o desenvolvedor encontra para fazer com que as funcionalidades apresentem resultados, no curto prazo. [Vernon 2013]

Observa-se que neste contexto fala-se a respeito de uma lógica comercial incorporada em elementos, ou seja não há uma separação de responsabilidade entre as camadas. Desse modo pode-se chamar de camadas, aquilo que tem uma separação de responsabilidades entre elas [Evans 2003]. Ainda segundo [Evans 2003] a arquitetura em camadas do sistema deve ser capaz de substituir uma camada que está no nível superior e colocar uma camada de outro tipo de aplicação e ela deve funcionar, isso quer dizer, se há uma aplicação WEB e adiciona uma aplicação Mobile, então todo o resto continua funcio-

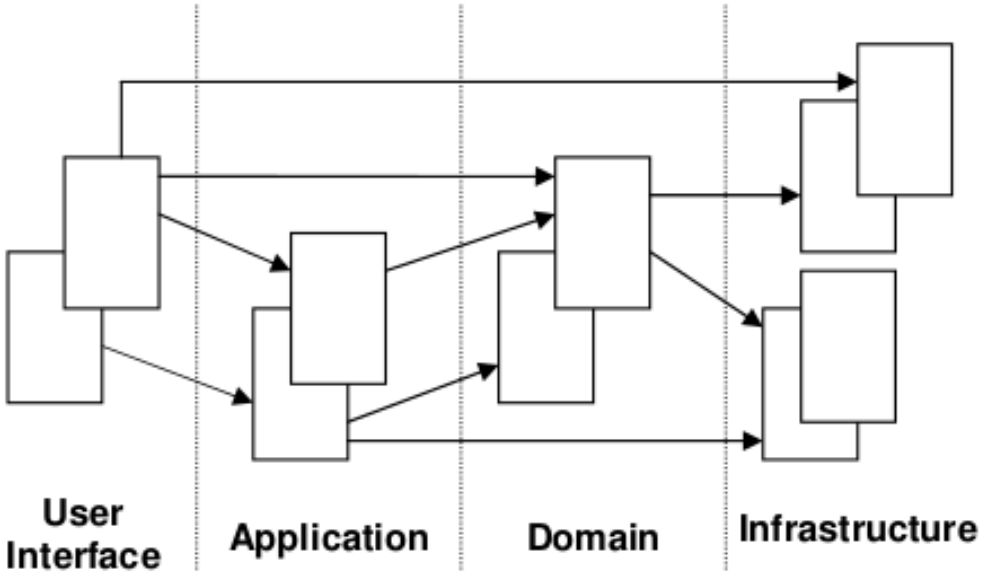

Figure 2. separação de responsabilidades

nando. Isso é possível por conta dessa separação, onde pode-se citar: Camada de acesso a dados, Camada de lógica de negócio, entre outras, sem a necessidade de recodificar tudo que já foi definido ou modelado. No entanto, quando o código é relacionado ao domínio e misturado com as outras camadas, torna-se extremamente difícil de ver e pensar [Evans 2003]. Mudanças superficiais na interface do usuário podem realmente mudar a lógica comercial. Para alterar uma regra de negócio pode exigir um rastreamento na Interface com Usuário, no código de banco de dados ou outros elementos do programa. Para evitar esse tipo de problema, deve-se isolar todo o código responsável por controlar o domínio de maneira que ele tenha uma única responsabilidade: implementar regras de negócio [Evans 2003].

No DDD a arquitetura proposta muda um pouco do MVC, embora conceitualmente nas duas divisões eles sejam bastante parecidas, algumas questões técnicas fazem com que elas sejam diferentes [Carlos Buenosvinos and Akbary 2014].

As camadas da arquitetura são descritas por [Evans 2003]:

User Interface: Responsável pela interação com o usuário, seja interpretando comandos ou exibindo informação para o mesmo. Tudo daquilo que o usuário interage diretamente, elementos visuais.

Application: Coordena as atividades da aplicação, sua responsabilidade é de trabalhar com os objetos de domínio, e não mantém estado de domino, ou seja, não contém regras de negócio. Pode por exemplo manter um estado de determinada transação. Porém ela possui uma ligação muito forte com a camada de domínio (*Domain*).

Domain: Camada que concentra toda a regra de negócio da aplicação. para [Evans 2003], esta camada é o coração de um software de negócios, ou seja DDD é utilizado em softwares de negócios com regras complexas.

Infrastructure: É a camada de mais baixo nível, responsável por interações com

infraestrutura técnica, a esta camada são colocadas as tarefas como interface com o sistemas de e-mail, banco de dados, sistemas de arquivos e outros objetos de infra-estrutura, ou seja, dar suporte tecnológico para as demais camadas.

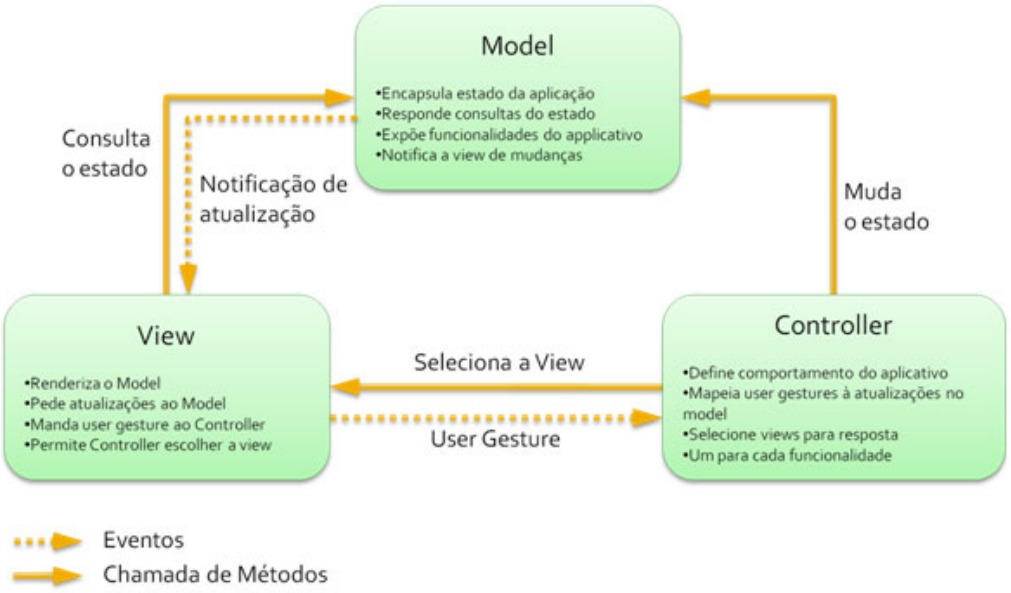

Figure 3. arquitetura MVC (Model-View-Controller)

Model View Controller (MVC) é o design pattern mais conhecido de todos. Seus conceitos remontam à plataforma Smaltalk na década de 1970. Basicamente uma aplicação que segue o pattern Model View Controller é dividida em três camadas. As letras que compõem o nome deste pattern representam cada um desses aspectos [Dall Oglio 2015].

2.6. Entidades

Muitos objetos não são fundamentalmente definidos por seus atributos, mas sim por uma linha de continuidade e identidade [Evans 2003]. Um objeto deve ser distinguido por sua identidade, ao invés de seus atributos. Esse objeto, que é definido por sua identidade, é chamado de entidade. O modelo deve definir o que significa ser a mesma coisa. Suas identidades devem ser definidas de modo que possam ser efetivamente controladas. Normalmente existem 4 maneiras de definir a identidade de uma entidade: Um cliente fornece a identidade, o próprio aplicativo fornece uma identidade, o mecanismo de persistência fornece a identidade ou outro contexto limitado fornece uma identidade [Carlos Buenosvinos and Akbary 2014].

Entidades têm considerações especiais de modelagem e arquitetura. Elas têm ciclos de vida que podem mudar sua forma e conteúdo, mas sua continuidade deve ser mantida. Suas definições de responsabilidades, atributos e associações devem girar em torno de quem elas são mais do que sobre os atributos que elas carregam. Entidades são objetos importantes de um modelo de domínio, e elas deve ser considerada a partir do começo da modelagem processo. É também importante para determinar se um objeto precisa de ser uma entidade ou não. [Avram 2007]

2.7. Objetos de Valor

Um objeto que representa um aspecto descritivo do domínio sem identidade conceitual é chamado de Objeto de Valor e entidades de implementação em software significa criar identidade [Evans 2003].

Segundo [Evans 2003] entender o *pattern Value Object* não é tão simples, já que na aplicabilidade exige abstração de orientação a objetos para identificar quando precisamos de um. Um ponto de partida é ter sempre em mente que objeto de valor é um objeto pequeno e que pode ser facilmente criado, e que não representa nenhuma entidade de domínio. Outro ponto é relacionado a igualdade, uma entidade de domínio seja na aplicação orientada a objetos ou no banco relacional, existe um modo de especificar que é único. Um objeto de valor não deve ser considerado apenas uma coisa em seu domínio. Como um valor, é medido, quantificado, ou descreve um conceito no Domínio [Carlos Buenosvinos and Akbary 2014].

Em banco de dados utiliza-se chave primária (PK), para tal identificação. Objetos de valor podem existir inúmeros objetos iguais ao mesmo tempo. Algumas classes de característica VO são: Dinheiro, Data, Coordenadas e etc [Carlos Buenosvinos and Akbary 2014].

Quando um objeto de valor é do tipo imutável no projeto, os desenvolvedores são livres para tomar decisões sobre questões como a cópia e compartilhamento em uma base puramente técnica, com a certeza de que a aplicação não depende de determinadas instâncias dos objetos [Evans 2003].

2.8. Serviços

Alguns conceitos do domínio não são naturais para modelar como objetos. Forçar a funcionalidade de domínio necessário para ser da responsabilidade de uma entidade ou valor ou distorce a definição de um objeto baseado em modelo ou adiciona objetos artificiais sem sentido (Evans, Eric 2003).

Serviços agem como interfaces para prover operações. Eles não possuem um estado interno, seu objetivo é simplesmente fornecer funcionalidades para o domínio sobre um conjunto de entidades ou objetos de valor [Avram 2007].

Declarando explicitamente um serviço, o conceito é encapsulado criando uma distinção clara no domínio. Desta forma evita-se criar a confusão de onde colocar essa funcionalidade, se em uma entidade ou objeto de valor. Serviços de domínio não possuem qualquer tipo de estado por si só [Carlos Buenosvinos and Akbary 2014].

2.9. Agregados

Agrupar as entidades e objetos de valor em agregados e definir limites em torno cada. Escolha uma entidade para ser a raiz de cada agregado, e controlar todo o acesso aos objetos dentro do limite através da raiz. Permitir que objetos externos para armazenar referências a apenas a raiz. referências transitória a membros internos pode ser passado para fora para o uso dentro de uma única operação. Porque o acesso controles de raiz, não podem ser surpreendidos por alterações para os internos. este arrangement torna prático para fazer cumprir todas as invariantes para objetos no agregado e para o Agregada, como um todo, em qualquer mudança de estado [Evans 2003]. Em um modelo dirigido por domínio,

utiliza-se o conceito de Agregado , que é um grupo de objetos associados que tratamos como sendo uma unidade para fins de alterações de dados, de forma que cada agregado possui um entidade raiz e um limite (o que está dentro do agregado). [Vernon 2013] Os agregados demarcam o escopo dentro do qual as invariantes (invariante: algo que não tem variação), tem que ser mantidas em cada estágio do ciclo de vida [Vernon 2013].

2.10. Fábricas

Criação de um objeto pode ser uma grande operação em si, mas as operações de montagem complexas não se encaixam a responsabilidade dos objetos criados. Combinando essas responsabilidades podem produzir desenhos desajeitados que são difíceis de entender. Fazendo o cliente construção directa turva a concepção do cliente, violações encapsulamento do objecto montado ou agregado, e excessivamente acopla o cliente para a implementação do objecto criado [Evans 2003]. Fábricas em DDD são as mesmas classes normalmente responsáveis pela criação de objetos complexos, onde é necessário esconder alguma implementação, onde cada operação deve ser atômica e se preocupar caso a criação do objeto falhe, minimizando a dependência e o acoplamento [Avram 2007]. Para manter o baixo acoplamento, o cliente faz uma solicitação por meio de uma factory (fábrica). A fábrica por sua vez cria o produto solicitado. Outra maneira de pensar é que a fábrica torna o produto independente de seu solicitante.[Vernon 2013] Fábricas podem criar Value Objects, que produzem objetos prontos, no formato final, ou Entity factories apenas para os atributos essenciais [Carlos Buenosvinos and Akbary 2014].

2.11. Repositórios

Um cliente precisa de um meio prático de adquirir referências a preexistente objetos de domínio. Se a infra-estrutura torna mais fácil para fazer isso, os desenvolvedores do cliente pode adicionar associações mais traversable, atrapalhando o modelo [Evans 2003]. Por outro lado, eles pode-se usar consultas para puxar os dados exatos de que precisam a partir do banco de dados, ou para puxar alguns objetos específicos, em vez de navegar a partir de raízes de agregação [Evans 2003]. Muitas regras no *domain model* acabam sendo colocadas nas consultas sql, consequentemente se perdendo da aplicação, os repositórios tentam fazer a transição entre o *domain* e a camada de persistência. Repositórios podem representar algo muito próximo aos objetos guardados na camada de persistência ou mesmo retornar calculos, somatórias ou relatórios [Vernon 2013]. Pensando em DDD, o *repository* é um padrão conceitual simples para encapsular soluções de SQL, mapeamento de dados (ORM, DAO, e etc), fábricas e trazer de volta o foco no modelo. Quando o desenvolvedor tiver construído uma consulta SQL, transmitindo essa consulta para um serviço de consultas da camada de infraestrutura, obtendo um *RecordSet* que extrai as informações e transmite para um construtor ou fábrica, o foco do modelo já não existe mais [Vernon 2013].

Repositórios representam coleções, enquanto DAOs estão mais perto do banco de dados. Normalmente, um DAO conteria métodos CRUD para um objeto de domínio particular [Avram 2007]. Coleções são importantes por serem uma forma de expressar um dos mais fundamentais tipos de variação na programação: a variação de número [Beck, Kent. 2013]. O padrão repositório representa todos objetos de um determinado tipo como sendo um conjunto conceitual. Ele age como uma coleção, exceto por ter recursos mais elaborados para consultas. Essa definição une um conjunto coeso de responsabilidades

para fornecer acesso às raízes dos agregados desde o início do clico de vida até o seu final [Carlos Buenosvinos and Akbary 2014].

2.12. Módulos

Quando você coloca algumas aulas juntos em um módulo, você está dizendo a próxima desenvolvedor que olha para o seu projeto para pensar sobre eles juntos. Se o seu modelo é contar uma história, Os módulos são capítulos [Evans 2003]. Uma preocupação comum ao criar um aplicativo segundo o DDD, é onde é colocar o código? Qual é a maneira recomendada para colocar o código no aplicativo? Onde colocar código de infra-estrutura? E mais importante, como devem os diferentes conceitos dentro do modelo ser estruturado? Há um padrão tático para isso: módulos [Carlos Buenosvinos and Akbary 2014].

Módulo é responsável pela representação de todos os fluxos de uma aplicação. Desde a sua criação até que ele seja entregue ao cliente. Além disso, é uma applicação independente [Carlos Buenosvinos and Akbary 2014].

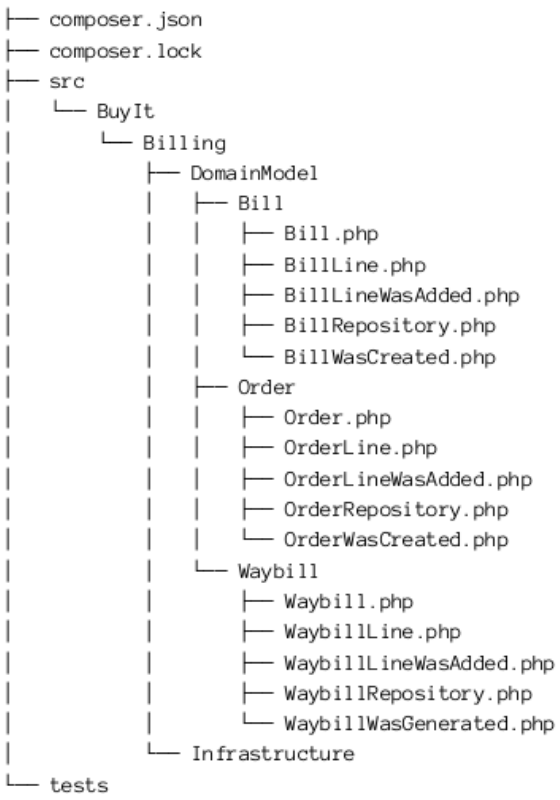

Figure 4. agrupando o domínio em módulos

Dar aos módulos nomes que passam a fazer parte da linguagem ubíqua e seus nomes devem refletir uma visão sobre o domínio [Evans 2003].

Módulos não separam o código mas separam o significado conceitual [Carlos Buenosvinos and Akbary 2014].

3. Estado da Arte

3.1. Domain-driven design in action: designing an identity provider

`http://www.diku.dk/forskning/performance-engineering/Klaus/speciale.pdf`

Nesta tese de mestrado, os princípios de *Domain-Driven Design* são usados para modelar um problema de negócio do mundo real, um provedor de identidade extensível (PDI). O trabalho mostra que a aplicação de *Domain-Driven Design* é uma boa abordagem para a modelagem de um domínio complexo. O projeto do trabalho é desenvolvido na empresa Safewhere e tem como interessados além do autor do trabalho também o diretor técnico e o CEO da Safewhere.

O trabalho cita que no início, a empresa por ter pouco capital para investir, ainda tinha como prioridade máxima mostrar um produto real. Dessa forma não houve comprometimento com design de software e modelagem. Desenvolvedores eram contratados, e cada um tinha seu próprio estilo de desenvolvimento e compreensção do domínio comercial.

E, que no inicio não parecia causar grandes problemas e a versão do produto foi entregue e totalmente funcional. Após o lançamento inicial a necessidade de novos recursos cresceu rapidamente. O produto estava funcionando bem, mas se tornou cada vez mais dificil adicionar novos recursos em funcionalidades existentes, como refatorar o código base. Apesar de orientações de codificação estabelecidas, havia algo faltando no processo de desenvolvimento. Uma das maneiras de como isso se manifesta em si foi que tínhamos várias classes na di entravarão um conjuntos com nomes semelhantes mas com significado desiguais. Foi também difícil explicar a arquitectura para novos membros da equipe, dado que o código não foi organizado de forma heterogênea. Em retrospecto, o problema foi que claramente não tinha um modelo nem um idioma onipresente.

Para ter a certeza de ter conseguido fazer uma fácil manutenção e um modelo extensível o autor solicitou a alguns dos colegas para rever o modelo. Entrevistou cada pessoa para que avalia-se a qualidade do design e os benefícios do DDD:

- Resolve bem o problema do negócio.
- Representa uma linguagem ubíqua concisa e expressiva, o que torna mais fácil de discutir.
- É flexível e tem uma clara separação de responsabilidades entre as camdas, tornando fácil refatorar
- Conduz a um estável e maduro do produto final.

A solução implementada pelo autor vem ao encontro da proposta de solução neste artigo uma vez que aplica os conceitos de *Domain-Driven Design* para tentar normalizar o processo de desenvolvimento de software. Sobretudo para garantir a manutenibilidade do sistema e padronizar o processo de codificação. Um dos pontos importantes que é citado pelo autor é o fato de repassar o conhecimento a novos membros da equipe, o atual artigo foca bastante na linguagem ubiqua, no caso é utilizado diagramas de classe e diagramas de caso de uso, respeitando os papéis dos atores. Isso tudo ainda ajuda para que o conhecimento sobre o negócio não fique preso a uma ou duas pessoas. E, o fator

tempo de desenvolvimento, é diretamente prorporcional ao entendimento das regras de negócio.

O maior valor do sistema de gerenciamento de GRU está no foco no negócio principal da empresa. As telas que serão apresentadas ao usuário final terão mais relação com a operação de negócio.

3.2. Implications of Domain-driven Design in Complex Software Value Estimation and Maintenance using DSL Platform

`http://www.inase.org/library/2015/zakynthos/bypaper/COMPUTERS/COMPUTERS-35.pdf`

Neste artigo é apresentado e analisado a ferramenta DSL. Plataforma de DSL é um serviço que permite a concepção, criação e manutenção de aplicações de negócio. O objetivo deste artigo é analisar as implicações do uso do DDD através da plataforma DSL sobre vários aspectos importantes do gerenciamento de software.

O objetivo deste trabalho é investigar possíveis benefícios da adopção de *Domain-Driven Design* no gerenciamento de software, com especial ênfase na fase de manutenção durante a produção de ativos de software. Inevitavelmente estas considerações irá reflectir sobre o valor do ativo de software e assim uma abordagem validada estimatida de ativos de software é chamado para implementar as principais características do objeto. Usando *Domain-Driven Design* pode ser utilizado para criar uma plataforma unificada para de desenvolvimento e evolução de sistemas de software complexos.

A Plataforma DSL permite a a automação do processo de desenvolvimento de aplicativos de negócios. A plataforma utiliza o modelo de negócio específico como entrada e saídas de componentes acabados para software de negócios correspondente sistema.

Dois principais desafios que podem ser eficazmente solucionados com Plataforma DSL e abordagem DDD subjacente é a eliminação de falta de comunicação entre clientes e desenvolvedores em equipes de desenvolvedores.

A outra é a eliminação do trabalho repetitivo realizado pelos desenvolvedores ao automatizar tarefas repetitivas no processo de desenvolvimento.

O principal obstáculo impedindo a maior aceitação de *Domain-Driven Design* na prática, é a falta de compreensão dos benefícios do DDD e as ferramentas potenciais que ele fornece.

Assim como no sistem de gerenciamento de GRU, a equipe de desenvolvimento às vezes esbarra na questão de conhecimento sobre a tecnologia que esta sendo utilizada para codificação, embora muitas as vezes a compreensão do domínio esteja bem disolvida na equipe.
comparar com o artigo atual

3.3. Evaluating Domain-Driven Design for Refactoring Existing Information Systems

http://dbis.eprints.uni-ulm.de/1373/1/Masterarbeit%20Herbert%20Hayato%20Hess.pdf

Esta tese de mestrado descreve a utilização de *Domain-Drive Design* na refatoração e evolução de um sistema legado na empresa MERCAREON.

O autor cita que o principal desafio é que o DDD foi concebido para aplicação em novos sistemas, e não para refatoração de sistemas existentes. E foi necessário fazer um mapeamento da arquitetura antiga no sentido de obter conhecimento necessário para utilizar o *Design-Domain Driven* de uma forma eficiente. Se fez necessário criar camadas que impedissem o vazamento de informações independentes para a nova arquitetura.

Embora o sistema de gestão esteja sendo executado com êxito, a empresa MERCAREON decidiu alterar a arquitetura do sistema introduzindo Domain-Driven Design. A justificativa se origina do sistema de ser muito antigo. Outro fator é a comunicação, pois para aumentar esse problema, existem desenvolvedor na Alemanha e na Polônia o que prejudica tanto pela distância física quanto nos termos utilados pela equipe de desenvolimento.

O sistema de TSM, em especial foi criado com uma arquitetura em camadas tradicionais, que sofre continuamente de complexidade. O Autor ainda afirma que como em qualquer arquitetura de refatoração, a arquitetura sugerida pelo DDD também pode sofrer pela complexidade, mas menos do que as propostas tradicionais. Durante o processo de refatoração foi definido utilizar linguadem ubiqua que dee ser utilizada para qualquer comunicação e deve-se mantê-la atualizada.

Um ponto importante a ser observado durante a criação do mapa de contexto é que delimitar contextos também ajuda a pensar em como distribuir as diferentes equipes.

comparar com o artigo atual

3.4. Architectural Improvement by use of Strategic Level Domain-Driven Design

http://dddcommunity.org/wp-content/uploads/files/practitioner_reports/landre_einar_2006_part1.pdf

Neste artigo é apresentado a utilização de *Domain-Drive Design* na empresa de petróleo Statoil ASA situada na Noruega, com foco nos conceitos do nível estratégico. O objetivo é aumentar a arquitetura corporativa utilizando a extensão da arquitetura empresarial e melhoria de software existente de empresarial para melhorar a arquitetura de software de um grande sistema corporativo.

O artigo cita a utilização de mapas de contexto e que a utilização foi de grande valor também pelo conhecimento adquirido, permitiu melhorar o escopo de novos projetos arquiteturais e melhoria do softwre existente de forma controlada.

O autor afirma que grandes empresas não tem um único núcelo. Mas que por outro lado, a nível de projeto, deve-se sempre ter um núcleo e que assim é melhor para conhecer o domínio principal e os objetivos.

A partir da análise da criação do mapa de contexto, um novo objetivo foi estabelecido e utilizado posteriormente para definir o escopo de um novo projeto, e este objetivo é a melhoria da arquitetura em novos projetos, gerenciamento e controle de recursos de documento.

Os contextos reais são derivados da Arquitetura corporativa, e transformando a arquitetura da empresa em uma ferramenta útila para a arquitetura de software de melhoria.

Um mapa de contexto é um desenho que documenta contextos de modelagem e suas relações. Grandes sistemas contém vários contextos de modelagem, então, eles representaram no contexto de modelagem de interesses, não as aplicações ou sistemas de informação que implementam os diferentes contextos. O autor ainda afirma que mapa de contexto representa o primeiro ítem no nível estratégico *Domain-Driven design*. Para o autor ficou a percepção de que a utilização do mapa de contexto e o seu papel na melhoria da arquitetura foi bem compreendido pela equipe, e que o sistema *Digital Cargo File (DCF)* foi refatorado em 2006 para corresponder as recomendações sugeridas.

A empresa Statoil também adotou formalmente o uso de mapas de contexto como um artefato da arquitetura. O autor finaliza afirmando que o uso de camadas de responsabilidade parece reduzir a complexidade percebida.

No presente artigo sobre o gerenciamento de GRU de Cronotacógrafo, este sistema representa apenas um núcleo de todo o ecossistema que envolve o processo de verificação de cronotaógrafo. A GRU participa de vários contextos durante o processo de verificação de cronotacógrafo. Dentre estes contextos, a GRU possui responsablidades diferentes dependendo do cenário onde se encontra ao realizar selagem, realizar ensaio, subsituir selos, cancelamento de GRU, emissão de relatórios de ensaio, etc.

3.5. Diretrizes para desenvolvimento de linhas de produtos de software com base em Domain-Driven Design e métodos ágeis

`http://www.teses.usp.br/teses/disponiveis/55/55134/tde-24032009-192923/publico/dissertacao.pdf`

Esta dissertação de mestrado tem como principal objetivo demonstrar através de um estudo de caso a aplicação de DDD (Domain-Driven Design) e métodos ágeis no desenvolvimento de linhas de produtos de software, a fim de analisar as principais vantagens em relação aos métodos tradicionais.

O autor afirma que o processo de desenvolvimento em linhas de produtos de software (LPS) geralmente é sequencial e que o projeto não está focado em um modelo de domínio, mas mais intenssamente preocupado com questões técnicas, como alocação de componentes e separação em subsistemas. O aumento no reúso do software leva a um aumento por transformações nos sistemas.

Os métodos tradicionais não incorporam o conceito de adaptabilidade a mudanças no seu corpo de princípios. Por não reconhecerem as mudanças como parte natural do desenvolvimento de software, esses métodos tentam evitá-la ao máximo, por meio de especificações prematuras e exaustivas.

O trabalho apresenta um estudo de caso, em que se desenvolve uma linha de produtos baseados em DDD para sistemas de transporte urbano. O BET (Bilhetes Eletrônicos

de Transporte) é um protótipo de sistema que tem por objetivo gerenciamento de sistemas de transporte urbano onde propõe-se um conjunto de diretrizes de desenvolvimento de LPS. Segundo o autor são regras de caráter prático que visam a guiar o desenvolvimento de novas linhas de produtos. Adaptadas a cada contexto particular, as diretrizes podem ajudar a alcançar maior flexibilidade e extensibilidade nas LPS.

O estudao de caso considera que as regras para transporte urbano variam de cidade para cidade, essas diferenças foram modeladas como variabilidades da linha, enquanto as regras e conceitos comuns foram desenvolvidos como parte do núcleo.

Foi priorizado um backlog contendo as histórias de usuário ordenadas por prioridade. Conforme sugere o DDD o sistema foi arquitetado em quatro camadas: apresentação, aplicação, domínio e infra-estrutura, e ainda sob o conceito do MVC (Model-View-Controller). A camada de aplicação é composta por classes de serviço que coordenam outros serviços ou objetos do domínio, na camada de infra-estrutura encontram-se classes para persistencia de dados e onde objetos de domínio não conhecem nada sobre persistencia. Toda essa organização favorece ainda a injeção de dependência.

Além disso, a prática de modelagem adotada parte do pressuposto de que não é vantajoso tentar prever possíveis cenários futuros no modelo enquanto eles não forem necessários. Por isso, a cada iteração, o modelo reflete apenas o conhecimento necessário para o desenvolvimento das funcionalidades previstas em suas respectivas histórias de usuário A criação das histórias e a conseqüente priorização do backlog foi feita com a ajuda de uma pessoa com bastante conhecimento sobre o domínio em questão Durante a implementação do sistema BET utilizando como base os princípios de DDD e métodos ágeis, surgiram vários problemas práticos de projeto. O projeto foi realizado com quatro Ao contrário de métodos tradicionais onde é encorajado criar o modelo completo de todo o projeto logo no início, baseado nas *features* que são as funcionalidades previstas para o sistema crescem a cada iteração.

De modo geral, o processo utilizado começa com um entendimento superficial do domínio, seus principais conceitos e regras. Neste ponto, o conhecimento obtido não é suficiente para pre- ver todos os desafios que o projeto apresentará. No entanto, esse contato inicial é de extrema importância. A partir dele, o conhecimento pode ser refinado, de modo a dar origem a um modelo de domínio iterativo. Tendo esse conhecimento superficial sobre o domínio, os desenvolvedores podem começar a planejar a primeira iteração. É importante que a implementação comece o mais cedo possível, porque a existência de software funcionando (mesmo que incompleto e sem interface amigável, por exemplo) é o que dá chance ao cliente ou especialista no domínio de dar uma resposta e guiar os desenvolvedores no caminho correto. Assim, quaisquer erros ou não-conformidades podem ser detectados tão logo sejam criados.

O estudo de caso sugere que é possível construir linhas de produtos fundamentadas em um modelo de domínio mais expressivo, comparado ao que propõem os métodos tradicionais. A ex- pressividade do modelo tem um grande impacto na facilidade de manutenção de uma linha de produtos e, conseqüentemente, na sua evolução. Com isso, a adaptabilidade a mudanças ao longo do tempo é bem maior. Além disso, um modelo de domínio expresso diretamente no código da aplicação reduz a necessidade de documentação não executável, como diagramas UML e docu- mentos de casos de uso.

Durante o desenvolvimento do projeto, houve uma preocupação em aumentar a coesão das classes e reduzir o acoplamento entre elas. Para tal, foram utilizadas várias técnicas e princípios, como polimorfismo e princípio da responsabilidade única. O *design* resultante mostrou-se bastante flexível e extensível. A inclusão ou alteração de regras de negócio ou mesmo de fluxos inteiros de atividade (como a funcionalidade de crédito em lote) podem ser feitas facilmente. A configuração das funcionalidades requeridas para um produto consiste apenas de configuração do grafo de dependências, por meio de uma linguagem específica de domínio.

A aplicação de DDD ao desenvolvimento de linhas de produtos de software mostrou-se bastante adequada. A razão disto é o enfoque do desenvolvimento baseado em uma linguagem ubíqua do domínio. Destarte, os conceitos do domínio revelam-se com muito mais clareza no código da LPS. O efeito mais relevante dessa transparência é uma maior facilidade de manutenção da linha a curto e longo prazo. As principais vantagens observadas na abordagem proposta em relação aos métodos tradicionais são: menor esforço na implementação de uma nova regra de negócios (em geral, uma nova classe, comparada a vários componentes e classes no modelo de componentes da implementação de referência) e maior capacidade de reúso, pela atribuição adequada de responsabilidades, pela divisão em vários níveis de granularidade (compatíveis com o nível de abstração da classe) e pela aplicação de princípios de orientação a objetos, como o princípio de substituição de Liskov e o princípio aberto-fechado.

Em algumas situações, o uso de uma linguagem ubíqua pode trazer problemas, embora estes não tenham sido experimentados neste trabalho. Trata-se dos casos em que as linguagens utiliza- das no contexto de cada produto sejam diferentes.

Como a equipe de desenvolvimento não tinham experiência anterior com o DDD e ainda tem de manter o actual sistema de TSM foi decidido contra a arquitectura refactoring do todo o sistema de TSM. Por esta razão, MERCAREON decidiu que apenas como um primeiro passo de um pequeno módulo, o Live Yardview, é criado usando a proposta de refatoração técnicas baseadas em DDD para estabelecer uma arquitetura sustentável bem. Este módulo vai coexiste com o actual sistema de TSM. Quando concluído, o módulo delimitada contexto é ser continuamente expandida até chegar a uma verdadeira arquitectura refatoração.

Isto é possível porque a DDD é baseado em um modelo adaptável que por sua vez é adaptada às novas necessidades utilizando o protótipo. Como um trabalho futuro, os benefícios da arquitetura refatoração contínua rumo ao novo sistema de informação DDD pode ser avaliado.

4. Solução Implementada

4.1. Modelando a Aplicação

Afim de auxiliar a comunicação entre os analistas e o cliente, o sistema e suas funcionalidades são apresentados conforme figura 5.

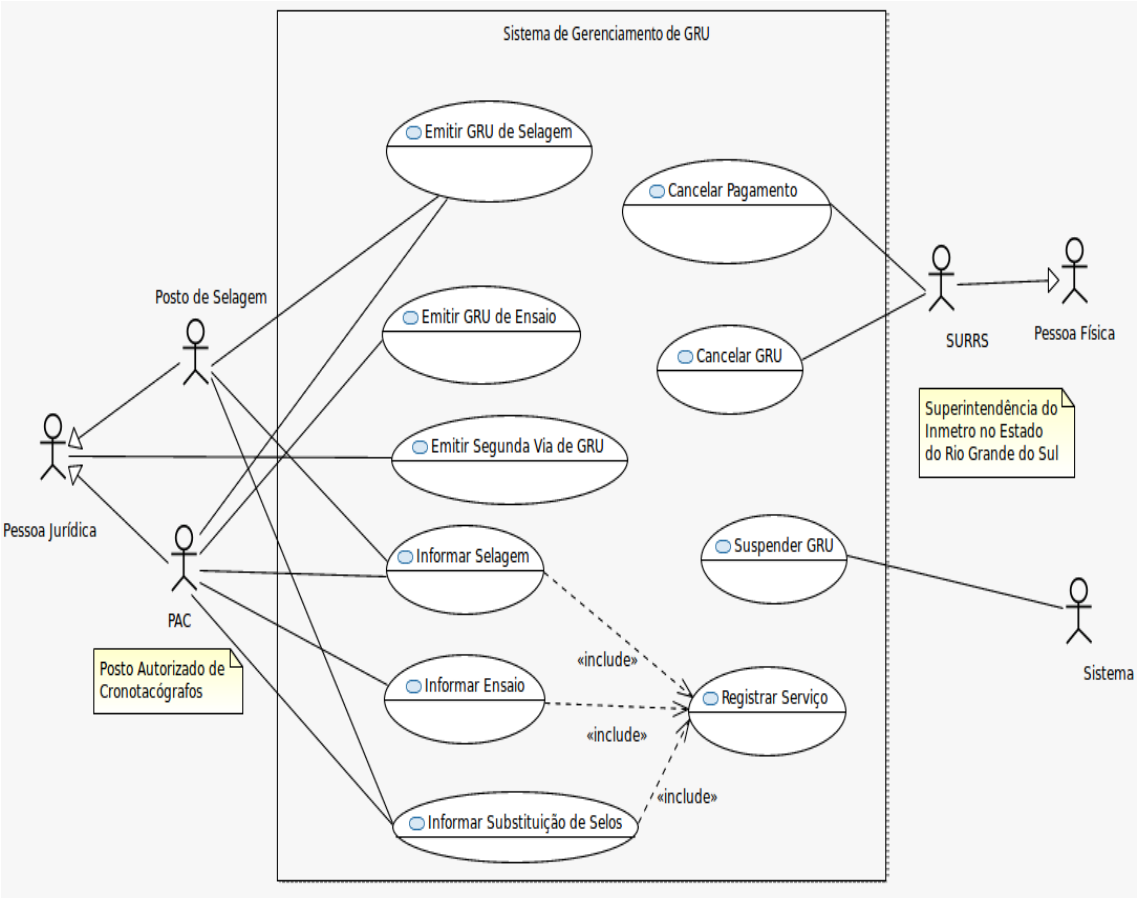

Figure 5. Padrões para auxiliar a modelagem de objetos

Table 1. Documentação do Caso de Uso Suspender GRU

Nome do caso de Uso	Suspender GRU
Caso de Uso Geral	Emitir GRU
Ator Principal	Posto
Atores Secundários	-
Resumo	Este caso de uso descreve as etapas percorridas por um usuário para emitir uma GRU de selagem.
Pré-condições	Usuário autenticado
Pós-condições	Realizar o pagamento da GRU
Fluxo Principal	
1. Informar a quantidade de créditos para cada GRU emitida	2. Calcular o valor a ser pago
Restrições/Validações	1. Usuário autenticado deve ser um posto de Selagem ou PAC

Table 2. Documentação do Caso de Uso Cancelar GRU

Nome do caso de Uso	Cancelar GRU
Caso de Uso Geral	Emitir GRU
Ator Principal	Posto
Atores Secundários	-
Resumo	Este caso de
Pré-condições	Usuário autenticado
Pós-condições	Realizar o pagamento da GRU
Fluxo Principal	
1. Informar a quantidade de créditos para cada GRU emitida	2. Calcular o valor a ser pago
Restrições/Validações	1. Usuário autenticado deve ser um posto de Selagem ou posto de Ensaio

Table 3. Documentação do Caso de Uso Cancelar Pagamento

Nome do caso de Uso	Cancelar GRU
Caso de Uso Geral	Emitir GRU
Ator Principal	Posto
Atores Secundários	-
Resumo	Este caso de
Pré-condições	Usuário autenticado
Pós-condições	Realizar o pagamento da GRU
Fluxo Principal	
1. Informar a quantidade de créditos para cada GRU emitida	2. Calcular o valor a ser pago
Restrições/Validações	1. Usuário autenticado deve ser um posto de Selagem ou posto de Ensaio

Table 4. Documentação do Caso de Uso Informar Selagem

Nome do caso de Uso	Cancelar GRU
1. Usuário autenticado deve ser um posto de Selagem ou posto de Ensaio	

Table 5. Documentação do Caso de Uso Informar Substituição

Nome do caso de Uso	Cancelar GRU
Caso de Uso Geral	Emitir GRU
Ator Principal	Posto
Atores Secundários	-
Resumo	Este caso de
Pré-condições	Usuário autenticado
Pós-condições	Realizar o pagamento da GRU
Fluxo Principal	
1. Informar a quantidade de créditos para cada GRU emitida	2. Calcular o valor a ser pago
Restrições/Validações	1. Usuário autenticado deve ser um posto de Selagem ou posto de Ensaio

Table 6. Documentação do Caso de Uso Informar Ensaio

Nome do caso de Uso	Cancelar GRU
Caso de Uso Geral	Emitir GRU
Ator Principal	Posto
Atores Secundários	-
Resumo	Este caso de
Pré-condições	Usuário autenticado
Pós-condições	Realizar o pagamento da GRU
Fluxo Principal	
1. Informar a quantidade de créditos para cada GRU emitida	2. Calcular o valor a ser pago
Restrições/Validações	1. Usuário autenticado deve ser um posto de Selagem ou posto de Ensaio

5. Referencias

References

Avram, A. (2007). *Domain-driven design Quickly*. Lulu. com.

Bezerra, E. (2006). *Princípios De Análise E Projeto De Sistemas Com Uml-3ª Edição*, volume 3. Elsevier Brasil.

Carlos Buenosvinos, C. S. and Akbary, K. (2014). *Domain-Driven Design in PHP*. Leanpub. Real examples written in PHP showcasing DDD Architectural Styles, Tactical Design, and Bounded Context Integration.

Dall Oglio, P. (2015). *Php-programando com orientaçao a objetos*. Novatec Editora.

Evans (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Guedes, G. T. (2009). Uml 2. *Uma Abordagem Prática, São Paulo, Novatec*.

Haywood, D. (2009). *Domain-driven design using naked objects*. Pragmatic Bookshelf.

Michaelis, D. (2011). Disponível em: http://michaelis.uol.com.br. *Acesso em 28/05/2016*, 7.

Millett, S. and Tune, N. (2015). Patterns, principles, and practices of domain-driven design.

Uithol, M. (2008). Security in domain-driven design.

Vernon, V. (2013). *Implementing domain-driven design*. Addison-Wesley.

Vlahovic, N. Implications of domain-driven design in complex software value estimation and maintenance using dsl platform.