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 system is used a framework that provides an architecture based on the MVC architecture. This paper presents a proposal for restructuring the current architecture according to the techniques of Domain-Driven Design. Throughout this work is done a study on the concept of Domain-driven design and how it helps in the field of identification, as well as the delegation of the architectural point of view responsibilities, beyond the case study of the organization of GRU module tachograph.*

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, neste sistema é utilizado um framework que oferece uma arquitetura baseada na arquitetura MVC. Este trabalho apresenta uma proposta de reestruturação da arquitetura atual segundo as técnicas de Domain-Driven Design. Ao longo deste trabalho é feito um estudo sobre o conceito de Domain-Driven design e como ele ajuda na identificação do domínio, bem como a delegação de responsabilidades do ponto de vista arquitetural, além do estudo de caso da nova organização do módulo de GRU de Cronotacógrafo.*

1. Introdução

Cronotacógrafo é o instrumento ou conjunto de instrumentos destinado a indicar e registrar, de forma simultânea, inalterável e instantânea, a velocidade e a distância percorrida pelo veículo, em função do tempo decorrido, assim como os parâmetros relacionados com o condutor do veículo, tais como: o tempo de trabalho e os tempos de parada e de direção [Inmetro 2011] e o Brasil está entre os países que tornaram o uso do cronotacógrafo obrigatório em ônibus e caminhões, pois registra o histórico das velocidades, distâncias percorridas e tempos de movimento e paradas do veículo [Inmetro 2011].

O processo de verificação do Cronotacógrafo tem basicamente três grandes etapas: emissão e pagamento da GRU (Guia de Recolhimento da União), realização da selagem e realização do ensaio metrológico. As informações do processo de verificação de Cronotacógrafo são gerenciadas através de um sistema web que inicialmente foi concebido para simples emissão de GRU onde o proprietário do veículo acessava o site do cronotaógrafo preenchia os dados pessoais e dados sobre o veículo, emitia e pagava GRU, após se dirigia à um posto de selagem para dar inicio ao processo, ao finalizar a selagem e dentro de um período determinado deve procurar um posto de ensaio para realizar o ensaio metrológico e assim recebia o certificado do Inmetro que garante que o instrumento juntamente com o veículo atendem aos padrões vigentes sob penalização de ser autuado pela polícia federal.

A cada etapa do processo de selagem e de ensaio o posto deve registrar no site do Cronotacógrafo as informações sobre os serviços realizados.

O sistema web de gerenciamento de verificação do cronotacógrafo existe há pelo menos oito anos e atende à todos os estados da federação. Foi desenvolvido em PHP com o *framework CakePHP*. Aplicações bem escritas em *CakePHP* seguem o *design pattern* MVC (*Model-View-Controller*) [CakePHP 2008]. O *framework* permitite uma programação orientada a objetos, mas é muito comum, por culpa do desenvolvedor ver código escrito de forma estruturada, implementação de regras de negócio dentro da camada *view* e a mesma regra duplicada ou triplicada dentro de *controllers*, classes que tem o papel apenas de repositório de funções e que não tem se quer relação com o propósito da classe. Algumas classes e métodos só fazem sentido para o desenvolvedor que a criou. As classes são altamente acopladas, e por isso alterações mesmo que pequenas se não bem testadas causam um grande efeito colateral que se propaga em outros módulos do sistema. Algumas informações para serem utilizadas no site do Cronotacógrafo tem como origem o SGI (Sistema de Gestão Integrada) que possui banco de dados próprio, e alguns termos que tem o mesmo significado sofreram alteração de nome no banco de dados do Cronotacógrafo, causando mais dificuldades quando os desenvolvedores tentam fazer os dois sistemas se comunicarem.

No dia 01 de janeiro de 2016 houve uma grande mudança no processo de emissão de GRU, atualmente os postos de selagem e de ensaio (PAC) 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 Cronotacógrafo e registrar os dados do serviço realizado, seja serviço de selagem ou de ensaio metrológico.

Essa alteração afetou diretamente a codificação no sistema, o código e a arquitetura que antes já eram complicados, ficou maior e mais complexo, pois as regras antigas deveriam ser mantidas sem alteração, e o sistema teve que comportar muitas regras novas, os desenvolvedores tiveram que adquirir conhecimento das novas regras de negócio e que algumas vezes ao serem implementadas se confundiam com regras antigas.

O objetivo deste artigo é fornecer uma estrutura organizada, altamente reutilizável e produtiva para o módulo de GRU, pois todas as etapas de verificação dependem da GRU, e utilizando os conceitos de *Domain-Driven Design* que segundo [Evans 2003] reúne um conjunto de conceitos, princípios e técnicas cujo foco esta no domínio e na lógica do domínio para que o ciclo de vida da aplicação possa ser prolongado.

Como roteiro, este artigo encontra-se estruturado da seguinte maneira. A seção

2.1 apresenta o conceito sobre DDD (*Domain-Driven design*). Na seção 2.2 é descrito o significado e importância da linguagem ubíquia. A seção 2.3 apresenta a importância da criação de um mapa de contexto. Na seção 2.4 descreve a importância dos blocos de construção e seus elementos para auxiliar na modelagem do modelo do domínio. A seção 2.5 descreve o que são entidades. A seção 2.6 descreve as características de um objeto de valor. Na seção 2.7 é apresentado o conceito de serviços. Na seção 2.8 descreve como trabalhar com objetos complexos através de fábricas. A seção 2.9 é descrito como utilizar repositórios para acessar dados externos. Na seção 2.10 é apresentado o conceito de módulos. Na seção 3 é apresentado trabalhos relacionados ao *Domain-Driven design* e as soluções propostas. A seção 4 descreve a solução referente ao problema apresentado neste artigo utilizando *Domain-Driven design*. Na seção 5 é discutido as limitações e dificuldades na implementação de *Domain-Driven design*.

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 [Vernon 2013].

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 Ubíqua

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 ubíqua que também representa a especificação formal do sistema [Vernon 2013].

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 ubíqua 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

Além de demonstrar como o modelo se relaciona dentro de outros contextos, permite também delimitar a aplicabilidade de um modelo do domínio para que a equipe de desenvolvimento tenha um conhecimento mais claro e que possa ser compatilhado de forma eficiente [Evans 2003].

Segundo [Evans 2003] e [Millett and Tune 2015] é de vital importância para proteger a integridade de cada modelo definir claramente os limites da sua responsabilidade no código.

Ainda segundo [Evans 2003], o mapa de contexto é uma ferramenta de modelagem muito importante no *Domain-Driven Design* e quando se pensa em vários subsistemas de uma aplicação como *Bounded Contexts*, se perde a visão geral do negócio como um todo, e que é inevitável que os vários delimitadores de contextos de uma aplicação serão necessários para comunicar ou compartilhar dados entre eles.

Um mapa de contexto é uma visão geral da aplicação como um todo. Cada contexto delimitado (*Bounded Context*) se encaixa dentro de um mapa de contexto para mostrar como eles devem se comunicar entre si e como os dados devem ser compartilhados.

Para [Millett and Tune 2015], *Bounded Contexts* permitem que se possa dividir um problema grande e complexo em problemas de menor complexidade e assim pode-se focar nas particularidades de forma individual sem interferências externas, o que permite ainda criar um linguagem em torno do problema específico que todos tem como definicção clara de cada termo relevante.

Normalmente em aplicações do meio web tem diferentes definições em contextos diferentes da aplicação. Dividindo a aplicação em contextos limitados, pode-se ter a certeza que as idéias, terminologias e conceitos da aplicação estarão plenamente entendidos [Millett and Tune 2015].

2.4. 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].

A figura 1 representa os *patterns* que ajudam na modelagem do *domain model* com o objetivo de apresentar alguns elementos-chave para modelagem de objetos e software do ponto de vista do DDD e que segundo [Evans 2003], 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.

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.

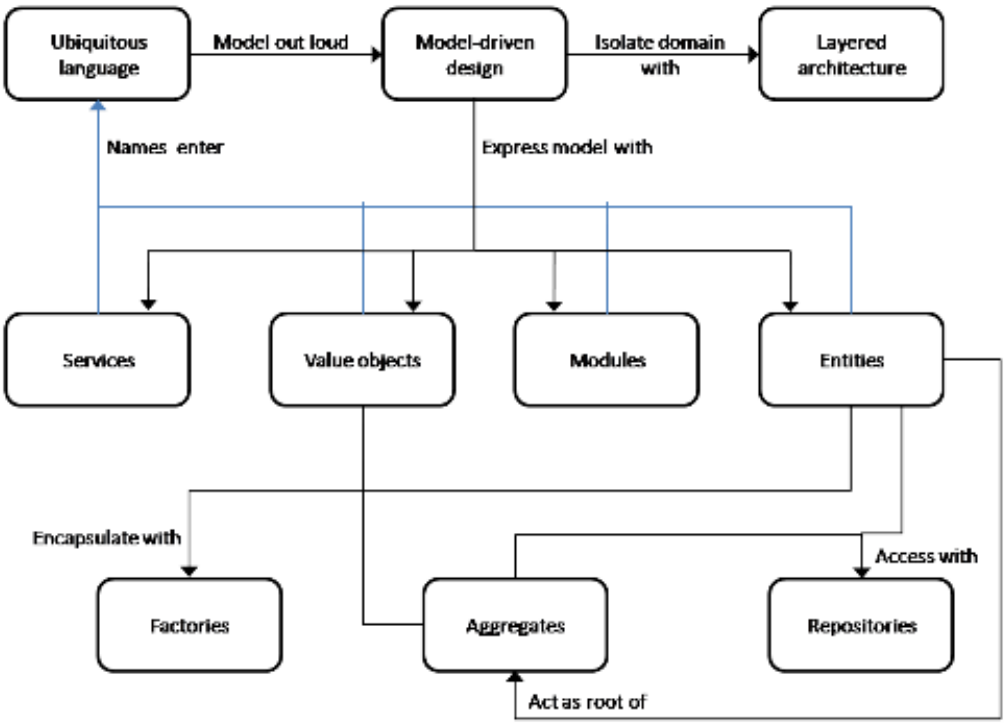

Figure 1. Padrões para auxiliar a modelagem de objetos

Os efeitos destes padrões é apresentar alguns dos elementos-chave para modelagem de objetos e software do ponto de vista do 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 e seguir os padrões de arquitetura para fornecer acoplamento flexível para as camadas acima. Concentrar todo o código relacionado com o modelo de domínio em uma camada e isolar a partir da interface do usuário, aplicação 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 a 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].

A figura 2 representa a separação em camadas da arquitetura, que segundo [Evans 2003] e [Vernon 2013] para que seja realmente funcional deve seguir algumas

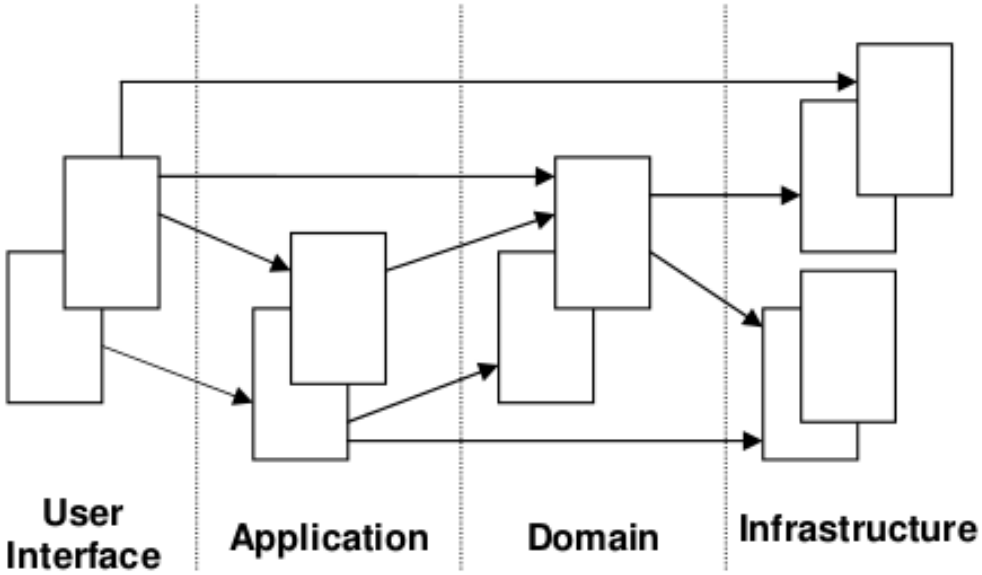

Figure 2. separação de responsabilidades

premissas: completamente isolada uma da outra, a comunicação entre elas deve ocorrer através de interfaces e a camada inferior não deve executar operações na camada superior. Ainda segundo [Vernon 2013] a arquitetura de camandas não se resume apenas na reusabilidade de componentes ou numa arquitetura que evolui juntamente com a aplicação mas no fato que ela oforece maior manutenibilidade.

Para [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 funcionando. 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.

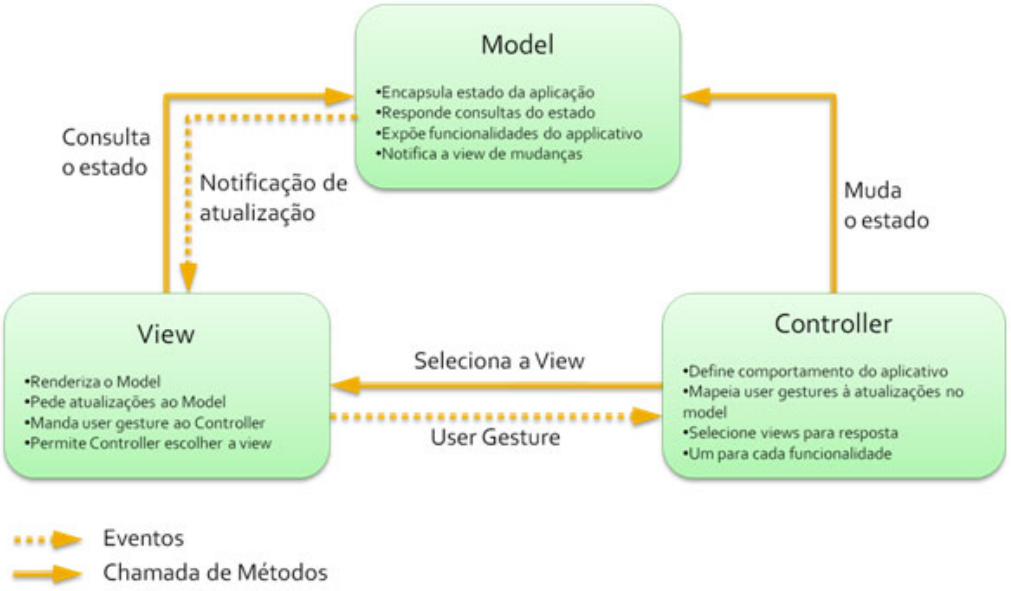

Figure 3. arquitetura MVC (Model-View-Controller)

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].

2.5. 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 [Vernon 2013]. Entidades são objetos importantes de um modelo de domínio, e ela 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]

2.6. 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].

2.7. Serviços

Alguns conceitos do domínio não são naturais para modelar como objetos [Vernon 2013]. Forçar a funcionalidade de domínio necessária 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 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].

2.8. Fábricas

Segundo [Avram 2007] a 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.

No domínio, fábricas ajudam a criar objetos complexos e agregados e definir uma interface para criar um objeto, mas deixar a escolha do seu tipo na subclasse, a criação a ser definida em tempo de execução [Carlos Buenosvinos and Akbary 2014].

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].

2.9. Repositórios

Um cliente precisa de um meio prático para obter referências e persistir objetos de domínio. Se a infra-estrutura torna mais fácil para fazer isso, os desenvolvedores do cliente podem adicionar associações mais acopladas, atrapalhando o modelo [Evans 2003]. Por outro lado, pode-se usar consultas para buscar os dados exatos de que precisam a partir do banco de dados, ou para buscar alguns objetos específicos, em vez de navegar a partir das 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 cálculos, 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 *Record-Set* 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 ciclo de vida até o final [Carlos Buenosvinos and Akbary 2014].

2.10. Módulos

Para aplicações complexas, o modelo tende a ficar muito grande, tornando-se difícil de ser compreendido como um todo [Carlos Buenosvinos and Akbary 2014]. Por esse motivo, é preciso reorganizar o modelo em módulos. Existem várias formas de coesão entre elementos de um módulo. Os módulos são capítulos [Evans 2003]. Uma preocupação comum ao criar uma aplicação segundo o DDD, é onde é colocar o código, qual é a maneira recomendada para colocar o código na aplicação, onde colocar código de infra-estrutura? E o mais importante, como devem os diferentes conceitos dentro do modelo ser estruturados. Existe 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 aplicação independente [Carlos Buenosvinos and Akbary 2014].

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

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 compreensão do domínio [Hoffmann 2009].

Ainda segundo [Hoffmann 2009] se o design não é pensado desde o início, e todos os envolvidos na programação solucionam os problemas do domínio conforme compreendem, existe grandes chances de acabar com um código confuso que é difícil de compreender, manter e extender.

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 manifestou foi que havia várias classes na aplicação em 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 *Domain-driven design*:

- 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.

Do posto de vista do desenvolvedor muitas vezes o modelo não fica tão completo quanto do ponto de vista do especialista do domínio e assim o modelo produzido pelo dono da obra não é tão rico e comunicativo quanto o produzido junto ao especialista do domínio [Hoffmann 2009].

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

Neste artigo é apresentado e analisado a ferramenta DSL. A Plataforma de DSL é um serviço que permite a concepção, criação e manutenção de aplicações de negócio [Vlahovic 2015]. O objetivo do artigo é analisar as implicações do uso do *Domain-driven design* através da plataforma DSL sobre vários aspectos importantes do gerenciamento de software.

Segundo [Vlahovic 2015] o objetivo é investigar possíveis benefícios da adoção de *Domain-Driven Design* no gerenciamento de software, com especial ênfase na fase de manutenção durante a produção de ativos de software. [Vlahovic 2015] afirma que inevitavelmente estas considerações iram refletir sobre o valor do ativo de software e assim é ncessário uma abordagem válida para implementar as principais características do objeto [Vlahovic 2015]. Usando *Domain-Driven Design* pode ser utilizado para criar uma plataforma unificada para de desenvolvimento e evolução de sistemas de software complexos [Vlahovic 2015].

A Plataforma DSL permite a a automação do processo de desenvolvimento de aplicativos de negócios [Vlahovic 2015]. A plataforma utiliza o modelo de negócio específico como entrada e saídas de componentes acabados para software de negócios correspondente sistema [Vlahovic 2015].

Para [Vlahovic 2015] são dois os principais desafios que podem ser eficazmente solucionados com Plataforma DSL e abordagem do DDD. 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 [Vlahovic 2015].

Segundo [Vlahovic 2015] 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.

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

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

Segundo [Hess 2016] o principal desafio é que o *Domain-Driven Design* 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 [Hess 2016]. Outro fator citado por [Hess 2016] é a comunicação, pois para aumentar esse problema, existem desenvolvedores na Alemanha e na Polônia o que prejudica tanto pela distância física quanto nos termos utilizados pela equipe de desenvolimento.

O sistema de TSM, em especial foi criado com uma arquitetura em camadas tradicionais, que sofre continuamente de complexidade [Hess 2016]. [Hess 2016] ainda afirma que como em qualquer arquitetura de refatoração, a arquitetura sugerida pelo *Domain-Driven Design* também pode sofrer pela complexidade, mas menos do que as propostas tradicionais. Durante o processo de refatoração foi definido utilizar linguadem ubíqua que deve 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 [Hess 2016].

Como a equipe de desenvolvimento não tinha experiência anterior com o *Domain-Driven design* e ainda tem que manter o atual sistema de TSM a MERCAREON decidiu que apenas como um primeiro passo de um pequeno módulo, o Live Yardview, que foi criado usando a proposta de técnicas de refatoração baseadas em *Domain-Driven Design* para estabelecer uma arquitetura sustentável [Hess 2016]. Este módulo vai coexistir com o atual sistema de TSM. Quando concluído, o módulo a partir do limite do contexto deverá ser continuamente expandido até chegar a uma verdadeira arquitectura de refatoração.

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

Neste artigo é apresentado a utilização de *Domain-Driven Design* na empresa de petróleo *Statoil ASA* situada na Noruega, com foco nos conceitos do nível estratégico. Segundo [Landre et al. 2006] o objetivo é aumentar a arquitetura corporativa utilizando a extensão da arquitetura empresarial e melhoria de software existente 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.

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 [Landre et al. 2006].

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 [Landre et al. 2006].

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 [Landre et al. 2006]. O autor ainda afirma que mapa de contexto representa o primeiro ítem no nível estratégico *Domain-Driven design*. Para [Landre et al. 2006] 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.

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

Esta dissertação de mestrado tem como principal objetivo demonstrar através de um estudo de caso a aplicação de *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.

Segundo [Macedo 2009] 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.

O trabalho apresenta um estudo de caso, em que se desenvolve uma linha de produtos baseados em *Domain-Driven Design* 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 [Macedo 2009]. Segundo [Macedo 2009] 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.

Segundo foi [Macedo 2009] priorizado um backlog contendo as histórias de usuário ordenadas por prioridade. Conforme sugere o *Domain-Driven Design* o sistema foi arquitetado em quatro camadas: apresentação, aplicação, domínio e infra-estrutura, e ainda sob o conceito do MVC (Model-View-Controller).

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 e durante a implementação do sistema BET utilizando como base os princípios de *Domain-Driven Design* e métodos ágeis, surgiram vários problemas práticos de projeto [Macedo 2009].

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 expressividade 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 documentos de casos de uso [Macedo 2009].

Durante o desenvolvimento do projeto, [Macedo 2009] descreve que 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 [Macedo 2009].

Para [Macedo 2009] a aplicação de *Domain-Driven Design* no 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. 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 [Macedo 2009]. Segundo [Macedo 2009] 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 [Macedo 2009].

3.6. Quadro Comparativo

A tabela 1 apresenta um quadro comparativo baseado na solulção dos artigos apresentados nesta seção, e que resume a conceito central utilizado em cada artigo na solução do problema do domínio segundo o *Domain-Driven design*.

Table 1. Soluções apresentadas por cada artigo baseados no *Domain-Driven design*

	Linguagem Ubíqua	Mapa de Contexto	Arquitetura
[Hoffmann 2009]	X		
[Vlahovic 2015]	X		
[Hess 2016]	X	X	
[Landre et al. 2006]	X		X
[Macedo 2009]	X	X	X
Gerenciamento de GRU	X	X	X

4. Solução Implementada

4.1. Diagramas de Casos de Uso

O atual sistema web de gerenciamento de verificação do cronotacógrafo que tem como ferramenta de desenvovlimento o *framework CakePHP* que inicialmente era responsável apenas pela simples emissão de GRU, então era um cenário simples de manter até por conta da regra de negócio que também era simples, fácil era identificar o propósito das classes e seus métodos, e por consequência se tornava fácil implementar as alterações de regras de negócio. Outro fator relevante é que somente um desenvolvedor era o responsável pela manutenção do sistema. Ao passar dos anos com a popularização da utilização de serviços web e a maior facilidade do acesso ao público à internet não somente pelo computador de casa mas também através de dispositivos móveis, fez com que no Cronotacógrafo muitos serviços que não existiam ou que eram realizados de forma manual passassem a ser realizados pelo atual sistema. E como o Inmetro do estado do Rio Grande do Sul é o orgão que mantém a equipe responsável por atender e regulamentar o processo de verificação de Cronotacógrafo abrangendo todos os estados do país, o sistema passou a ter que suportar muitas novas funcionalidades, funcionalidades estas que refletem diretamente desde a emissão da GRU até a emissão do certificado final.

Assim houve a necessidade de se aumentar aos poucos a equipe de desenvolvimento e muitas vezes é difícil alinhar o conhecimento, manter o mesmo padrão no desenvolvimento do código, refatorar, realizar testes unitários, mesmo porque novos programadores são vinculados a equipe e outros acabam saindo.

Entendendo que o framework provê uma arquitetura MVC sugerindo uma separação de responsabilidades, e se entendia também que isso era o suficiente para manter o código organizado do ponto de vista do domínio do problema, mas conforme o domínio foi aumentando e ficando complexo a estrutura do *framework* se manteve a mesma, apenas acumulando classes e funções dentro das pastas *Controllers*, *Models* e *Views*.

Algumas informações do sistema do Cronotacógrafo tem origem em outro sistema, o SGI (Sistema de Gestão Integrada) que possui banco de dados próprio, e alguns termos que tem o mesmo significado sofreram alteração de nome no banco de dados do Cronotacógrafo, causando mais dificuldades quando os desenvolvedores tentam fazer os dois sistemas se comunicarem. No dia 01 de janeiro de 2016 houve uma grande mudança no processo de emissão de GRU, atualmente os postos de selagem e de ensaio (PAC) 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 Cronotacógrafo e registrar os dados do serviço realizado, seja serviço de selagem ou de ensaio metrológico.

Essa alteração afetou diretamente a codificação no sistema, o código e a arquitetura que antes já eram complicados, ficou maior e mais complexo, pois as regras antigas deveriam ser mantidas sem alteração, e o sistema teve que comportar muitas regras novas, os desenvolvedores tiveram que adquirir conhecimento das novas regras de negócio e que algumas vezes ao serem implementadas se confundiam com regras antigas.

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

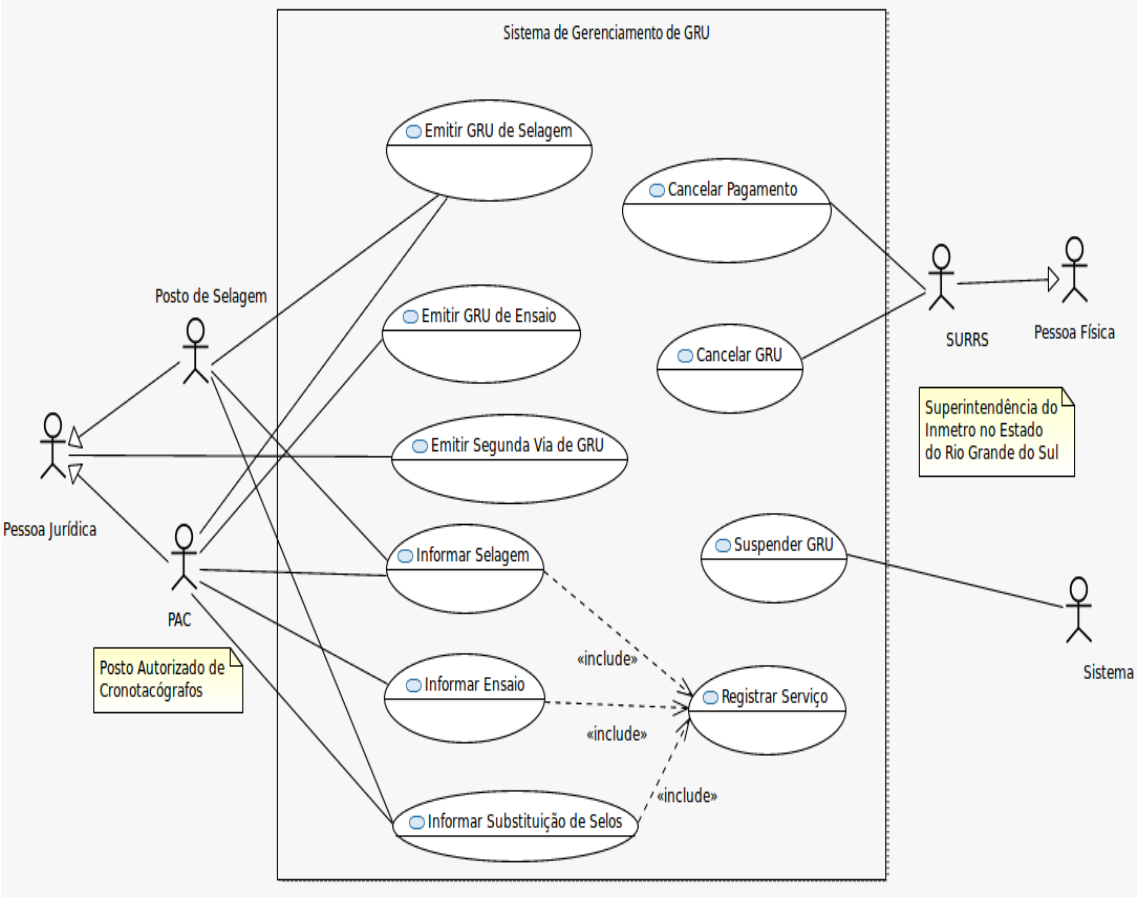

Figure 4. Padrões para auxiliar a modelagem de objetos

Table 2. Documentação do Caso Emitir Gru de Selagem

Nome do caso de Uso	Emitir Gru de Selagem
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	
Ações do Ator	Ações do Sistema
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 3. Documentação do Caso de Uso Informar Selagem

Nome do caso de Uso	Informar Selagem
Caso de Uso Geral	Emitir GRU
Ator Principal	Posto
Atores Secundários	-
Resumo	Este caso de uso descreve as etapas percorridas por um usuário para informar selagem.
Pré-condições	Usuário autenticado; Gru Paga
Pós-condições	Informar dados da selagem
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Informar a GRU 2. Informar o veículo	3. Informar os dados da selagem
Restrições/Validações	1. Usuário autenticado deve ser um posto de Selagem ou PAC 2. Validar veículo 3. Validar GRU

4.2. Mapa de Contexto

A figura 5 representa uma visão geral do domínio GRU e em quais outros subdomínios o domínio GRU também é utilizado.

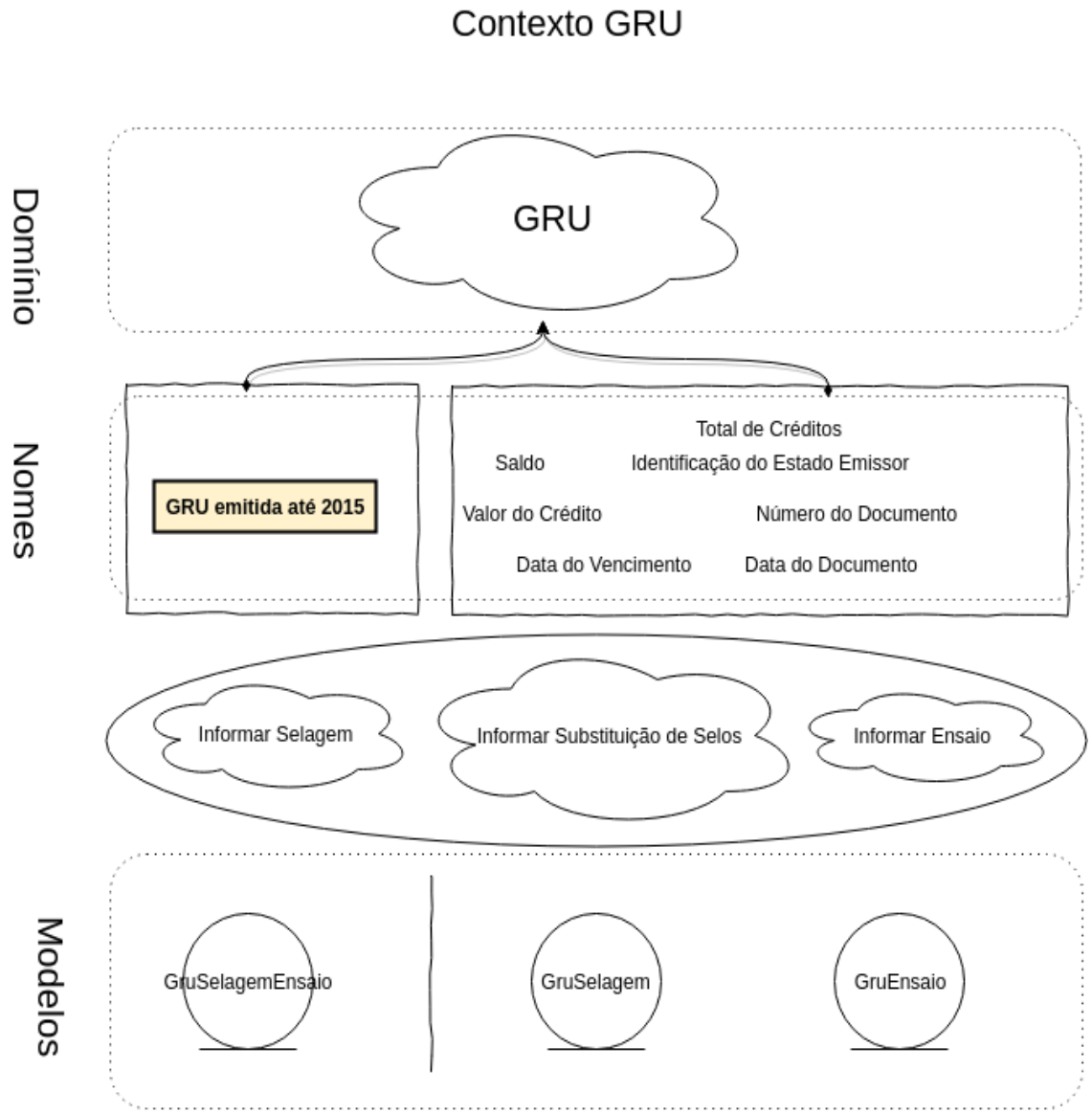

Figure 5. Padrões para auxiliar a modelagem de objetos

4.3. Arquitetura da Aplicação

A figura 6 representa através do diagrama de pacotes a estrutura hierarquica dentro da aplicação segundo os conceitos do *Domain-Driven design*.

Todas as camandas tem como dependência a camada *Domain*, com foco no domínio do negócio como é proposto pelo *Domain-Driven design*, as outras camandas não conseguem agir sem o domínio do negócio. A interface precisa acessar coisas do domínio e se comunicar com a aplicação e a infraestrutura para ser montada. A Aplicação precisa da infraestrutura e da intereface. A aplicação tem por responsabilidade capturar as requsiçãoes e conectar a interface com o restante da aplicação.

Mais externo ao contexto do sistema tem-se o *Database* que representa a fonte de dados do sistema. Também torna-se evidente a relevância maior do domínio e a necessidade de tirar o banco de dados do centro da aplicação.

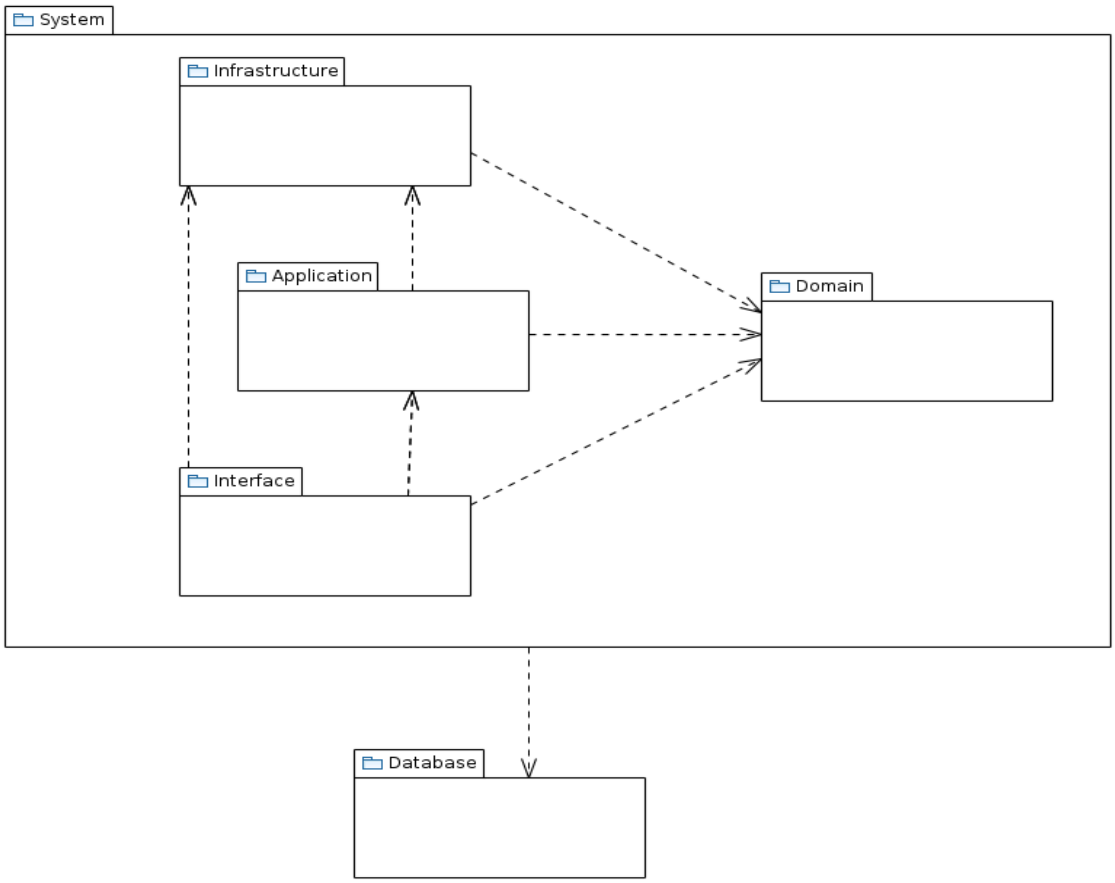

Figure 6. Representação resumida da arquitetura através do diagrama de pacotes

A figura 7 representa através do diagrama de pacotes a mesma estrutra apresentada na figura 6 mas de forma detalhada. É possível entender como está organizada a aplicação e a responsabilidade de cada camada. Nesse nível de detalhamento é visível notar a redução do acoplamento, aonde na camada de dominio concentra-se todo o conceito de *Domain-Driven design*.

Destaca-se a camada *Domain* que comportará repositórios, fábricas, serviços,

agregados e entidades. Na infraestrutura tem-se acesso a dados, configuração de serviço da aplicação, e de bibliotecas de terceiros.

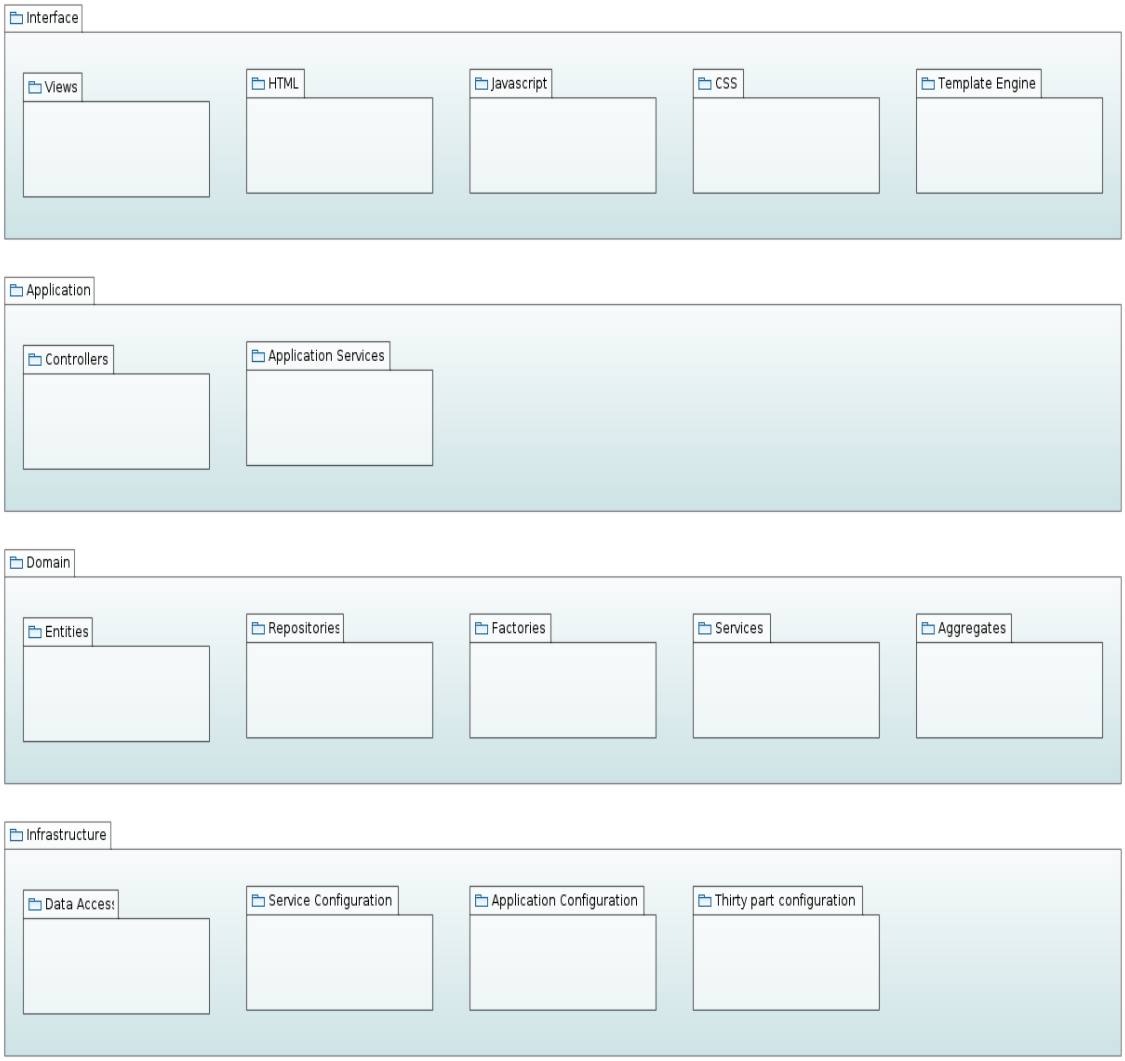

Figure 7. Representação detalhada de cada camada

4.4. Repositórios

A figura 8 representa o diagrama de classes relacionando classes que tem como responsabilidades acessar dados externos a aplicação. E isso permite maior flexibilidades quando se precisa trabalhar com mais de uma fonte de dados diferente, como criar repositórios específicos para fonte de dados específicos. Poderia a aplicação possuir um repositório específico para OracleRepository, MongoDBRepository, SqlServerRepository e nelas conter implementações específicas de acesso a dados.

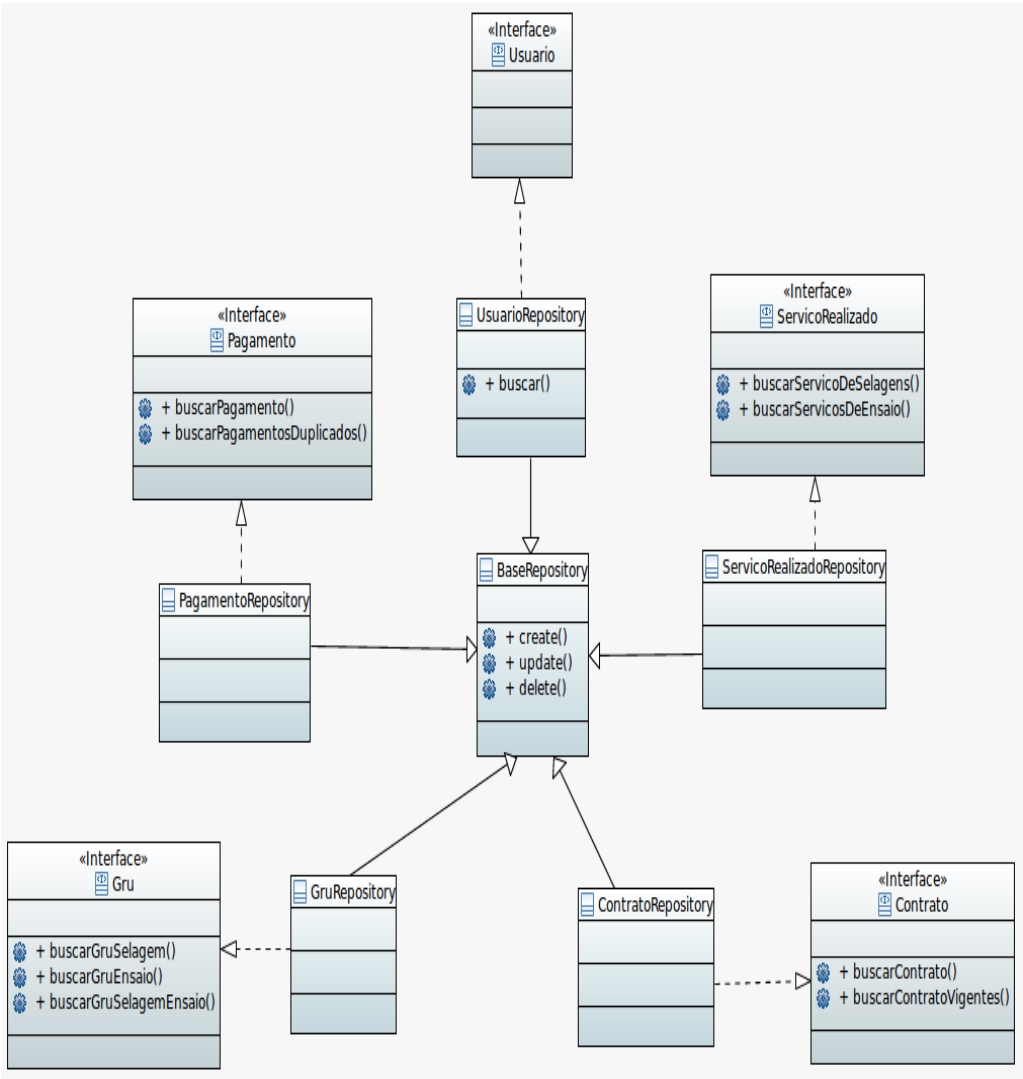

Figure 8. Representação de repositórios no diagrama de classes

4.5. Módulos

A figura 9 apresenta a organização do sistema na camada do domínio, foi organizado dentro dela além de entidades, quem tem maior afinidade com o contexto, também repositórios, interfaces, objetos de valor e fábricas.

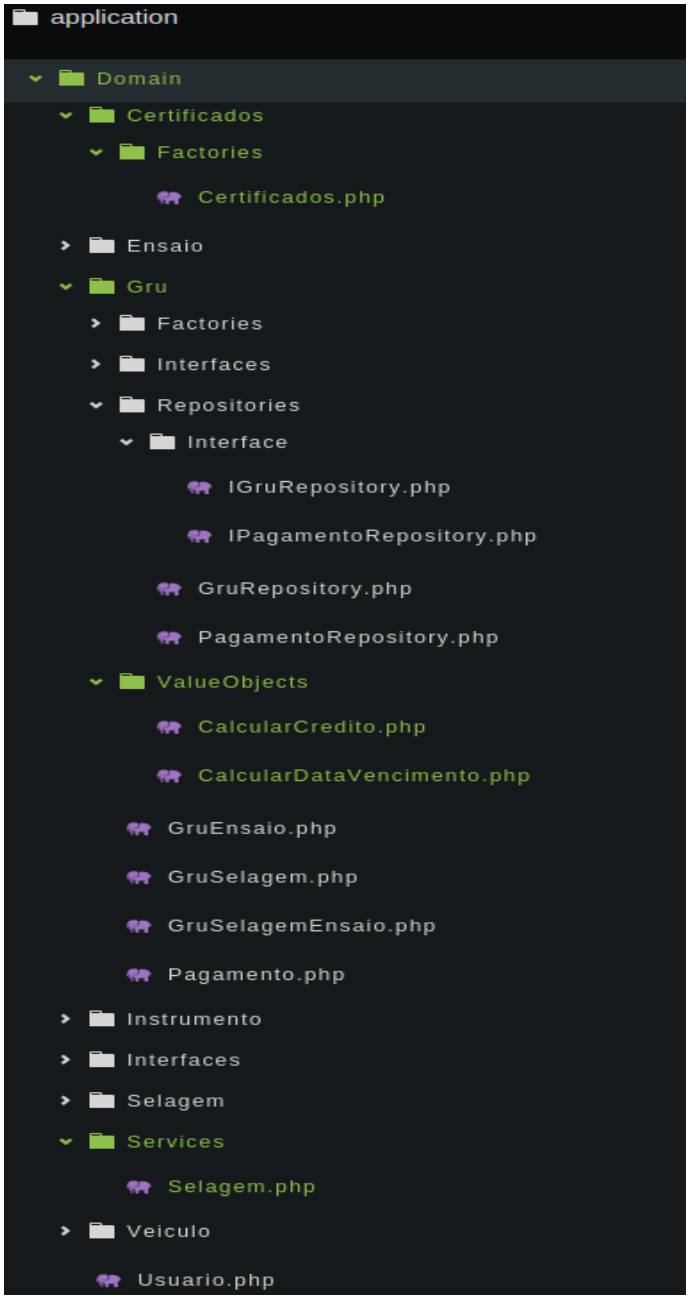

Figure 9. Representação da camada *Domain* baseada em módulos

5. Limitações

Segundo [Aniche 2015] não é o *framework* que vai fazer o software parar, nem ele que será mantido no futuro, são as classes que mantém as regras de negócio que precisam de atenção.

Essa afirmação se reflete por inteiro no sistema do Cronotacógrafo, não somente pela utilização do *framework*, não somente pelas implementações do código que apesar de possuir classes e métodos, é visível que a codificação fica longe do paradigma de orientação à objetos, existe também uma má tradução das regras de negócio quando transformadas em código. Mas o sistema consegue atingir o objetivo, que é entregar valor ao cliente segundo as regras de negócio definidas. O problema realmente é quando começam a exisitr exceções a regra e as classes já estão tão acopladas que é difícil identificar e separar responsabilidades, ainda mais quando o desenvolvedor que dará manutenção não é o mesmo que criou a funcionalidade, e assim acontece no Cronotacógrafo, os desenvolvedores não são responsáveis apenas por um módulo, assim como o módulo poderá ter vários responsáveis diferentes.

O amadurecimento da equipe também é muito importante para tentar aplicar *Domain-Driven design*, e concerteza o não entendimento dos conceitos descritos por [Evans 2003] causará confusão nos membros da equipe, e no caso deste artigo, tão difícil quanto compreender os conceitos de *Domain-Driven design* é tomar decisões com relação a quais contextos pertencem os objetos de domínio. Pois como em qualquer sistema as regras de negócio podem variar bastante, e talvez os objetos que antes faziam sentido dentro de um contexto podem não se encaixar no mesmo contexto a medida que as regras de negócio evoluem.

Atualmente o sistema do Cronotacógrafo utiliza o banco de dados da Oracle, e todas as classes tem uma implementação direta referenciando o banco de dados Oracle, não existe nenhuma classe intermediaria que permita abstrair as chamadas ao banco. Dessa maneira surgiu a necessidade da utilização de repositórios para chegar nos objetos de conhecimento do domínio. Passando então a fazer parte da linguagem ubíqua, uma vez que não deveria ser utilizado somente como algo interno ao código, estando presente também nas conversas com os especialistas do domínio e por consequencia no modelo do domínio. Mantendo-se o princípio da linguagem ubíqua não existe problema ao trazer termos técnicos se todos entendem o conceito, e se no contexto do domínio GRU existisse outro nome que fizesse mais sentido do que o nome repositório esse novo nome seria utilizado, mesmo os desenvolvedores sabendo que no fundo aquilo é um repositório.

O problema das pessoas é que utilizando *Domain-Driven design* acreditam que nele concentra-se toda a essência das coisas. O *Domain-Driven design* não é uma descrição de padrões de projeto, embora tenha alguns, ele auxilia o time de desenvolvedores a ficar atento em alguns pontos-chave para conseguir um melhor *design*, mas não ensinará um *design* melhor para o problema específico de gerenciamento de GRU, por exemplo.

Abri-se mão de classes coesas e flexíveis para ganhar os benefícios do *framework* e classes difíceis de serem testadas e mantidas são escritas pela simples questão da produtividade [Aniche 2015].

6. Conclusão

Este artigo demonstrou através de um problema real a aplicação de alguns conceitos de *Domain-Driven design* na solução do problema de gerenciamento de GRU de Cronotacógrafo, e a partir dele é possível considerar vários fatores que podem levar ao sucesso ou ao fracasso do projeto mesmo seguindo todas as etapas do *Domain-Driven design*.

Tão ou mais importante do que aplicar *Domain-Driven design* é necessário que todos os desenvolvedores da equipe tenham o domínio de orientação a objetos juntamente com SOLID e GRASP por exemplo, e que são importantíssimos para ajudar a pensar no *design* do software, em como as classes serão relacionadas e estruturadas. Conhecer ferramentas de testes unitários, ser capaz de trabalhar com desenvolvimento guiado por testes, entender sobre a importância da integração contínua, automatização de rotinas, boas práticas de desenvolvimento de software e tudo mais que puder ser relevante para ajudar a construir software confiável e manutenível de forma rápida e com ritmo e qualidade sustentável.

Quando se fala em orientação a objetos pensa-se logo em classes, heranças, polimorfismo, encapsulamento. Mas a essência da orientação a objetos também tem coisas como: alinhamento do código com o negócio, favorecer a reutilização e baixo acoplamento. *Domain-Driven design* poderia ser visto como uma evolução da orientação a objetos, mas ele não foca em tecnologia, mas sim em entender as regras de negócio e como elas devem estar refletidas no código e no modelo de domínio.

O *Domain-Driven design* permite realmente alinhar o conhecimento dos desenvolvedores e dos especialistas do domínio produzindo software mais coerente com o negócio e reduzindo as chances de que o conhecimento sobre o modelo do domínio fique nas mãos de poucas ou de uma única pessoa.

O grande desafio ao propor o desenvolvimento de software utilizando *Domain-Driven design* é manter a equipe comprometida desde o início com os novos conceitos, fazendo-os entender dos benefícios, pois cada um teve experiências anteriores com outros projetos complexos que não se utilizou da aplicação de conceitos mais evoluídos sugeridos pelo *Domain-Driven design* e mesmo assim o software entregou valor ao cliente. A equipe também terá dificuldades senão conhecer muito bem o *framework*, no caso deste artigo o *CAKEPHP*, que pela versão utilizada já pode ser considerado ultrapassado e que tornaria mais trabalhoso e custoso para o cliente aplicar *Domain-Driven design*.

Independente do framework e aplicando-se muito bem os conceitos de orientação a objetos é possível chegar bem próximo de alguns conceitos apresentados pelo *Domain-Driven design*. Pode-se afirmar que *Domain-Driven design* é programar utilizando orientação a objetos com responsabilidade e qualidade.

7. Referencias

References

Aniche, M. (2015). *Orientacão a Objetos e SOLID para Ninjas*. Casa do Codigo.

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

CakePHP (2008). http://book.cakephp.org/1.3/pt/the-manual/beginning-with-cakephp/understanding-model-view-controller.html. *Acesso em 21/05/2016*.

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.

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

Hess, H. (2016). *Evaluating Domain-Driven Design for Refactoring Existing Information Systems*. PhD thesis, Ulm University.

Hoffmann, K. B. (2009). Domain-driven design in action: Design an identity provider. Master's thesis, University of Copenhagen.

Inmetro (2011). http://cronotacografo.rbmlq.gov.br/o-que-e-cronotacografo. *Acesso em 20/05/2016*.

Landre, E., Wesenberg, H., and Rønneberg, H. (2006). Architectural improvement by use of strategic level domain-driven design. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 809–814. ACM.

Macedo, O. A. C. (2009). *Diretrizes para desenvolvimento de linhas de produtos de software com base em Domain-Driven Design e métodos ágeis*. PhD thesis, Universidade de São Paulo.

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. (2015). Implications of domain-driven design in complex software value estimation and maintenance using dsl platform.