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.

Este artigo está organizado da seguinte forma. A seção 2 descreve o que é a Linguagem Ubíqua e sua importância. A seção 3 introduz o conceito de DomainDriven Design. A seção 4 descreve como isolar o domínio das demais partes do sistema. A seção 5 fala sobre representação do modelo no software e a diferença entre entidade e objeto de valor. A seção 6 trata de padrões para se manter o ciclo de vida de um objeto de domínio. A seção 7 demonstra algumas técnicas descritas pelo DDD em um caso de uso.

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. Arquitetura

2.3. 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.

2.4. 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. 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]

2.6. Objetos de Valor

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

Segundo [Evans 2003] entender o pattern Value Object não é tão simples, já que na aplicabilidade exige abstração de OO para identificar quando precisamos de um. Um ponto de partida é ter sempre em mente que objetos 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 objeto 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]

Enquanto um objeto de valor é imutável, gestão da mudança é simples: não há qualquer alteração para além da substituição completa. Objetos imutáveis pode ser livremente compartilhada, como no exemplo tomada eléctrica. 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. 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, para que os serviços de domínio são operações apátridas. [Carlos Buenosvinos and Akbary 2014]

2.8. 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. (Eric Evans, 2003) Em um modelo dirigido por domínio, utilizamos 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). 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.

2.9. 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, Eric 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. 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. Fábricas podem criar Value Objects, que produzem objetos prontos, no formato final, ou Entity factories apenas para os atributos essenciais.

2.10. 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. Por outro lado, eles podem 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, Eric 2003) Muitas regras no domain model acabam sendo colocadas nas query?s, consequentemente se perdendo da aplicação, os repositórios tenta 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. Pensando em DDD, 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.

Repositórios representam coleções, enquanto DAOs estão mais perto do banco de dados, muitas vezes sendo muito mais table-centric. Normalmente, um DAO conteria métodos CRUD para um objeto de domínio particular.(Buenosvinos, Carlos. Soronellas, Christian and Akbary, Keyvan, 2016) 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.

2.11. 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. (Evan, Eric 2003) Uma preocupação comum ao criar um aplicativo seguinte DDD, é onde é que vamos colocar o código? O que é a maneira recomendada para colocar o código no aplicativo? Onde é que vamos colocar código de infra-estrutura? E mais importante, como devem os diferentes conceitos dentro do modelo ser estruturada? Há um padrão tático para isso: módulos. Hoje em dia, as estruturas de todos o código em módulos. Mas DDD vai queridos passo adiante e há preocupações técnicas são consideradas quando se usa módulos. De fato, trata módulos como uma parte do modelo. Dar aos módulos nomes que passam a fazer parte da linguagem ubíqua. módulos e seus nomes devem refletir uma visão sobre o domínio. (Evan, Eric 2003) Módulos não separam o código mas separam o significado conceitual. Buenosvinos, Carlos. Soronellas, Christian and Akbary, Keyvan (2016)

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`

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`

3.3. Using Domain-Driven Design to Evaluate Commercial Off-The-Shelf Software

`http://dddcommunity.org/wp-content/uploads/files/practitioner_reports/landre_einar_2006_part2.pdf`

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

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 e/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 méto- dos ágeis, surgiram vários problemas práticos de projeto. Sempre que possível, as soluções encon- tradas para cada um desses problemas foram generalizadas com o objetivo de serem reutilizadas em projetos posteriores

4. Solução Implementada

4.1. Subsections

The subsection titles must be in boldface, 12pt, flush left.

5. Considerações Finais

6. Figures and Captions

Figure and table captions should be centered if less than one line (Figure 1), otherwise justified and indented by 0.8cm on both margins, as shown in Figure 2. The caption font must be Helvetica, 10 point, boldface, with 6 points of space before and after each caption.

Figure 1. A typical figure

In tables, try to avoid the use of colored or shaded backgrounds, and avoid thick, doubled, or unnecessary framing lines. When reporting empirical data, do not use more decimal digits than warranted by their precision and reproducibility. Table caption must be placed before the table (see Table 1) and the font used must also be Helvetica, 10 point, boldface, with 6 points of space before and after each caption.

7. Referencias

Bibliographic references must be unambiguous and uniform. We recommend giving the author names references in brackets, e.g. [Knuth 1984], [Boulic and Renault 1991], and [Smith and Jones 1999].

The references must be listed using 12 point font size, with 6 points of space before each reference. The first line of each reference should not be indented, while the subsequent should be indented by 0.5 cm.

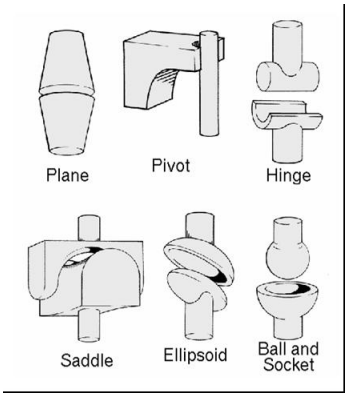

Figure 2. This figure is an example of a figure caption taking more than one line and justified considering margins mentioned in Section 6.

Table 1. Variables to be considered on the evaluation of interaction techniques

	Chessboard top view	Chessboard perspective view
Selection with side movements	6.02 ± 5.22	7.01±6.84
Selection with in-depth movements	6.29±4.99	12.22±11.33
Manipulation with side movements	4.66± 4.94	3.47±2.20
Manipulation with in-depth movements	5.71 ±4.55	5.37 ±3.28

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.

Boulic, R. and Renault, O. (1991). 3d hierarchies for animation. In Magnenat-Thalmann, N. and Thalmann, D., editors, *New Trends in Animation and Visualization*. John Wiley & Sons ltd.

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.

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.

Knuth, D. E. (1984). *The TEX Book*. Addison-Wesley, 15th edition.

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.

Smith, A. and Jones, B. (1999). On the complexity of computing. In Smith-Jones, A. B., editor, *Advances in Computer Science*, pages 555–566. Publishing Press.

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.