ENGENHARIA
DE SOFTWARE APLICADA

Por: Rogério Magela

ATHENAS
ENGENHARIA DE SOFTWARE
www.athenassoftware.com.br

Índice

Introdução

Nosso objetivo é fundamentar o leitor, no menor tempo possível, nos principais conceitos de Engenharia de Software. Decididamente e conscientemente abrimos mãos dos rigorosos conceitos que envolvem uma exposição acadêmica, e nos submetemos a necessidade prática diária.

O beneficio de assim proceder reflete nossa capacidade de aplicar o possível e colher resultados concretos, em detrimento a teorizar o impossível e colher apenas resultados imaginários.

Abordaremos conceitos avançados na produção de um software, com o objetivo de direcionar o leitor e fazê-lo entender onde está, qual a direção que propomos e o caminho a seguir. Esperamos dar solidez ao leitor em todas estas questões.

No entanto, e o mais importante, tentar mostrar ao leitor como transformar conceitos puros em algo que opera na realidade de nosso dia a dia. Ou seja, como transformamos conceitos teóricos e como eles afetam os *ifs then elses* de nosso dia a dia, saindo da dimensão do discurso para entrar na dimensão do real.

Engenharia de Software

A Engenharia de Software tem como objetivo "garantir" a construção de um software. Da mesma forma que a Engenharia Civil visa "garantir" a construção de um edifício. Para isso, a Engenharia Civil se utiliza de um arsenal matemático que garante uma série de resultados. Nós, de software, não dispomos deste arsenal matemático.

Conseqüentemente, convencionamos definir metas ou objetivos que almejamos atingir, com a expectativa de termos uma espécie de teorema. Se estas metas ou objetivos forem atingidos, o resultado será que "garantirmos" o software em construção.

Veja que nossa garantia é técnica. Não de processo. CMMI nível 5 não nos da qualquer garantia técnica. Em nosso exemplo de Engenharia Civil, seria somente uma eximia empresa responsável pela contratação, alocação, e todo o administrativo e burocrático necessário à construção do edifício.

Estas metas e objetivos podem ser vistos como axiomas. Por ora, não podemos dizer se estes são todos os axiomas que precisamos, ou se precisaremos de outros, ou se algum destes é dedutível de outro. Enfim, não podemos tratá-los com o rigor matemático. E, sendo assim, para diferenciá-lo, iremos denominá-los Princípios.

Definimos a existência de 2 tipos de princípios:

Princípios Externos – São os quantificadores que determinarão a qualidade externa do produto final. Qualidade está intrisicamente ligada à aceitação do produto e assim, obviamente, ao seu sucesso ou fracasso.

São eles: Corretude, Confiabilidade, Robustez, Desempenho, Ergonomia, Verificabilidade, Facilidade de Manutenção, Reutilização, Portabilidade, Compreensibilidade, Interoperabilidade, Produtividade, Desvio de Planejamento, Visibilidade e Segurança.

Princípios Internos – São os quantificadores que determinarão a qualidade técnica do produto final. Fornecem um guia que deverá estar presente em todas as técnicas ou processos de produção de um software.

São eles: Rigorismo, Separação de Aspectos, Modularidade,

Abstração, Generalidade, Antecipação a Mudanças, Refinamento e Classificação.

Uma questão simples que se coloca é a seguinte: dizer ao programador ou analista de sistema para fazer um software robusto ou usando abstração muda o que? Isso é suficiente? O Analista ou Programador saberá exatamente o que fazer?

É justamente este o ponto. Porque acreditamos que não muda quase nada. Não podemos parar por aqui. Pois se o fizermos, será apenas um discurso. Nada mais. E dissemos acima, que sairíamos do discurso e entraríamos no real.

Portanto, temos que de alguma forma garantir que usaremos estes princípios em nossos softwares. Como fazer isso? É justamente neste contexto que surgem conceitos como Processo de Desenvolvimento, Modelagem, Pattern e Frameworks. Acreditamos que a correta aplicação destes conceitos é suficiente em nos "garantir" a construção técnica de um software.

Processo de Desenvolvimento

Um processo de desenvolvimento em Engenharia de Software seria um roteiro a ser seguido para termos uma espécie de fábrica de software. Podemos pensar comparativamente uma fabrica de automóveis, onde o maquinário propriamente dito estaria relacionado com a tecnologia necessária em garantir que nossos princípios sejam seguidos, e todo o resto estaria relacionado a gestão do produto e administração da fábrica.

RUP, XP e os métodos ágeis em geral possuem as duas vertentes, mas a questão fundamentalmente técnica, ou seja, garantir que nossos princípios sejam seguidos, não são facilmente suportados.

Nosso intuito aqui é essencialmente técnico. Estes processos podem ser qualificados como excelentes tendo em vista vários julgamentos e propósitos. Porém, eles são insuficientes em relação ao que foi definido precisamente neste artigo: o de garantir aplicação dos princípios apresentados anteriormente..

Vale ressaltar que algum processo de desenvolvimento é necessário para a construção de um software. E não estamos aqui nos opondo aos processos acima. Estamos apenas constatando que no estreito olhar de garantir o uso técnico destes princípios, estes processos não são o caminho.

O caminho correto seria passar pela Modelagem e uso de um Framework. Pois, estes conceitos estão diretamente relacionados com os reais produtos de nosso software, o modelo de análise e o código-fonte, que são em ultima instância o software.

Modelagem

Modelar um software representa a atividade ou capacidade de transformar os requisitos do usuário em objetos computacionais apropriados através de uma linguagem.

Não significa criar diagramas intuitivos que facilitem o entendimento humano da questão. Modelar não é diagramar. Diagramar implica em simplesmente mostrar de uma forma coerente uma questão ou problema de para facilitar o desencadeamento do pensamento humano.

Implica em operacionalizar a realidade. Ou seja, criar um modelo computacional com uma série de atributos computacionais que permitam a execução real do software em si.

Acreditamos que nossa linguagem de modelagem ainda não é capaz disso. Ou seja, após modelar, executar o programa criado, testá-lo e debugá-lo sem qualquer linguagem de programação envolvida. De fato, algumas ferramentas já trabalham dessa forma, mas com algumas limitações.

Com base em nossa exposição constata-se que modelar é algo imprescindível na construção de um software. Assim, estamos alinhados com o movimento denominado MDD(Model Driven Development). Ressalta-se contudo que modelar é necessário, mas não suficiente para se produzir um software.

Mas, o que nos interessa aqui é o seguinte, vamos criar um paralelo rápido para garantir entendimento:

Se vamos construir uma casa, o que vem a sua mente? Se vamos construir um carro, o que vem a sua mente?

Não percebemos, mas em uma casa, imaginamos rapidamente uma cozinha, quarto, banheiro, sala no mínimo. E em um carro, imaginamos pneus, motor, bancos, volante, etc.. no mínimo. E quando alguém lhe pede um novo software? O que vem a mente?

No software em si, absolutamente nada. Em sua arquitetura podemos inferir, rede, banco de dados, servidores, etc... Mas isso não tem nada a ver com o software em si. Em nosso exemplo seria o mesmo que se pensando em uma casa, imaginássemos o cimento, a areia, a água, o tijolo, etc... Ou seja, um nível semântico a mais nas outras engenharias.

Como poderemos então subir um pouco mais nosso nível semântico quando falamos de um software? Esta questão nos leva ao conceito de Pattern.

Pattern

Inicialmente o conceito de Pattern ganhou um vigor maior e tornou-se forte devido ao livro do Design Pattern do Gamma e companhia. Este conceito vigora até hoje como sinônimo de Pattern.

No entanto este conceito tem evoluído, e hoje não nos limitamos mais ao Design. Genericamente podemos pensar em um Pattern como uma construção que tende a se repetir em uma área de conhecimento. Aplicado à Engenharia, podemos estipular que nossos pensamentos em relação ao que é um carro ou uma casa seria um Pattern.

As outras Engenharias trabalham com conceito de arquitetura. Que de certa forma, tem uma ligação com o conceito de Pattern. No entanto, Pattern esta limitado a um conceito menor, onde o todo, é de menor importância, e a parte o principal foco.

Não seria interessante quando pensarmos em um software, pensarmos em algo? Como quando pensamos em uma casa, imaginamos logo cozinha, banheiro, sala, quartos, etc...

Acreditamos que nossa engenharia de software está dando seus passos nesta direção. O excelente livro de Eric Evans inaugura o denominado DDD(Domain-Driven Development). Estes domínios seriam uma espécie de separação de aspectos diferentes existentes em um software, de forma que devemos separá-los mentalmente, mesmo antes de sua construção. Como fazemos quando pensamos em uma casa.

Estes domínios ou aspectos podem ser enquadrados como Patterns de nossa Engenharia de Software, ao qual, todo software deverá tentar se orientar. Evitar por exemplo, de colocar a cama ao lado do vaso sanitário. Algo impensável na construção de uma casa, mas que mantendo as devidas proporções, acontece rotineiramente quando se produz um software.

Estes Pattern isoladamente apenas delimitam e separam aspectos em um software. No entanto, a forma que estes Pattern se juntam para produzir efetivamente uma construção podemos denominar arquitetura.

Em Engenharia Civil o ato de olhar uma construção barroca, permite que uma nova construção siga esta arquitetura. Portanto, os olhos fazem o papel de meio pelo qual esta arquitetura pode ser repetida.

E em software? Como conseguimos garantir que uma determinada forma de unificar ou dar vida a estes Pattern pode ser repetida com garantias pelos programadores? Neste ponto, acrescentamos uma questão com o mesmo peso ou ainda maior, como garantimos que os princípios de Engenharia de Software serão efetivamente seguidos?

Estas questões nos remetem inevitavelmente para o conceito de Framework.

Framework

O Framework representa a forma com que estes Pattern, uma outra série de decisões de projeto irão guiar a construção de um novo software.

Dando continuidade a nossa exposição o framework permite que a arquitetura alvo seja seguida. Fazendo valor não só a arquitetura de acordo com os Pattern em uso, mas também garantindo que os princípios, ou ao menos alguns deles, sejam atendidos de antemão. Assim tornar inconsciente para o programador, que ele na verdade está seguindo os princípios ou o fazendo valer.

Dentro do exposto defendemos que o uso de um framework representa o único meio que dispomos para garantir a construção de um software.

Através e pela extensão de um framework podemos gradativamente dar vida a um software. No entanto, um framework é um guia de construção que fundamenta a construção de um software, mas em nossa área precisamos que ele se torne um programa, um executável, algo que possa ser interpretado por uma máquina ou hardware. Como então podemos executar um framework?

Este questionamento nos leva ao conceito de CEE ou Servidores de Aplicação.

CEE (Component Execution Enviroment) / Servidores de Aplicação)

Um CEE representa um ambiente de execução de componentes. Neste momento, nos falta trazer a cena o conceito de orientação a objetos e CBD(Component Based Development) inerente à toda tecnologia aqui apresentada. Entendemos que um framework se utiliza destes conceitos e eles são fundamentais para sua existência.

Se pensarmos que um framework na verdade está implementado via objetos e componentes, um CEE representa o ambiente que permita que ele execute.

O conceito de CEE não é muito utilizando, mas seu correlato, Servidor de Aplicação é bem conhecido. Um servidor de aplicação representa na verdade uma instância concreta de um CEE.

Oportuno e conveniente dizer que um framrework não necessariamente está relacionado a servidores de aplicação. A relação feita aqui representa nossa visão conceitual de como um software deve ser distribuído e executado em um hardware. Nossa visão deriva de uma abordagem CBD e condiciona, na verdade, a Arquitetura de Sistema que usamos em nossos softwares.

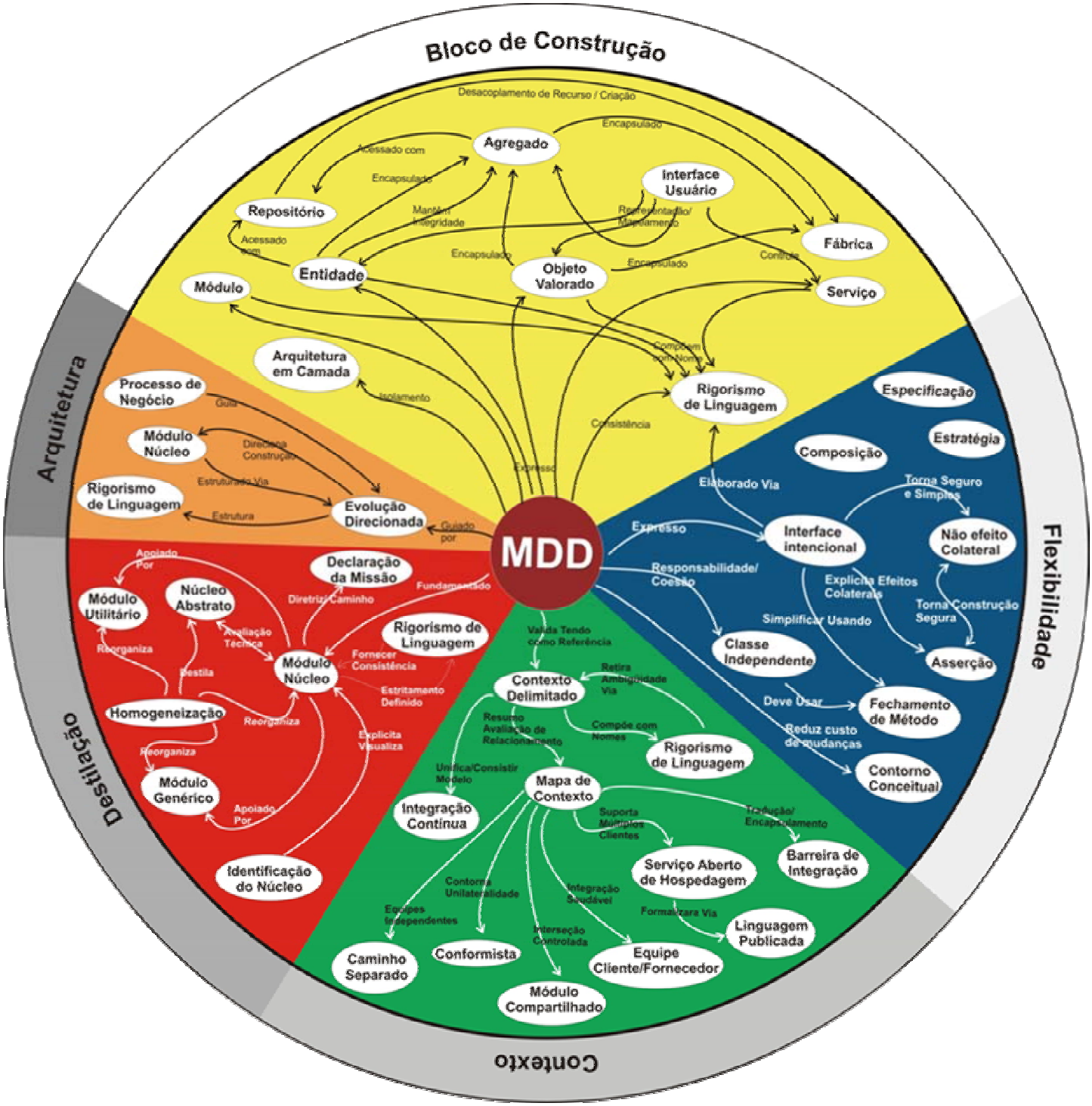
Arquitetura de Sistemas/Componentização

Nosso conceito de Componente não é outro senão aquele endossado pela abordagem CBD. Não compartilhamos do conceito de componente oriundo da orientação a objetos. Na verdade, um componente pela abordagem CBD não precisa ser implementado via orientação a objetos. CBD não precisa de OO. Embora, seu uso seja maximizado por OO.

Nossa Arquitetura de Sistema esta condicionado assim a 4 camadas físicas de processamento.

- USC – User Component. Meio de processamento em contato direto com o usuário. Utilizamos preferencialmente um browser para esta camada, mas qualquer meio de comunicação entre usuário e sistema ficaria nesta camada.
- WKC – Workspace Component. Meio de processamento responsável pelo casamento de impedância ou intermediação entre as camadas USC e EPC.
- EPC – Enterprise Component. Meio de processamento responsável pela execução dos componentes de negócio. Verdadeiro CEE ou Servidor de Aplicação.
- RSC – Resource Component. Meio de processamento responsável pelo armazenamento dos objetos que compõe o software ou memória do software.

Pattern de Domínio da Engenharia de Software



BLOCODECONSTRUÇÃO - Building Blocks

Para modelarmos um software precisamos constantemente tomar decisões de projeto. Existem certos balisadores ou padrões que podem nos auxiliar nestas decisões, facilitando e guiando nosso raciocinio em direção a melhor solução

.MDD(Model Driven Development) visa apresentar a nível de domínio do negócio padrões de modelagem consistentes, que devem guiar e departamentalizar a criação de qualquer software

.Os padrões presente em BLOCODECONSTRUÇÃO representam a estruturar nuclear de sustentação para todo modelo construído, e representam por si só um meio robusto de comunicação entre as equipes, permitindo uma organização fundamental no modelo, e uma comunicação consistente entre as equipes

RIGORISMODELINGUAGEM - Ubiquitous Language

- Linguagem do modelo deverá ser a mesma do domínio
- Use o modelo como único meio de comunicação com os especialistas do domínio
- Compartilhe o modelo com os especialistas do domínio
- A linguagem do modelo deverá ser expressa diretamente no código-fonte
- Mudança conceituais na linguagem implica em mudança no modelo
- A linguagem do modelo deverá ser fechada. Ou seja, definido em relação aos próprios elementos da linguagem, ou em elementos universais

MDD - MODEL-DRIVEN DEVELOPMENT

- Fazer o máximo para manter a frase verdadeira: Modelo é o código e o código é o modelo
- Toda e qualquer alteração no código deverá ser antes realizada no modelo, e toda alteração no modelo deverá ser refletida no código
- Modelo deverá nascer orientado a Design. Deverá existir um e somente um modelo. Portanto, modelos de analise de negócio ou modelo de análise de sistema pode existir enquanto caminhar em direção ao Design, mas não como um modelo separado que será mapeado para Design. Não existe mapeamento entre os modelos, gradativamente o modelo de negócio se torna o modelo de análise e depois Design e este último deverá ser mantido ao logo de toda vida do software
- A equipe de desenvolvimento poderá ter contato com os especialistas de domínio. Um bom modelo permite tal contato, evitando a monopolização através de analistas de negócio ou sistema

- O modelo de domínio deverá ser o modelo de Design. Se existe diferenças consideráveis entre estes modelos, o software produzido é suspeito
- Um mapeamento difícil ou complexo entre o modelo de Design e o código implica em problemas no modelo

ARQUITETURAEMCAMADA - Layred Architecuture

- Utilize 4 camadas na produção de um software:
- User Interface(USC), Application(WKC), Domain(EPC) e Infrastructure(RSC)
- Não espalhe regras de domino pelas camadas. Elas devem ficar em sua quase totalidade na camada EPC. Podendo, para facilitar performance e iteração ser colocada na camada USC, mas replicada na EPC mesmo assim, e em condições limites de performance serem colocadas na RSC
- Matenha os objetos de domínio livre de qualquer questão técnica inerente as camadas. Por exemplo, detalhes de interface do usuário ou persistência de dados. Cada camada deverá ser responsável pelo aspecto que a representa e os objetos de domínio não podem ser influenciados pelas outras camadas
- Torne cada camada coesa mais com baixo acoplamento entre elas. De forma a uma camada poder evoluir sem afetar a outra camada

ENTIDADE – Entities

- São objetos definidos por seu estado, continuidade e identidade, e não somente pelos valores de seus atributos
- Erro na interpretação da identidade de um objeto poderá levar a dados corrompidos e erros no software
- São objetos que possuem ciclos de vida, nascendo, vivendo e possivelmente morrendo
- São objetos onde a questão da identidade e persistência no tempo são fundamentais

OBJETOVALORADO - Value Objects

- São objetos que não possuem identidade conceitual. Eles descrevem alguma característica de alguma coisa, ou descrevem coisas propriamente ditos
- São objetos que representam aspectos descritivos do domínio e não possuem identidade conceitual.
- Representa o que são e não quem são. Cor e 7 são exemplos. Interessa qual cor e não quem é a cor, interessa qual numero e não quem é o numero 7

- Estes objetos de alguma forma devem ser diferenciados dos ENTIDADE, isso porque a maioria dos framework igualam estes objetos fazendo com que se perca a semântica de não identidade destes objetos e sua dependência de um ENTIDADE
- Podem ser persistentes ou transientes. Sendo transiente, pode ser criado apenas para trafegar entre parâmetros de métodos(classes de transporte)
- OBJETOVALORADO são imutáveis. Não podem ser alterados. Ou criados ou excluídos, jamais alterados
- OBJETOVALORADO não possuem identidade.
- Objetos agregados são candidatos a OBJETOVALORADO, pois dependem do ciclo de vida do "pai" e possivelmente não precisariam ser tratados como ENTIDADE

SERVIÇO - SERVICES

- São objetos com responsabilidade definida que nascem de ações e atividades do domínio do negócio mas que não representam ENTIDADE nem OBJETOVALORADO
- São de natureza transiente e atuam em vários ENTIDADE mudando seus estados e acordo com uma ação necessária solicitada ao software
- Objetos devem ser definidos para agrupar responsabilidade.(classes de atividade) Não espalhe esta responsabilidade em vários objetos muito menos em ENTIDADE
- Os nomes de seus métodos deverão estar alinhados com RIGORISMODELINGUAGEM, ser não existir, pensar na possibilidade de acrescentar(definição). Seus parâmetros e retorno devem ser objetos do domínio
- SERVIÇO aqui estão relacionado a camada EPC. Serviços oferecidos por outras camadas e que não estejam relacionado ao domínio do negócio são denominadas classes de serviço/utilitária

MÓDULO – Module/Package/Component/Subsystem

- Módulos devem ser tratados como elementos ativos do modulo, e não simplesmente como mecanismo organizacional(componentes e subsistemas são ativos, pois representam a arquitetura e consequentemente afetam diretamente o código)
- Módulos devem ter baixo acoplamento e alta coesão(axiomas arquiteturais)
- Se o seu modelo fosse uma historia, os módulos seriam os capítulos, portanto, devem como já dito possuir semântica clara, e não apenas fatores organizacionais
- Os nomes dos módulos devem fazer parte da RIGORISMODELINGUAGEM
- Use "subsistema" como um módulo ativo de larga escala, use um "componente" para

agrupar elementos ativos de média escala dentro de um "subsistema" e use "pacote" para realizar divisões organizacionais(não ativa)

- As classes deverão estar dentro de um "componente" e se o framework permitir, dividida em pacotes separados por aspectos

AGREGADO - Aggreagate

- É um grupo de objetos relacionados que devem ser tratados como uma única unidade em se tratando de mudanças
- Cada AGREGADO tem um objeto raiz e uma fronteira definida. A fronteira define o que esta dentro do AGREGADO . O objeto raiz é um ENTIDADE e os objetos agregados são OBJETOVALORADO
- O objeto raiz(ENTIDADE) é o único objeto que poderá ter relacionado com outros objetos externos ao AGREGADO . Objetos internos ao AGREGADO só poderão ter relacionamentos entre si
- Excluir o objeto raiz implica em excluir todos os objetos do AGREGADO
- Visa manter a consistência de um grupo de objetos durante mudanças. No grupo de objetos em questão, uma pré-condição deve ser observada, uma semântica e invarianças. Em relação a todo grupo de objetos
- Um modelo deve evitar contenção e favorecer invarianças
- Todo acesso aos OBJETOVALORADO devem ser feito via o objeto ENTIDADE raiz

FÁBRICA - Factory

- Encapsula a criação de um objeto ENTIDADE, OBJETOVALORADO ou AGREGADO não deve ser de responsabilidade do próprio objeto. Algum mecanismo deve ser criado que permita que tais objetos seja criados
- FÁBRICA são objetos criados cuja responsabilidade é criar objetos ENTIDADE, OBJETOVALORADO ou AGREGADO
- Objetos CDB
- Se os objetos são criados somentes em memória as regras ou invarianças dos objetos devem ficar aqui. Caso haja acesso a qualquer meio(RDBMS) um mecanismo diferente para reconstituição de objetos se faz presente.

REPOSITÓRIO - Repositories

- Objetos ENTIDADE possuem ciclos de vida. Se estes objetos forem persistidos em alguma

mídia e posteriormente no tempo reconstruídos, devemos criar mecanismos de acesso e reconstrução destes objetos. Os REPOSITÓRIO

- Objetos AGREGADO devem ser acessados somente via raiz. Mesmo que se deseje acessar um OBJETOVALORADO. Estes objetos só podem ser acessados via navegação pelo ENTIDADE raiz
- Encapsule questões técnicas de acesso a mídia(RDBMS), ou seja, acessos a SQLs não devem ser visíveis, nem o mecanismo da linguagem de programação utilizado para acessa-lo, e também não se deve transferir qualquer responsabilidade deste mecanismo para o ENTIDADE ou OBJETOVALORADO
- Deve ser proibido acessar dados crus, ou seja, somente através de objetos. Jamais se deve acessar dados via SQL, por exemplo
- REPOSITÓRIO podem trabalhar conjuntamente com FÁBRICA, em se tratando de RDBMS, isso é aconselhável
- Deverá em sua assinatura possuir métodos para incluir, alterar e excluir objetos
- Deverá em sua assinatura possuir métodos para selecionar objetos de acordo com critérios pré-determinados, fornecendo um meio de instanciação de um objeto bem como meios para acessar uma coleção de objetos, de forma totalmente encapsulada, sem transparecer questões tecnológica de como este processo é realizado
- Objetos CGR
- Regras gerencias(invarianças) devem ser tratadas em objetos REPOSITÓRIO. Embora, algumas literaturas o vejam em FÁBRICA achamos incorreto. Em termos de RDBMS, o local correto seria aqui mesmo

INTERFACEUSUARIO

- É considerado ANATEMA utilizar datalive ou smartui no desenvolvimento de qualquer interface com o usuário. Exceção de relatórios que possuem tecnologias próprias e podem fazer uso de recursos SQL diretamente.
- Todas a telas construídas devem ser realizadas com elementos de interfaces nativos, que deverão ser mapeados para ENTIDADE, AGREGADO, OBJETOVALORADO ou WRAPPER e ser controlado via SERVIÇO
- Toda regra de negócio deverá ser realizada no na camada EPC(Enterprise Component). No entanto, regras simples que podem ser validadas no cliente evitando uma chamada ao servidor para aumentar o desempenho podem ser realizadas, desde que devidamente encapsuladas
- Estereotipamos as telas da seguinte forma: CAD – Cadastro, PRO – Processo, CSL –

Consulta, REL – Relatório , GRD – Grades, PES – Pesquisas, COB – Combos. As telas efetivas ainda podem ser extendidas, sendo chamadas EXT.

MODELOFLEXIVEL – Supple Design

- A meta primaria de um software é servir ao usuários, no entanto, antes de servir ao usuário um software deve servir aos desenvolvedores
- MODELOFLEXIVEL é o complemento necessário a todos os modelos. Através de iterações e de novos insumos, o material deverá ser transportado para o modelo, sem forçar nenhuma estrutura, de forma simples e clara. No entanto, onde efetivamente queremos chegar com estas palavras e "conselhos"? Que técnica ou forma de trabalho nos levará a um MODELOFLEXIVEL?
- O desenvolvedor atua em 2 papéis. Em primeiro lugar ele é cliente de um modelo. Portanto, deverá ser capaz de forma rápida, simples e clara navegar pelo modelo, e mentalmente orientar sua solução da forma a representar cenários de forma flexível com o modelo existente. E em segundo lugar, o modelo deve permitir com relativa facilidade a alteração e inserção de novos conceitos, também de forma rápida, simples e clara. Cumprindo estas metas, teremos um MODELOFLEXIVEL

INTERFACEINTENCIONAL – Intention-Revealing Interfaces

- Os nomes devem antes de tudo nos direcionar para intenções. Secundariamente significados e mais concretamente OQUE será feito. Nunca, jamais, COMO será feito ou estar relacionado ao mecanismo de COMO será feito
- Se dado uma classe e seus métodos um cliente(desenvolvedor) necessitar olhar o código para dizer OQUE ele faz, então o valor do encapsulamento e reuso foi perdido
- O nome de uma classe, método ou atributo deve estar orientado para OQUE se faz e não COMO se faz
- Os nomes devem estar de acordo com o RIGORISMODELINGUAGEM
- Escrever testes para novos métodos ou classes antes de implementá-lo, poderá ajudar a inferir se os nomes estão definidos corretamente
- In the public interfaces of the domain, state relationships and rules, but not how they are enforced; describe events and actions, but not how they are carried out; formulate the equation but not the numerical method to solve it. Pose the question, but don´t present the means by which the answer shall be found

NÃOEFEITOCOLATERAL - Side-Effect-Free

- Um efeito colateral representa qualquer mudança no estado do software que afetará operações futuras
- Operações podem ser divididas em 2 tipos. Operação Função e Operação Comando. A operação função retorna sempre o mesmo valor, como uma função matemática. Uma função comando, modifica o estado do software, afetando portanto uma próxima execução
- Tente separar e diferenciar métodos oriundos de operações função e comandos. Definindo métodos função e métodos comandos separadamente. Em conjunto com o uso de INTERFACEINTENCIONAL teremos uma nomenclatura mais sólida e ao mesmo tempo uma análise de impacto e risco
- OBJETOVALORADO ou XML são tipos de retornos possíveis e seguros, tendo em vista que são constantes e imutáveis e sem um processamento extra não podem mudar o estado do software(gerar efeito colateral)
- Em uma lógica mais apurada de um método comando, mesmo com uma pequena queda no desempenho, crie métodos função de apoio. Separando claramente acessos somente leitura de atualizações nos dados e mudança de estado presentes em métodos comando

ASSERÇÃO – Assertion

- Explicita o uso de métodos comando. Visa tornar consciente a construção, uso e conseqüências de métodos que alteram o estado do software, em sua maioria, alterando estado de ENTIDADE
- Lembramos que a implementação de uma mesma assinatura(em uma interface), poderá produzir efeitos colaterais diferentes, implicando que uma assinatura não é suficiente para nos dizer algo sobre efeitos colaterais
- Idealmente deveríamos ter uma forma de inferir o efeito colateral de métodos assertivos
- O uso das assertivas pré-condição, semântica e pós-condição deverão se fazer presente, sempre que necessário no código. A nível de modelagem através de expressões lógicas referindo-se somente a pré-condição e pós-condição. A semântica(COMO) não deverá estar presente em modelagem, e o uso de pós-condição é obrigatório. Mesmo que no código, a pós-condição seja omitida e incorporada a semântica
- Invariantes também devem ser definido inicialmente a nível de ENTIDADE, mas também a nível de AGREGADO e ainda a nível de REPOSITÓRIO ou FÁBRICA conforme o caso

- Testes deverão ser orientados a assertivas. Portanto, orientados a pré-condição e pós-condição

CONTORNOCONCEITUAL – Conceptual Contours

- Unificar em um grande método várias responsabilidades de forma monolítica, possui uma série de desvantagens, sendo um aumento considerável de complexidade e duplicação de código um dos mais relevantes. A assinatura do método mesmo seguindo INTERFACEINTENCIONAL não diz absolutamente nada sobre seu funcionamento interno. Somente analisando o código, saberemos realmente o que esta acontecendo
- Por outro lado, quebrar em pequenos pedaços classes e métodos podem eliminar por completo um conceito. Forçando uma navegação do programador em várias classes e métodos para entender seu funcionamento.
- Durante a manutenção de um software e consecutivas reconstruções perdemos o CONTORNOCONCEITUAL. Devemos sempre a todo custo tentar mantê-lo, mesmo sob a pressão sempre constante na manutenção e entrega de um software
- Pondere: de um lado mudanças e estabilidade de outro reconstruções(refactoring) sucessivas. A constante destas iterações deverá ser CONTORNOCONCEITUAL
- Seguir CONTORNOCONCEITUAL não implica em ser desnecessário realizar refactories no software. Sempre é necessário o refactoring, mesmo continuamente se atendo ao CONTORNOCONCEITUAL
- Quando uma nova manutenção é realizada e fica claro a necessidade de uma nova reorganização na classe ou nos seus métodos. FAÇA-A não deixe para amanha

CLASSEINDEPENDENTE – Standalone Class

- Sempre quando possível, devemos evitar relacionamentos entre classes. Pois, uma classe com vários relacionamentos, representa forças que são difíceis de manter coesa
- No entanto, sabemos que na maioria das vezes, o problema dita os relacionamentos. Sendo assim, deverá existir alguma técnica que permita um encapsulamento em MÓDULO do problema.
- Utilizamos CBD como sendo esta técnica. E dividimos os MÓDULO em Sub-sistemas e componentes e em aspectos. Esta abordagem permite um gerenciamento de complexidade e ajuda a criar CLASSEINDEPENDENTE
- Técnicas na construção de diagramas devem ser seguidas. O que deve e o que não se deve estar presente nos diagramas(vide AP)

- Se possível, tente criar mecanismos que isolem classes. Arquitetura de software favorece isso, pois cria regras rígidas de dependência entre sub-sistemas. Sub-sistema de Parametrização também deve ser incorporado, bem como de Abstração(MÓDULONÚCLEO)
- CLASSEINDEPENDENTE esta estritamente relacionamento com baixa acoplamento entre classes. Meta sempre buscada

CLOSURE OF OPERATIONS(Fechamento de Método)

- Se possível, devemos privilegiar métodos cujo retorno seja do mesmo tipo dos parâmetros
- Isso geralmente só é possível com tipos primitivos ou OBJETOVALORADO, mas eventualmente também se aplica a ENTIDADE
- Ser uma operação fechada em relação a um tipo, implica em receber o tipo como parâmetro e retornar o tipo, provavelmente outra instancia, como retorno
- Importante relacionar este conceito com RIGORISMODELINGUAGEM em relação aos nomes das operações. Visto que não poderemos criar nomes de classes ou métodos fora da RIGORISMODELINGUAGEM, ou seja, nossos nomes estão fechados em relação a RIGORISMODELINGUAGEM

ESPECIFICAÇÃO - Specification

- Regras de Negócio(Business Rules) geralmente estão distribuídas e não podem, sem distorções, serem incoporadas a ENTIDADE e OBJETOVALORADO. Variações destas regras e suas combinações em novas regras podem destruir a semântica e o significado do objeto. No entanto, não fornecendo meios de modelar estas regras, elas podem se espalhar no código, e o modelo não mais expressar a realidade
- O conceito de predicados é utilizado em lógica de programação para separar e combinar regras de objetos, com as devidas reservas e adaptações.
- ESPECIFICAÇÃO são objetos geralmente do tipo OBJETOVALORADO onde seus métodos são predicados que avaliam alguma regra de um ou mais objetos, retornando uma coleção dos objetos que satisfazem aquela regra ou um boolean true ou false se satisfaz ou não
- Podemos ainda pensar em 3 tipos diferentes de ESPECIFICAÇÃO
- Validação: Class ContainerSpecification { public boolean isReefer (Container prCandidatocontainer) { .....} }. Deve atuar preferencialmente em 1 objeto somente, quando muito, atuando em uma coleção em memória. Sua ligação com REPOSITÓRIO deve ser feita externamente e não via dependência. Pois, em teoria um objeto

ESPECIFICAÇÃO poderá atuar na camada USC, podendo assim receber objetos de outras fontes diferentes de REPOSITÓRIO e não podem criar vinculo direto com RDBMS

- Seleção: Class ContainerSpecification { public Set setReefer(…) { …} }. Atua em vários objetos. Ainda podendo ser utilizado na camada USC, portanto, ainda independente do RDBMS
- Processo: public interface ContainerEmOperacaoSpecification { public void Unitizar(…)}. Além de atuar em vários tipos de objetos, não só objetos do tipo da especificação, representam os verdadeiros processos da camada EPC. Devem ser gerados via assinaturas e interface e não implementados via class. Deverá ser implementado via SERVIÇO
- Podemos ainda dentro da classe ESPECIFICAÇÃO implementar os métodos AND, OR e NOT de forma a facilitar e tornar limpo o processo de composição de regras

ESTRATÉGIA - Strategy

- Quando existirem formas alternativas de executar um processo, a complexidade de escolher o processo a executar e a complexidade do processo em si, podem comprometer o programa construído(tornando ele complexo e de difícil manutenção)
- Estes processos alternativos devem ser modelados como um conceito, utilizando ESTRATÉGIA, refletindo o código-fonte
- Fatore a parte variante dos processos em separados objetos "ESTRATÉGIA"
- Implemente utilizando o pattern STRATEGY do GAMMA. Múltiplas versões do objeto ESTRATÉGIA representam as maneiras que o processo pode ser realizado
- Aqui ESTRATÉGIA não é design e sim domain. Ou seja, não representa algoritimos diferentes e sim caminhos ou formas diferentes de executar um processo

COMPOSIÇÃO - Composite

- Quando existir um relacionamento parte todo, onde a parte e o todo devem ser tratados indistintamente e receber as mesmas operações, o pattern COMPOSITE deverá ser aplicado
- Aplicável somente para um hierarquia onde todas as partes representam o mesmo tipo conceitual
- Defina um tipo abstrato que representa todos os membros da composição. Métodos aplicados aos membros devem ser repassados para os filhos ou aplicados diretamente se for folha. As folhas implementam os métodos normalmente, neste caso, para realizar a

responsabilidade definida. Clientes usam o tipo abstrato, indiferente se estão se comunicando com que parte do COMPOSIÇÃO

CONTEXTO - Context

- Um modelo deve ser consistente, ou seja, seus elementos devem ter o mesmo significado em qualquer contexto do software, e não deve existir regras contraditórias para o mesmo elemento. Um modelo consistente representa um modelo unificado. Unificação é um axioma de qualquer modelo
- Assumimos que a total unificação de um grande modelo de um grande sistema não é viável nem leva a um custo benefício favorável. Problemas de coordenação de todos os tipos, problemas especializados que não podem ser satisfeitos levando o usuário a controles paralelos, ou por outro lado se tentarmos satisfazer a todos cairemos em um modelo extremamente complexo, representam alguns determinantes para este posicionamento
- Sendo assim, é necessário uma forma de delimitar a fronteira do que será unificado e do que não será unificado, do que representa o contexto do modelo e do que não representa
- Os padrões do CONTEXTO visam definir técnicas para reconhecer, comunicar, e escolher os limites de um modelo e seu relacionamento com outros modelos

CONTEXTODELIMITADO - Bounded Context

- Em um grande projeto podemos ter múltiplos modelos. Sem as devidas providências combinar modelos diferentes a nível de código-fonte poderá levar a efeitos imprevistos e códigos de difícil compreensão. A comunicação entre equipes fica comprometida e confusa
- Temos que definir um contexto especifico de aplicabilidade do modelo, bem com suas fronteiras
- Todo modelo possui um CONTEXTO
- Defina explicitamente o contexto dentro do qual o modelo esta sendo definido. Defina explicitamente suas fronteiras em termos de processos de negócio e das equipes de trabalho, onde ele será usado especificamente dentro da organização, ambiente técnico como componentes e esquemas de banco de dados. Mantenha o modelo sempre consistente dentre destes limites estabelecidos, e não deixe que o ambiente externo corrompa seu modelo, através da aplicação de outros pattern de domínio
- Um CONTEXTODELIMITADO delimita a aplicabilidade de um modelo em particular, para que os membros de sua equipe tenham um entendimento claro e compartilhado do que

tem que ser consistente e como este modelo se relaciona com outros modelos com outros contextos

- Cada modelo representa na verdade um dialeto. E sua linguagem é definida pela RIGORISMODELINGUAGEM
- A integração com outros modelos e contextos deverá ser realizada via fronteiras e envolverá mapeamentos e traduções entres estes contextos, que deverá ser analizado explicitamente e separadamente

INTEGRAÇÃOCONTINUA – Continuos Integration

- Tendo definido um CONTEXTODELIMITADO temos que garantir que o manteremos consistente
- Quando um grande numero de pessoas trabalha em um mesmo CONTEXTODELIMITADO teremos uma grande tendência a fragmentação do modelo e com o tempo uma perda de consistência e no pior caso uma completa distorção de todo contexto e de todo modelo
- Novos desenvolvedores que desconhecem a profundidade do modelo, poderão ingenuamente fazer alterações que comprometam uma série de regras explicitas ou implícitas, tornado o modelo ilegível e possivelmente afetando aspectos como desempenho e uma enorme dificuldade de manutenção, devido ao acréscimo exponencial de complexidade desnecessária
- Partes do modelo e do código são duplicados ou por não entendimento que outra parte do modelo atende os mesmos requisitos, ou por receio ou medo de alterar um código que esteja já funcionamento eficientemente
- INTEGRAÇÃOCONTINUA implica que todas alterações/tarefas realizadas dentro de um contexto especifico será integrado(merged) consistentemente e aproveitar cada nova alteração/tarefa para rever se já não houve uma quebra prévia do modelo. Sendo assim, a INTEGRAÇÃOCONTINUA será feito continuamente e a cada alteração no modelo
- Integração em MDD opera sempre em 2 níveis: 1) Integração do modelo e 2) Integração do código. Modelo e código devem sempre estar alinhados. E nesta ordem. Alteramos sempre o modelo 1º e imeditamente alteramos o código
- INTEGRAÇÃOCONTINUA deverá sempre estar fundamentado na RIGORISMODELINGUAGEM

MAPADECONTEXTO – Context Map

- Dependendo do tamanho do projeto um único CONTEXTODELIMITADO não é suficiente para integrar uma visão global da empresa e seus processos de negócio

- Quando existe a necessidade de mapeamento entre contextos diferentes temos que verificar qual representa a melhor abordagem, visto que uma grande fonte de distorção entre contextos pode ser oriunda do mapeamento incorreto entre contextos
- Problemas políticos entre equipes, espaço físico diferente, cidades diferentes, coordenadores ou gerentes diferentes podem levar a quebra de um contexto em outros contextos e a integração entre contexto ser necessário mesmo dentro de um único CONTEXTODELIMITADO
- Antes do mapeamento propriamente dito concentre-se em definir precisamente o CONTEXTODELIMITADO e certifique-se de verificar também a RIGORISMODELINGUAGEM. Descreva então os pontos de contato entre os contextos, apresentando os pontos de mapeamento e os pontos de compartilhamento . Após garantido esta etapa, passe efetivamente para o mapeamento em si
- Decidindo qual tipo de mapeamento iremos usar, todos os membros da equipe devem ser comunicados e deve-se compartilhar o modelo e a forma de mapeamento entre todos
- Um MAPADECONTEXTO não representa o mapeamento em si. Representa o estado atual e todo cenário para o mapeamento. Representa como os CONTEXTODELIMITADO estão definidos e todo seu contorno. Visa limpar e organizar a casa para que o mapeamento efetivo possa ser realizado, e além disso, em um nível ainda alto de abstração mapear os elementos e verificar se existe precisamente uma correspondência semântica entre os 2 CONTEXTODELIMITADO. Ou seja, garantir que a pré-condição de mapeamento, sob o ponto de vista dos contextos sendo integrados, estão satisfeitos
- Mudanças nos modelos podem ser necessárias. Realize as mudanças antes e coloque-as em produção na pratica. Somente após esta mudança na realidade devemos retornar ao MAPADECONTEXTO e efetivamente iniciar o mapeamento

MÓDULOCOMPARTILHADO - Shared Kernel

- Equipes grandes demais trabalhando em um único CONTEXTODELIMITADO poderá comprometer a INTEGRAÇÃOCONTINUA, tornado-a muito difícil de ser realizada e com um tempo elevado
- Pode ser interessante, e até mesmo vital, separar o modelo em dois ou mais CONTEXTODELIMITADO e trabalhar de forma independente, a não ser, por um núcleo básico de comum acordo entre as equipes denominado MÓDULOCOMPARTILHADO
- As equipes poderão efetuar alterações em seu próprio CONTEXTODELIMITADO e realizar a INTEGRAÇÃOCONTINUA de forma natural, e continua. No entanto, quando uma das equipes necessitar alterar o MÓDULOCOMPARTILHADO, deverá ser solicitado uma

aprovação por parte das outras equipes que utilizam do mesmo MÓDULOCOMPARTILHADO

- MÓDULOCOMPARTILHADO implica não só modelo, mas também código-fonte e outros itens relacionados, como banco de dados. E espera-se que o MÓDULOCOMPARTILHADO seja atualizado com menos freqüência comparado ao INTEGRAÇÃOCONTINUA das equipes
- Atualizações no MÓDULOCOMPARTILHADO implicar em novos testes para as equipes que trabalham com ele. Alterações no MÓDULOCOMPARTILHADO devem ser feito isoladamente das outras equipes, e só depois de aprovadas, passarem para área de produção

EQUIPECLIENTEFORNECEDOR - Customer/Supplier Team

- Relacionamento estabelecido entre equipes de CONTEXTODELIMITADO diferentes, onde uma equipe cliente(EC) depende de uma equipe fornecedora(EF)
- Geralmente a EF possui seus próprios clientes e seus próprios prazos, estabelecendo um relacionamento difícil com a EC. Sendo assim, é fundamental que exista um contrato balizado por um coordenador ou gerente estabelecendo a EC como mais um cliente da EF. Para que a EF não comprometa os prazos da EC visando atender seu próprio cronograma e seus clientes "legítimos". E por outro lado, todos esquecerem dos prazos da EF, e a EC achar que seus prazos são os mais importantes
- Tente definir uma assinatura ou uma especificação que defina precisamente o que a EC espera da EF. Este contrato deverá delinear não somente a entrega dos produtos da EF a EC, como também balizar futuras alterações da EF, para que esta, não precise ficar remetendo continuamente suas alterações para validação da equipe EC
- Caberá também a equipe EC se comprometer a aguardar os prazos, e não solicitar mudanças na especificação contratual definida entre as equipes continuamente. Se possível, defina um programa de teste, criado pela EC para ser executado pela EF através de suas INTEGRAÇÃOCONTINUA

CONFORMISTA - Conformist

- Um EQUIPECLIENTEFORNECEDOR assumi uma cooperação entre equipes. Onde a equipe EF é fornecedora da equipe cliente EC. Quando a equipe EF por qualquer razão, altruística ou não, não coopera adequadamente, temos que procurar outros caminhos para integração
- Neste contexto, existem 3 caminhos a seguir. O primeiro e mais radical seria abandonar a equipe EF e incorporar em sua própria equipe seu CONTEXTODELIMITADO. Esta

opção deve ser analisada realisticamente, visto que embora possa ser uma atitude de herói e libertação, o custo de agir desta maneira pode se tornar alto demais. Se este realmente for o caminho estaremos no domínio de CAMINHOSEPARADO. O segundo caminho seria manter a integração mas criar uma camada robusta que permita diminuir ao máximo a dependência através de BARREIRADEINTEGRAÇÃO. Esta camada seria necessária para encapsular, alinhar a abstração, separar aspectos e realmente diminuir a dependência, visto que foi assumido que não teremos o devido suporte. Mas, por outro lado, reconhecemos o valor da EF

- O terceiro caminho seria assumir a dependência, talvez por tempo limitado, entre EC e EF. Este caminho seria CONFORMISTA. Embora possa parecer o caminho do anti-herói, pode ser o melhor a seguir tendo em vista custo e uma boa dose de gerencia. Podemos assumir o valor da equipe EF e tentar nos ajustar a este relacionamento. Se ele se mostrar vantajoso, e realmente simplificar nosso trabalho através de qualidade e outros aspectos.

BARREIRADEINTEGRAÇÃO – Anticorruption Layer

- Assumindo que será efetivamente necessário a integração entre 2 CONTEXTODELIMITADO mantido por equipes diferentes, mas que por motivos maiores, não será possível alinhar as equipes via EQUIPECLIENTEFORNECEDOR ou até mesmo CONFORMISTA o caminho a seguir é proteger a equipe cliente EC de mudanças ou de um modelo pobre da equipe EF. Equipe EF aqui pode representar um software legado
- Isolar em um modelo contratual via interfaces TODAS às classes e TODAS as integrações com EF. Deverá ser evitado, espalhar no código, ou no modelo, integrações diretas entres os contextos sem passar pela camada única de integração BARREIRADEINTEGRAÇÃO
- Sendo assim, crie uma camada isolada que forneça ao cliente a funcionalidade necessária em termos do seu CONTEXTODELIMITADO. Ou seja, métodos criados na visão do CONTEXTODELIMITADO do cliente. Esta camada deverá conversar e mapear os 2 CONTEXTODELIMITADO, idealmente, sem ser necessário alterar os 2 sistemas. Internamente, esta camada deverá ser tradutora nas duas direções entre os 2 modelos.
- O uso de pattern como FACADE ou WRAPPER via ADAPTOR ou MUTATOR pode ser considerado. FACADE representando a interface externa, que transporta o pedido para um ADAPTOR que por sua vez chama vários TRANSLATORES, cada um responsável por uma especificidade

CAMINHOSEPARADO – Separate Ways

- Se EQUIPECLIENTEFORNECEDOR, CONFORMISTA e BARREIRADEINTEGRAÇÃO não representarem o caminho correto para integração a opção será isolar de forma independente os 2 CONTEXTODELIMITADO via CAMINHOSEPARADO
- Dentro deste contexto a regra seria: crie um CONTEXTODELIMITADO totalmente independente um do outro permitindo soluções mais especializadas em um contexto menor e mais simples

SERVIÇOABERTODEHOSPEDAGEM - Open Host Service

- Quando um subsistema deve ser integrado com vários sistemas, personalizar cada integração pode simplesmente paralisar qualquer equipe
- Sendo assim, defina um protocolo de acesso ao subsistema, SOA representa uma boa opção, oferecendo via SERVIÇO os serviços do subsistema.
- Crie tudo em formato de protocolo aberto, completamente documentado, de forma a facilitar o uso pelos clientes. Com o tempo, adicione novas funcionalidades, mantendo sempre a documentação em dia
- Havendo a necessidade de um cliente especial ter serviços personalizados, analise a questão com cuidado. E se realmente for necessário, crie SERVIÇO separados daqueles que são públicos e de uso geral

LINGUAGEMPUBLICADA - Published Language

- Existem domínios de conhecimento ou CONTEXTODELIMITADO com grande representação. Nestes casos, o mapeamento por si só pode se tornar algo demasiadamente complexo, e ele mesmo se tornar um CONTEXTODELIMITADO de mapeamento, funcionando de forma independente e tratando toda complexidade inerente ao mapeamento entre domínios
- Se existir, procure usar uma linguagem ou um modelo de mapeamento. DTD por exemplo, aplica-se a traduzir/mapear XML. XMI por exemplo pode ser usado para mapear modelos UML. CML para industria química. Mapeamento entre ontologias seria outro exemplo
- LINGUAGEMPUBLICADA representa o CONTEXTODELIMITADO ou a linguagem de mapeamento propriamente dita, responsável pela tradução, e que deverá ser usado, em detrimento a fazer internamente todo mecanismo de mapeamento

DESTILAÇÃO - Distillation

- DESTILAÇÃO representa o processo de separar os componentes de um modelo que representam a essência do domínio do negócio de forma a tornar o modelo tanto útil como valioso, e proteger suas partes de acordo com seu valor
- Um modelo representa conhecimento destilado. DESTILAÇÃO visa abstrair estrategicamente a essência do domínio de acordo com prioridades do próprio negócio
- Nosso esforço deve ser no sentido de separar o que realmente caracteriza nosso software e lhe confere um valor, e de outras partes que compõem o software mais não representam seu núcleo valorado
- Os padrões aqui relacionados podem ser equiparados com um jardineiro, que apara e corta os ramos que prejudicam os principais ramos de uma arvore em crescimento, observando e tendo em vista sempre o ramo principal

MÓDULONÚCLEO - Core Domain

- Um software difícil de entender é difícil de mudar. O desenvolvimento começa a ficar especializado, cada desenvolvedor atua em componentes específicos. Códigos duplicados começam a surgir pela falta de visão do todo, o sistema inicia seu caminho para o legado
- Uma realidade que temos que assumir é a de que existem componentes privilegiados e outros menos importantes. Os primeiros devem ser cuidadosamente refinados e de preferência pela melhor equipe. No entanto, geralmente os melhores de uma equipe se isolam em questões técnicas de infra-estrutura e nem mesmo o MÓDULONÚCLEO de um modelo, são objetos destes desenvolvedores
- MÓDULONÚCLEO representa os componentes mais importantes de um modelo e que definitivamente representam densamente o valor do software
- MÓDULONÚCLEO deverá nitidamente ser identificado em um modelo(SUB _Abstração). Nele, devem permanecer os principais conceitos ou atividades da área de negócio. Deverá ter o menor tamanho possível(não crescer demasiadamente).
- Aloque os melhores desenvolvedores para atuar no CORE. Gaste todo esforço necessário em sua modelagem, MODELOFLEXIVEL. Apoio ou refine os outros MÓDULO (sub-sistemas) em direção a se basear no MÓDULONÚCLEO

MÓDULOGENÉRICO – Generic Subdomain

- Identifique claramente subdomínios que ou não acrescentam nada do negócio em si, apesar de sua importância e domínio comum(SUB_Parametrização), ou que suportam ou

apóiam o núcleo mas não pertencem ao núcleo propriamente dito(outros SUBs). Estes subdomínios são denominados MÓDULOGENÉRICO

- Para estes MÓDULO existem 4 opções: 1ª. Compra de terceiros(off-the-shelf) 2ª. Um modelo publico 3ª. Terceirizar construção(outsourced) e 4ª. Implementar internamente(in-house)
- Só realmente inicie um MÓDULOGENÉRICO, exceção SUB_Parametrização, quando o MÓDULONÚCLEO estiver relativamente estável, visto que qualquer mudança no MÓDULONÚCLEO afeta todos os MÓDULOGENÉRICO, e investimentos inteiros podem ser perdidos
- Oriente os desenvolvedores mais experientes para o MÓDULONÚCLEO e os menos experientes para os MÓDULOGENÉRICO

DECLARAÇÃODAMISSÃO - Domain Vision Statement

- Representa um resumo ou visão executiva do propósito do modelo. Condensada em alguns poucos parágrafos de texto somente. De forma que qualquer profissional, do presidente da empresa ao programador, entenda de forma sucinta o propósito do software. Este resumo será a base durante todo o desenvolvimento do MÓDULONÚCLEO
- Sendo assim, não deve abrir linguagem técnica e sim explicitamente do negócio sendo modelado
- Deverá ser definido o mais cedo possível, e se possível, antes mesmo de existir qualquer modelo construído. No entanto, poderá ser ajustado durante o desenvolvimento do modelo, e realinhado constantemente ao longo de sua construção

IDENTIFICAÇÃODONÚCLEO – Highlighted Core

- Embora o DOMAIN VISION STATEMENT aponte na direção do MÓDULONÚCLEO, precisamos de alguma forma saber quando estamos perto ou longe como a conhecida brincadeira de quente e frio. Precisamos de meios que nos permitam explicitar o MÓDULONÚCLEO, ou seja, modelar seus componentes e definir sua fronteira
- Obviamente encontrar MÓDULOGENÉRICO e definir seus correspondentes MÓDULO(sub-sistemas + componentes) e seus relacionamentos(Arquitetura de Software) levam indiretamente definir o MÓDULONÚCLEO. No entanto, estamos em um ponto do projeto onde só temos o DECLARAÇÃODAMISSÃO, portanto, precisamos dentro deste contexto identificar o MÓDULONÚCLEO
- Sugerimos a criação de diagramas chamados DESTILAÇÃO dentro do seu modelo UML. Vários destes diagramas, no inicio sem conexão entre si, seriam na verdade

rascunhos. Com o tempo, estes diagramas DESTILAÇÃO irão sumindo do modelo e o MÓDULONÚCLEO irá nascendo

- Protótipos de tela também são bem-vindos. E também algumas consultas e relatórios que o usuário esperava ver como resultados. Estes protótipos acompanhados de um modelo, mesmo no estagio de DESTILAÇÃO, nos ajudam a delimitar o MÓDULONÚCLEO

MÓDULOUTILITÁRIO – Cohesive Mechanism

- MÓDULONÚCLEO e MÓDULOGENÉRICO não representam a totalidade organizacional de um modelo. Algoritimos específicos, e algumas vezes de grande complexidade, são representativos como estrutura de dados ou soluções oriundas da ciência da computação, mas NÃO estão inseridas no contexto de negócio propriamente tido
- Por exemplo, em um software de armazenagem de container em um pátio. Embora extremamente necessário, cálculos matemáticos representados por matrizes de transformação, embora utilizados pelo próprio MÓDULONÚCLEO, não pertencem ao negócio em si. Se não pertencem ao negócio não poderão ser agrupadas MÓDULOGENÉRICO. Eles representam na verdade um MÓDULOUTILITÁRIO
- Um MÓDULOUTILITÁRIO pode ser visto com facilidade como um mecanismo ou algoritimo complexo, necessário tanto ao MÓDULONÚCLEO como a algum MÓDULOGENÉRICO, mas que não fazem parte do negócio em si. Um outro exemplo, seria um software de entrega de mercadoria. Que pela manhã, deve abastecer vários caminhões de entrega. Um algoritimo de grafos ou outro qualquer poderia ser usado para propor o menor caminho a ser seguido para realizar todas as entregas
- Da mesma forma que temos vários MÓDULO tanto no MÓDULONÚCLEO como nos vários MÓDULOGENÉRICO, isso também se aplica ao MÓDULOUTILITÁRIO. No entanto, o padrão INTERFACEINTENCIONAL deve ser observado com maior atenção aqui, devido ao fato que na verdade, estamos definindo um pequeno framework ou interface que deve ser flexível e de fácil uso para todo o resto do modelo

HOMOGENEIZAÇÃO – Segregated Core

- Com o tempo mesmo se utilizando do INTEGRAÇÃOCONTINUA o MÓDULONÚCLEO e mesmo o MÓDULOGENÉRICO sairá vagarosamente do seu eixo. E será necessário um intervenção cautelosa e criteriosa visando realizar uma "limpeza" ou "ajuste" no modelo
- HOMOGENEIZAÇÃO implica em reorganizar/rever o MÓDULONÚCLEO, MÓDULOGENÉRICO e também MÓDULOUTILITÁRIO. Revendo o MÓDULONÚCLEO em cascata, revemos MÓDULOGENÉRICO

- O objetivo é definir quais realmente são as classes, os componentes, e os relacionamentos que constituem efetivamente o MÓDULONÚCLEO. Os componentes, classes ou relacionamentos que não passarem no critério de realmente ser necessário ao MÓDULONÚCLEO, deverão ser ou movidos para algum MÓDULOGENÉRICO existente ou até mesmo ser necessário a criação de um novo MÓDULOGENÉRICO
- O mesmo posteriormente deve ser aplicado aos MÓDULOGENÉRICO

NÚCLEOABSTRATO – Abstract Core

- MÓDULONÚCLEO deverá sempre ser construído via NÚCLEOABSTRATO
- Identifique os principais conceitos do modelo procurando por classes candidatas a serem abstratas. O modelo abstrato criado deverá expressar os principais relacionamentos e interações entre as classes e componentes do MÓDULONÚCLEO. A especialização do MÓDULONÚCLEO será realizado via os MÓDULOGENÉRICO
- Realize a devida amarração técnica no MÓDULONÚCLEO criado com NÚCLEOABSTRATO. Especifique e implemente todas as regras e todas as invariâncias visando deixar o MÓDULONÚCLEO inquebrável, de forma totalmente independente dos MÓDULOGENÉRICO

ARQUITETURA - Large-Scale Structure

- Em um grande projeto, se não utilizarmos qualquer principio que nos guie ou forneça meios organizacionais para interpretarmos o modelo, suas diversas visões, utilização de patterns, ou os mecanismos de expansão e guia para construção do MÓDULONÚCLEO ou dos MÓDULOGENÉRICO, cairemos inevitavelmente em uma floresta escura e ficaremos completamente perdidos
- Defina um padrão de regras e mecanismos que garanta a expansão do modelo de forma coordenada e consistente. Permitindo um entendimento dos mecanismos que regem a criação do modelo, e o entendimento de todas as partes que o compõe.
- Em outras palavras, relacione o controle e a gerencia na construção de um modelo com a arquitetura de software. Grandes softwares devem estar fundamentos em arquitetura de software, e esta arquitetura de software, deverá por si só ser alcançada via técnicas definidas de modelagem

EVOLUÇÃODIRECIONADA – Evolving Order

- Modelar um software indiscriminadamente sem nenhum processo ou regra para criação de MÓDULO, poderá levar a um modelo anárquico, caótico e com o tempo, altamente

complexo

- A simples criação de diagrama, e o seu respectivo empacotamento em um MÓDULO, não é suficiente para fornecer uma arquitetura ao software sendo produzido
- EVOLUÇÃODIRECIONADA implica em definir uma forma explicita de criação de uma arquitetura, que seja não somente flexível, mas também extensível. Por arquitetura, implica de fato técnicas para criação de MÓDULO e distribuição de classes nestes módulos
- .Deve-se atentar para que a arquitetura definida não seja rígida demais, forçando que o modelo de negócio seja deformado para se adaptar a arquitetura, o que deverá ser conscientemente evitado
- Acreditamos que TODO e qualquer software deverá estar fundamentado em uma arquitetura de software
- Se possível o framework utilizado deverá possuir alguma forma de validar a arquitetura. Pois, com o tempo desenvolvedores desavisados podem corrompê-la, e as linguagens de programação tradicionais, não possuem uma forma intrisica de absorver conceitos arquiteturais.

PROCESSODENEGÓCIO – Business Process

- Acreditamos que todo software representando um modelo de negócio, deverá fundamentar a arquitetura de software, baseado no modelo de processos de negócio da empresa
- O PROCESSODENEGÓCIO conterá os processos e as atividades de negócio. Dentro, encontraremos os respectivos pontos sistêmicos. Que aplicando técnicas especificas nos levará a uma arquitetura componetizada. Existem razões práticas e teóricas para se definir uma arquitetura de software baseada nos processos de negócio da empresa, estas razões podem ser encontradas em técnicas que demonstram como realizar este processo
- SYSTEM METAPHOR, RESPONSIBILITY LAYER, KNOWLEDGE LEVEL, PLUGGABLE COMPONENT FRAMEWORK representam outras visões para criação de uma arquitetura para grandes projetos. Como filosofia organizacional, todos possuem seu valor, na prática em grandes projetos e dentro do contexto de negócio achamos que o PROCESSODENEGÓCIO cumpre de forma imbatível este propósito
- Fora do âmbito de contexto de negócio, outras estruturas organizacionais que levam à modelos arquiteturais diferentes podem e devem ser buscadas. Um modelo não deve ser construído sem alguma formalidade mínima que defina sua arquitetura

O Athenas Wisdom Framework 3.0

A Athenas Engenharia de Software oferece uma linha de framework chamada Wisdom, que atua dentro dos conceitos acima apresentados.

O Athenas Wisdom Framework 3.0 é uma ferramenta completa de produtividade para apoio ao desenvolvimento de sistemas para negócios. Ideal para empresas de desenvolvimento de Software, ou departamentos de TI de organizações de qualquer porte que precisem desenvolver soluções personalizadas e adequadas às demandas de seus clientes, sejam eles internos ou externos.

A versão 3.0 do Athenas Wisdom Framework representa um ganho ou salto substancial de produtividade e na forma de se produzir software.

O Athenas Wisdom Framework 3.0, desenvolvido seguindo os princípios da Engenharia de Software, está disponível em 3 produtos diferentes, de acordo com as demandas de sua empresa e seus clientes:

- Athenas Wisdom Framework 3.0 Client JavaScript
- Athenas Wisdom Framework 3.0 Server Java
- Athenas Wisdom Framework 3.0 Server C#

O Athenas Wisdom Framework Client 3.0 JavaScript está relacionado à camada USC. Foi desenvolvido usando JavaScript e orientação a objetos e obedecendo ao modelo abstrato Wisdom 3.0.

Todos os componentes do Athenas Wisdom Framework 3.0 Client JavaScript atualizam o conteúdo das páginas utilizando o DOM (Document Object Model), o que torna a atualização dinâmica e ágil.

Em conjunto com os componentes existem varias classes que tornam o desenvolvimento Orientado a Objetos mais rápido e produtivo.

Além disso o Athenas Wisdom Framework 3.0 Client JavaScript pode ser usado combinado à diversas linguagens, como HTML, PHP, Java, C# entre outras.

O Athenas Wisdom Framework 3.0 Server C# e Java estão relacionados à camada EPC. Foram desenvolvidos utilizando C# e Java respectivamente. Ambos obedecem ao modelo abstrato

Wisdom 3.0. Portanto, são intercambiáveis e possuem exatamente as mesmas assinaturas, classes e métodos.

A camada RSC é representa por bancos de dados como Oracle, Sql Server, PostgreSQL entre outros. A camada WKC inerente ao papel de intermediação não possui um tratamento diferenciado e é fornecido conjuntamente com todos os produtos da Família Athenas Wisdom Framework 3.0.

O Athenas Wisdom Framework 3.0 fornece uma base sólida, robusta, segura e escalável para o desenvolvimento de soluções voltadas às áreas de negócios.

Com ele é possível desenvolver soluções que se conectam a diversos sistemas, bancos de dados, e outros legados.

Graças à sua arquitetura componentizada, o Athenas Wisdom Framework 3.0 consegue reduzir consideravelmente os custos de manutenção e evolução das soluções com base nele desenvolvidas. O Wisdom Framework 3.0 é de fácil e rápida implantação e aprendizado, sem deixar de lado segurança, robustez e estabilidade.

Estas caracteristicas da 3ª versão do Wisdom Framework foram conseguidas graças a dez anos de evolução constante por parte da equipe Athenas Engenharia de Software, durante o desenvolvimento de projetos em clientes como os Portos do Rio de Janeiro e Santos, que precisam de sistemas estáveis e seguros que trabalhem 24 horas por dia, todos os dias.

Entre em contato com a Athenas Engenharia de Software e descubra mais sobre a família Athenas Wisdom Framework 3.0

Sobre o Autor

Rogério Magela é Engenheiro de Computação formado pela Pontifícia Universidade Católica do Rio de Janeiro (PUC-RJ), com Pós-Graduação em Telecomunicações pela Universidade Veiga de Almeida (UVA).

Já realizou palestras em faculdades como a UVA, Candido Mendes, UFRJ, e eventos como FENASOFT e COMDEX.

É autor de diversos artigos para a Developrs Magazine, e dos livros: Analise Orientada a Objetos, Projeto Orientado a Objetos (os primeiros no Brasil em orientação a objetos com UML), e Engenharia de Software Aplicada: Fundamentos e Principios

Até 1997 trabalhou como consultor e foi diretor de tecnologia de um grupo empresarial, de onde desligou-se para fundar a primeira empresa brasileira de Engenharia de Software, onde lançou o primeiro processo de desenvolvimento de softawre do Brasil em parceria com a Sterling Software e a Computer Associates.

Em 2000 fundou a Athenas Engenharia de Software, empresa nacional que oferece uma gama completa de produtos para apoio ao desenvolvimento de sistemas e soluções para o mercado de negócios, onde atua como Diretor.

Sobre a Athenas Engenharia de Software

A ATHENAS ENGENHARIA DE SOFTWARE é uma empresa brasileira estabelecida no mercado desde 1997 que oferece uma gama completa de produtos para apoio ao desenvolvimento de sistemas e soluções para o mercado de negócios.

Os produtos ATHENAS ENGENHARIA DE SOFTWARE destacam-se por sua facilidade de aprendizado e uso, auxiliando a reduzir custos e aumentar a produtividade e reduzir custos, sem deixar de lado escalabilidade, versatilidade, segurança, robustez e baixo custo de manutenção. Qualidades adquiridas ao longo de anos de constante aprimoramento.

Além disso, um portifólio completo de serviços está a disposição dos clientes ATHENAS ENGENHARIA DE SOFTWARE, como desenvolvimento, suporte, manutenção, treinamento, entre outros.

As soluções ATHENAS ENGENHARIA DE SOFTWARE visam auxiliar o desenvolvimento não apenas de sistemas, nas também de soluções personalizadas e inovadoras, que sejam confiáveis, estáveis, seguras, escaláveis, dentro dos custos e prazos estipulados sempre com qualidade.

Além da Engenharia de Software, a metodologia de desenvolvimento ATHENAS engloba elementos do CBD (Component Based Development), Objectory, Patterns e Catalysis, que serviram como orientadores dos novos processos de desenvovimento como também do padrão SOA.

Seguindo esses princípios e como possui processos claros e definidos de gestão de projetos, suporte e evolução de seus produtos e serviços, a ATHENAS ENGENHARIA DE SOFTWARE é capaz de oferecer altos níveis de qualidade.

A Athenas representa uma opção nacional sólida, que garante um resultado seguro e dentro dos valores de investimentos estabelecidos.

Estrada do Galeão 1035, Sala 227
Jardim Guanabara, Ilha do Governador
Rio de Janeiro, RJ, CEP: 21931-630
Tel/Fax: +55-21-3368-7468
athenas@athenassoftware.com.br