As Leis do Mundo dos Objetos - Melhores Práticas em Orientação a Objetos

Veja nesse artigo algumas das melhores práticas em Orientação a Objetos, vistas de forma didática a partir de exemplos divertidos e concretos.

Introdução

O advento do computador pessoal (PC) deu o impulso comercial definitivo a área de Ciências da Computação. Hoje em dia, é indiscutível a presença e importância dos sistemas computadorizados em nossas vidas, seja através do PC, Internet, celulares, PDA´s, videogames, etc.

Para atender aos anseios dessa verdadeira Sociedade Digital, sistemas complexos com milhares de linhas de código são produzidos todos os anos. Com isso uma questão se tornou imperativa: como lidar com a crescente complexidade do desenvolvimento de software?

A resposta veio através do Paradigma da Orientação a Objetos (POO).

Orientação a Objeto

Esse novo paradigma foi uma evolução natural da Programação Estruturada/Modular. A idéia em si é genial: visualizar um programa como sendo uma rede de relacionamento entre objetos. A grande sacada é dividir a complexidade do sistema em pequenas unidades lógicas (objetos), tornado-a mais facilmente gerenciável. E além do mais, é uma solução mais natural para nós, visto que lidamos com objetos diariamente no "mundo real".

No mundo físico, cada objeto têm um papel bem definido. Todos sabem para que serve um garfo, um telefone, uma caneta, etc. Podemos dizer que todo objeto têm uma ou mais finalidades (comportamento) e características físicas (atributos).

Isso não é novidade para aqueles que lidam com linguagens OO. Mas o que muitos desconhecem é que, assim como no mundo físico, o mundo dos objetos virtuais também têm suas "leis da física" que, se respeitadas, resultarão em sistemas bem projetados.

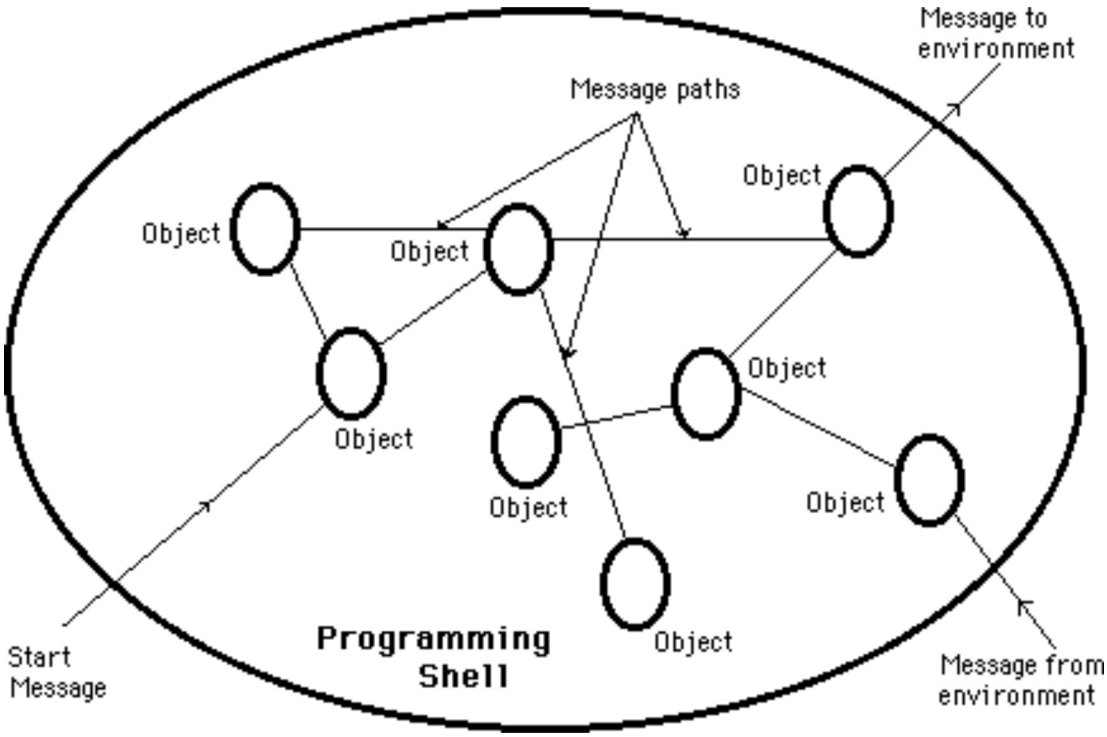

Figura 1: Rede de Objetos

As Leis da Física do Mundo dos Objetos

Você já deve ter ouvido falar, principalmente de programadores seniores e arquitetos, termos como GRASP, SOLID, Design Patterns, Débito Técnico, Coesão, Acoplamento, Lei de Demeter, Convention Over Configuration, TDD, DDD, DRY, Imutabilidade, Refactoring, etc. Essa sopa de letrinhas, embora presente amplamente na literatura sobre OO, não faz parte do cardápio de boa parte dos programadores de software, infelizmente. Por não respeitarem as "leis da física" do mundo OO, acabam gerando sistemas altamente complexos, de díficil manutenção e engessados.

Devemos ter sempre em mente que o objetivo da OO foi justamente administrar a complexidade dos sistemas, através de unidades de código (objetos) bem definidos que se relacionam entre si. A questão chave está

em como montar esses blocos de informação de uma forma que o sistema resultante seja coeso e modular.

Para isso, íremos nos debruçar sobre cada lei da física desse mundo dos objetos, para que os desenvolvedores possam visualizar as vantagens de se adotar esses conceitos na sua programação diária.

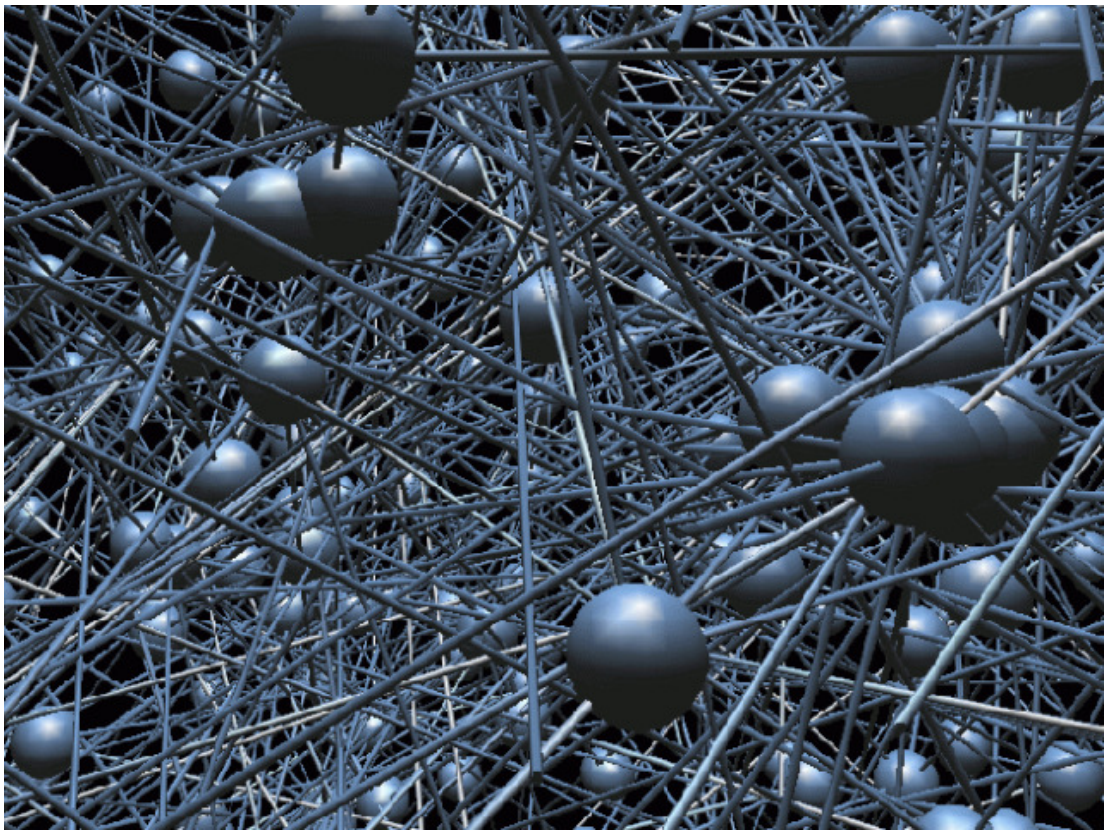

Figura 2: Sistema OO Complexo

Primeira Lei da Física OO: Dois corpos não ocupam o mesmo lugar no espaço

Segundo as leis de Newton, dois corpos não podem ocupar o mesmo lugar no espaço. Essa lei física, de fácil entendimento, é, por sua vez,

transgredida diariamente pelos desenvolvedores. Teimamos em desafiar constantemente as leis naturais, criando verdadeiras aberrações físicas, objetos frankstein's, que fazem de tudo, que agregam demais, que sabem demais.

Por exemplo, temos a seguinte classe:

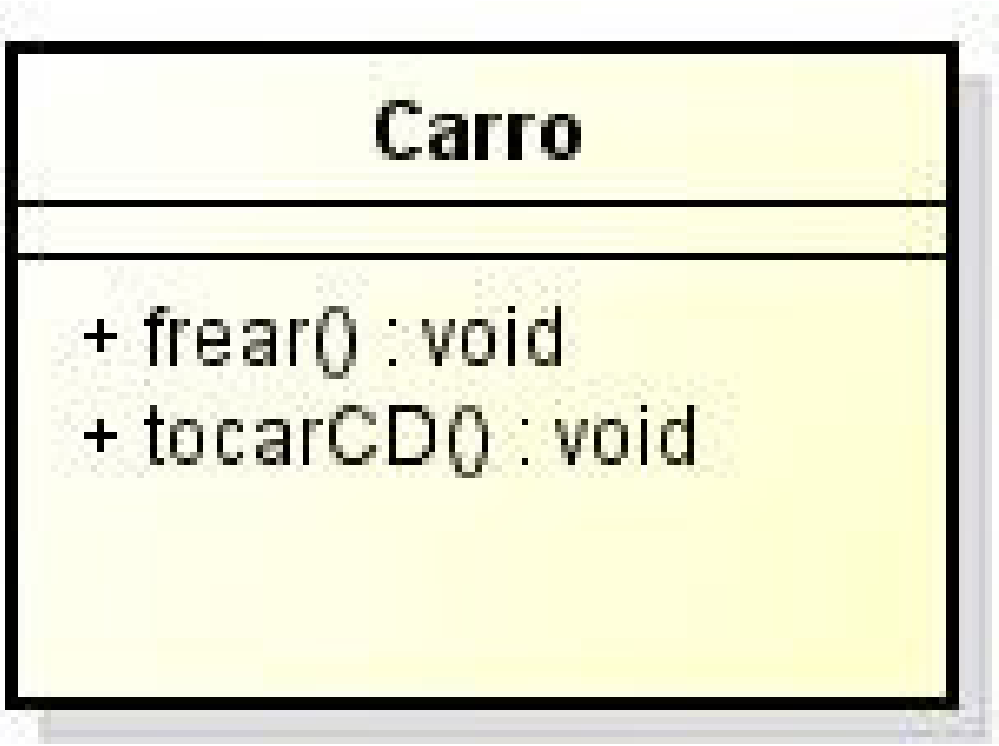

Figura 3: Classe Carro

A classe acima representa o conceito de carro do mundo físico e possue dois métodos, frear() e tocarCD(). Não há nada demais nessa classe. Será?

Se pararmos para pensar, o método tocarCD() realmente deveria fazer parte da classe Carro? Afinal quem toca o CD não é o carro, e sim o rádio embutido nele.

Uma modelagem mais adequada seria então:

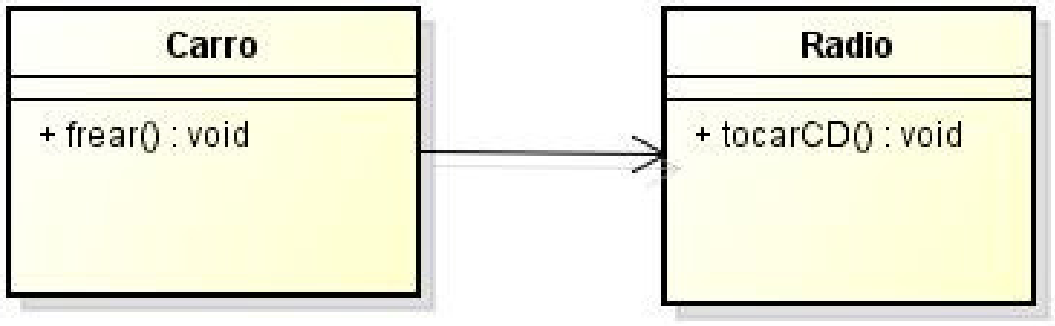

Figura 4: Classes Carro e Radio

É assim que funciona no mundo real. Cada objeto realiza o seu papel. No caso da figura 3, é como se estivessemos criando um objeto híbrido, que desempenha a função dos dois objetos, simultaneamente. Na verdade, essa classe nem merecia ser chamada de Carro, e sim Carradio (Carro + Radio).

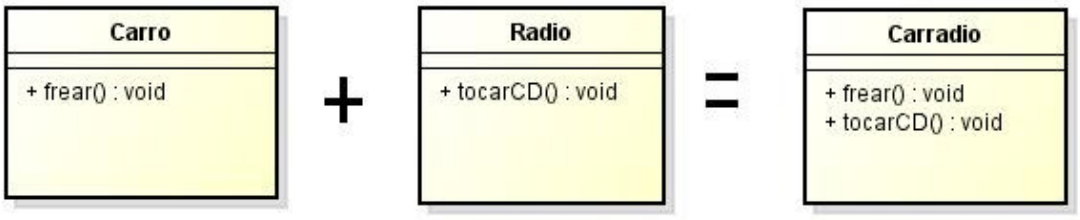

Figura 5: Uma clara violação das Leis de Newton

Evitar que objetos assumam responsabilidades que não lhe dizem respeito. É esse o significado do SRP (Single Responsibility Principle), o "S" do acrônimo SOLID.

Quando um objeto faz coisas que estão fora do seu contexto, dizemos que ele têm baixa coesão, ou baixa coerência. A classe Carro é incoerente, pois tenta desempenhar tanto o papel de Carro como de Radio, o que distorce o modelo de dados (atribuir responsabilidades aos objetos de forma coerente é conhecido também como o padrão Especialista da Informação, um dos pilares do GRASP).

Mas porque deveríamos nos preocupar em criar 2 objetos (Carro e Radio), se podemos simplesmente deixar todos os métodos na classe Carro (ou Carradio)?

Para isso, devemos observar a segunda lei da física OO.

Segunda Lei da Física OO: Princípio Fundamental da Dinâmica (F = m.a)

Figura 6: Brinquedo Lego

Essa famosa Lei de Newton nos diz que massas diferentes, submetidos a mesma força, irão produzir acelerações diferentes. Mas o que diabos isso tem a ver com OO?

Bem, quanto maior for o objeto (em termos de responsabilidade, objetos agregados, etc), maior o custo do sistema para (re)utilizá-lo.

Vejamos o fascinante brinquedo Lego, que consiste em criar diversas estruturas através do encaixe de blocos básicos. A partir de peças simples, coesas e bem definidas podemos construir obras divertidíssimas, elegantes

e complexas. Esse simples jogo demonstra um dos objetivos principais da OO: prover a fácil reutilização (e integração) de objetos para a montagem de sistemas complexos.

Quanto mais simples for o seu objeto, mais propenso a reutilização ele será. E é assim que o mundo físico funciona, pois cada objeto é a composição de outros objetos. O corpo humano é formado de orgãos, que são formados por células, que são formados por moléculas, átomos, etc. Essa é a natureza fractal do mundo real. A OO também possue natureza fractal, pois um objeto ou é único, ou é composto de outros objetos.

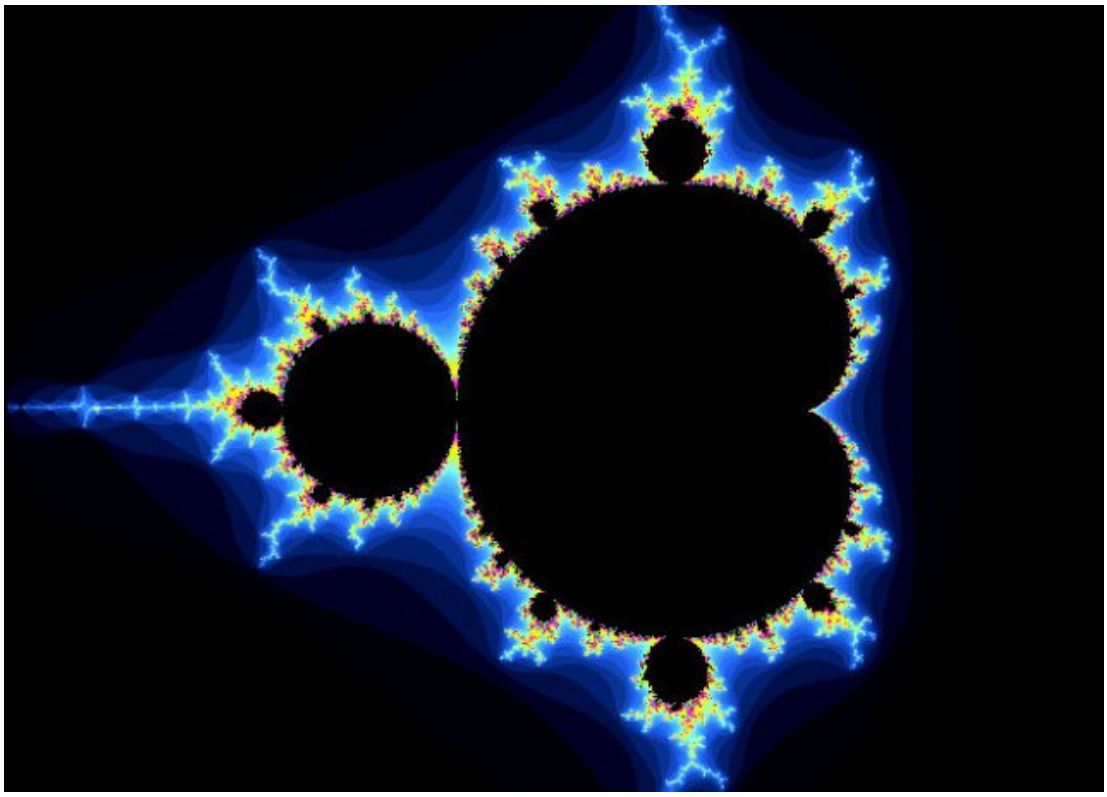

Figura 7: Fractal

Fazendo outra analogia: através da madeira, podemos construir mesas, cadeiras, paredes, etc. O esforço para converter esse objeto simples em algo mais complexo é muito menor do que converter uma cadeira em uma

mesa.

Objetos complexos possuem diversas desvantagens:

- difíceis de mudar sem quebrar a rede de objetos
- difíceis de serem reaproveitados
- elevam o tráfego de rede, seja através de RMI, XML, JSON, etc.
- dificulta a mudança de regras de negócio
- etc

Podemos, então, resumir essas duas leis nos seguintes axiomas:

- Crie objetos com uma única responsabilidade (alta coesão).
- De preferência a uma rede de objetos pequenos, especialistas, do que um objeto "faz-tudo" (granularidade).
- Objetos pequenos e coesos são mais fáceis de serem utilizados mais vezes na rede de objetos (reutilização de código).

Terceira Lei da Física OO: A Lei da Gravitação Universal

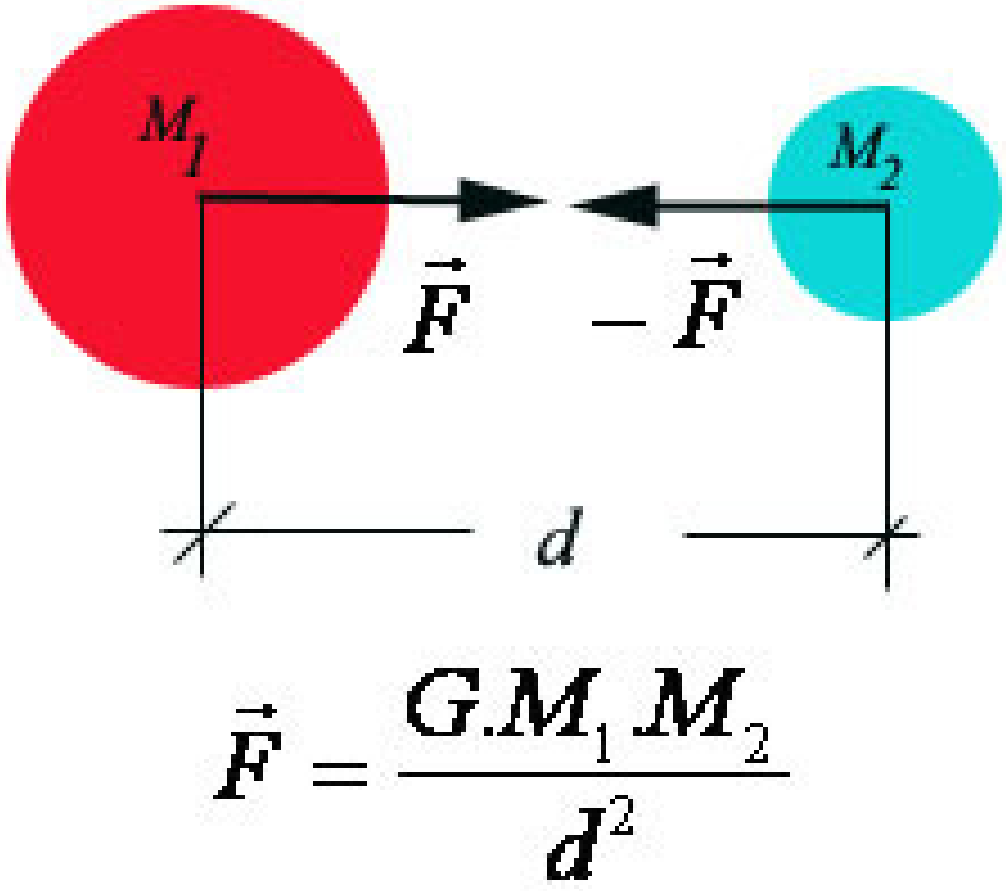

Figura 8: Lei da Gravitação

Objetos com muita massa são inconvenientes, pois necessitam de um enorme esforço para serem utilizados. Quanto maior, mais energia deve ser aplicada (realmente a gravidade é implacável).

Objeto pesados não são apenas objetos que assumem responsabilidades demais. Eles também podem ser considerados pesados quando possuem muitos componentes "agregados".

O que seriam "componentes agregados"? Uma analogia do mundo real ajudará a compreender esse conceito.

Você, como consumidor, ao comprar um carro, pode optar por trocar vários de seus componentes ao longo da vida útil do mesmo. Vidros, portas, pneus, motor, etc. Isso é possível graças ao design modular do carro, onde cada componente têm uma interface bem definida com o restante do sistema e são extremamente coesos. Podemos trocar a bateria do carro, optando por diversas marcas, bastando que elas sigam a especificação que define como ligar a bateria com o resto do sistema daquele modelo de carro. O mesmo pode ser dito de computadores, motos, casas, etc.

Agora imagine o seguinte cenário: ao comprar um carro, o fornecedor impossibilita alterações nos componentes, limitando a nossa liberdade de customização do mesmo. Isso com certeza seria um absurdo. Tão absurdo quanto uma imobiliária definir os móveis que devemos comprar na casa que adquirimos dela.

Se os consumidores ficam irritados quando compram produtos que não permitem alterações, porque os programadores (que também são consumidores) produzem classes amarradas, limitando o uso por parte de outros programadores? Para ilustrar esse cenário, veja o seguinte exemplo:

Listagem 1: Classe Carro

```
public class Carro {
    ....
    public Carro() {
        this.motor = new MotorFord();
        this.portas = new ArrayList<Porta>();
        this.pneu1 = new Goodyear();
        this.pneu2 = new Goodyear();
        this.pneu3 = new Goodyear();
        this.pneu4 = new Goodyear();
        this.radio = new Pionner();
    }

    public void tocarCD(CDRom cdrom) {
```

```
            this.radio.tocarCD(cdRom);
        }

        public void buzinar() {
            this.buzina = new BuzinaTabajara();
            buzina.ativar();
        }
        ...
    }
```

O uso do operador new é indicador forte de um príncipio claro: imposição. O criador dessa classe fez o mesmo papel do fornecedor mesquinho que lhe vendeu aquele carro não-customizável, pois ao definir previamente os componentes do seu carro, retirou sua liberdade de escolha.

Em OO isso é conhecido como forte acoplamento entre objetos, e a solução para isso é conhecida como Princípio da Inversão da Dependência (outro componente do SOLID) ou Inversão de Controle.

A partir da analogia com o consumidor, fica claro entender o papel da Inversão de Dependência: permitir a customização de objetos e módulos. Para que isso seja possível, devemos trabalhar com abstrações e com a estratégia de Inversão de Controle.

A Inversão de Controle é conhecida também como o príncipio de Hollywood ("*não nos chame, nós chamamos você*"). Em OO, diríamos que a classe Carro irá transferir a responsabilidade de criação de suas dependências para um contêiner de injeção de dependência (Spring, Guice, PicoContainer), que é uma forma de se implementar Inversão de Controle.

O exemplo do Carro têm uma característica emblemática. Podemos trocar vários componentes do carro, não importa de que fornecedor ele venha, com uma única condição: que siga a especificação padrão do componente para aquele tipo de carro. A palavra chave aqui é a especificação.

A especificação é um contrato entre o componente e o resto do sistema. O que se deve levar em consideração são as funcionalidades que o componente provê ao mundo exterior, e não como essas funcionalidades são implementadas. Temos 2 conceitos claros então: a utilidade de um objeto e como ele implementa isso. Para ficar mais claro, usaremos um exemplo do mundo real: o garfo.

O garfo pode ser de metal, de plástico ou até de madeira, mas isso não interfere no seu próposito, sua utilidade, sua atividade fim. Fica claro então como a finalidade de um objeto é diferente da sua implementação, que pode variar.

Esse conceito de especificação, de contrato, dos serviços que um objeto pode fornecer são implementados através de abstrações como as interfaces, classes abstratas, polimorfismo, herança e até padrões de projeto em linguagens OO.

O Príncipio de Inversão de Dependência é fundamental para a OO, pois atuando juntamente com Abstrações e Desig Patterns, possibilita a criação de redes de objetos coesas e leves. Afinal, decidir a criação das dependências (padrão Criador do GRASP) pode ser responsabilidade demais para nossos estimados objetos.

Os axiomas dessa lei, são portanto:

- Favoreça o desacoplamento de código (baixo acoplamento), através de conceitos como Inversão de Dependência ou Inversão de Controle.
- Uso de abstrações / padrões para separar funcionalidade de implementação. (encapsulamento, herança, polimorfismo).
- Uso de abstrações / padrões para separar módulos do sistema (um módulo também pode ser visto como um objeto, geralmente representando uma rede de objetos. Por ser um objeto, está sujeito as mesmas leis da física do mundo OO, ou seja, o relacionamento

entre os módulos também deve também privilegiar coesão, baixo acoplamento, etc. Para isso existe conceitos como OSGI, SOA, etc).

Quarta Lei da Física OO: Teoria do Caos

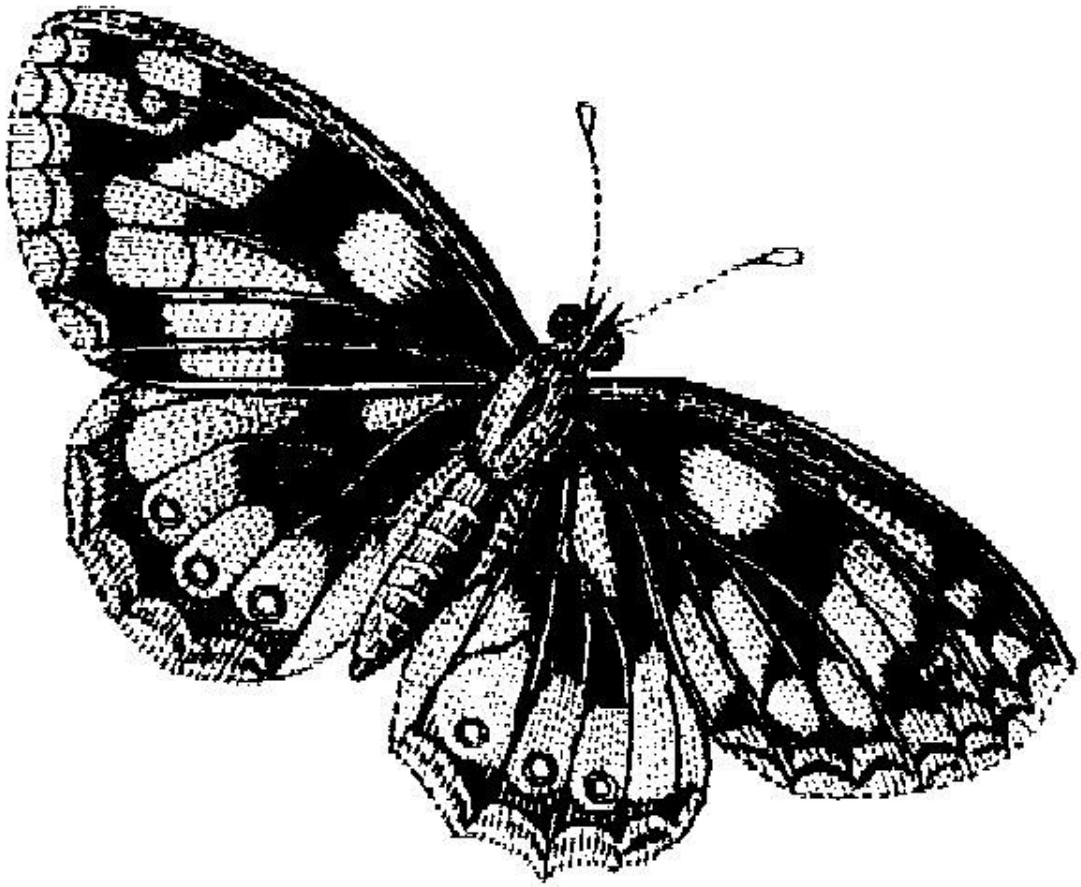

Figura 9: Efeito Borboleta

A teoria do caos diz que, qualquer mudança no sistema, por menor que seja, pode levâ-lo a um comportamento imprevísivel ao longo do tempo. O exemplo clássico é o chamado efeito borboleta: um simples bater de asas de uma borboleta no Texas pode gerar um tufão em outra parte do mundo, o que demonstra a intrincada interligação do mundo.

Sistemas dinâmicos estão sujeitos a esse efeito, pois lidam com dezenas de

váriaveis que influenciam na complexidade. Quanto maior a quantidade de variáveis do sistema, mais complexo ele será, e mais propenso a imprevisibilidade também.

O que diremos então dos sistemas com milhares de linhas de código, que lidam com centenas de regras de negócio, interação entre objetos a nível micro (classes) e macro (módulos, subsistemas), etc? Alguma vez o seu Windows já apresentou um comportamento inesperado?

Afinal, regras de negócio e requisitos mudam constantemente, forçando os programadores a alterar essa malha de objetos, o que tende, se for feita de forma descuidada, a gerar as consequências do efeito borboleta, ou seja, comportamento inesperado do sistema (bugs).

Nota: Em IA, comportamento inesperado ás vezes é bem vindo, principalmente de algoritmos genéticos, redes neurais, etc.

Numa rede de objetos, todo e qualquer objeto e seus relacionamentos são importantes. Uma única linha de código incorreta num sistema com milhares de linhas de código pode comprometer a imagem do mesmo. A rede de objetos é como uma corrente, cujo valor equivale ao seu elo mais fraco.

Um dos conceitos do nosso ramo científico que merece destaque é o TDD (Test Driven Development – Kent Beck). O TDD, se seguido corretamente, pode garantir a qualidade do seu sistema, pois ele ataca o problema da complexidade a nível atômico (classes e objetos individuais), propagando-a nos níveis subsequentes. Uma analogia sempre é bem vinda. No exemplo do carro, você espera que todos os seus componentes tenham sido testados (motor, freio, cd player, buzina, injeção eletrônica, pneus, etc), não é mesmo?

Realmente, o ramo de Engenharia é muito mais maduro do que a Ciências

da Computação, pois os engenheiros conseguem entregar redes de objeto complexas (imaginem quantos transistores, resistores, diodos, etc uma CPU possue?) com uma taxa de erros ínfima. As técnicas OO são o primeiro passo que todo o desenvolvedor deve seguir se quiser construir sistemas com o mínimo de qualidade possível. A passos curtos, a Ciências da Computação vai ser tornando cada vez mais madura, mas ainda falta muito para chegar ao nível das outras ciências exatas.

Os axiomas dessa lei, são portanto:

- Use o TDD para controlar a complexidade no nível quântico (objetos), estendendo aos módulos da baixo nível para os de alto nível.
- Use ferramentas que possibilitem todo o tipo de teste: JUnit, JMock, Selenium, JMeter, etc.

Quinta Lei da Física OO: Entropia

Figura 10: Entropia

A Entropia, em linhas gerais, mede o grau de "desordem" de um sistema, até que ele caminhe para a irreversibilidade.

Em OO, o acumulo de más práticas propicia o surgimento do chamado Débito Técnico. O Débito Técnico seria então, um indicador do custo de se alterar um sistema. Quanto maior for, mais esforço é necessário para se alterar uma regra sem gerar comportamento imprevísivel (bugs).

Para lidar com a crescente complexidade, e a mantê-la administrável, não basta apenas seguir as quatro leis anteriores. Elas ajudam e muito a manter o sistema coeso e previsível, mas elas são leis estruturais, que definem conceitos como responsabilidade, criação, dependência, granularidade, coesão, ou seja, a parte arquiteturial da rede de objetos. E quanto ao lado dinâmico da rede?

Como gerenciar então as mudanças de regras, a intrincada rede, cuidar tanto a nível quântico como macro da relação entre objetos? Para isso, um grupo de físicos do mundo OO (Kent Beck, Martin Fowler, Craig Larman, Eric Gamma, Eric Evans, Uncle Bob, Scott Ambler, etc) criou técnicas como TDD, Refactoring, DDD, DRY, Convention Over Configuration, Integração Continua, etc (na verdade, muitos dessas técnicas são conhecidas desde os primórdios da programação Modular, mas que foram difundidas por esses cientistas da computação).

Não se preocupe agora com essas siglas. Tenha em mente que o objetivo principal delas é atacar a complexidade do sistema, garantindo a qualidade do mesmo em todos os níveis quânticos e relativistas, minimizando a imprevisibilidade.

Os axiomas dessa lei, são portanto:

- Controlar a complexidade do sistema através de técnicas consagradas como Integração Continua, TDD, Refactoring, Inversão de Controle, DDD, etc
- Abordagem atômica: todas as peças da engrenagem devem estar funcionando corretamente. Utilize TDD para garantir a qualidade a nível quântico.
- Sistemas de computador são sistemas dinâmicos e devem ser tratados com seridade.

Conclusão

Espero que esse artigo possa contribuir para o entendimento das importantes Leis OO que regem o mundo dos objetos virtuais. Afinal, se os engenheiro conseguem entregar redes de objeto gigantescas com alta qualidade, nós, cientistas da computação, não podemos ficar para trás. O foco é a qualidade.

Abraços e até a próxima!

Nota: Para mergulhar nessas "leis da física", sugiro a leitura dos livros indicados em Referências.

Referência:

- http://en.wikipedia.org/wiki/GRASP_%28object-oriented_design%29
- http://www.argonavis.com.br/cursos/java/j930/tutorial/Design_Patterns.pdf
- http://pt.wikipedia.org/wiki/Lego
- http://pt.wikipedia.org/wiki/Efeito_borboleta
- http://www.infoq.com/br/news/2009/10/dissecting-technical-debt

Leitura Recomendada:

- UML e Padrões – Craig Larman
- Programador Pragmático - David Thomas Roberts; Whiteford, Andrew Hunter
- Clean Code – Robert C. Martin
- Agile Software Development, Principles, Patterns e Practices – Robert C. Martin

- Refactoring – Martin Fowler
- Design Patterns – Gang Of Four
- Effective Java – Joshua Bloch
- Padrões Arquiteturiais de Aplicações Corporativas – Martin Fowler
- Test Driven Development – Kent Beck
- Domain Driver Design – Eric Evans

por Marcelo Senaga
Engenharia de software lover ♡