Sessão – Nomes

Em uma análise superficial sobre um código, uma das afirmações que poderiam ser feitas seria que sua composição é basicamente um conjunto de palavras reservadas da linguagem e um número imenso de nomes escolhidos pelos desenvolvedores. Um código Smalltalk praticamente não contém nenhuma palavra relevante para a sintaxe, uma vez que quase todo o trabalho pode ser feito somente através de envio de mensagens para os objetos.

Ao nos deparar com essa realidade, a primeira atividade que deveríamos nos focar para o desenvolvimento de um código limpo é a escolha de bons nomes. A seguir serão listados alguns aspectos que podem nos auxiliar a encontrar bons nomes que se encaixam no estilo de programação buscado ao longo do trabalho.

Nomes que Revelam Intenção

A seguinte afirmação foi retirada do livro Clean Code: "O nome de uma variável, função ou classe deveria responder todas as grandes questões. Deveria contar porque o elemento existe, o que faz e como deve ser usado. Se um nome precisa de um comentário, então ele não revela sua intenção." O exemplo abaixo, extraído do mesmo livro, ilustra a dificuldade do leitor compreender um código com nomes não reveladores.

```
caminhoR(Vértice v):
	Vértice w
	est[v] = 0
	for(w = 0; w < tamanho(); w++)
		if(adj(v,w))
			if(est[w] == -1)
				caminhoR(w)
```

Esse método pertencente à uma classe *Grafo* é difícil de ser compreendido sem um vasto conhecimento da convenção de notações e o algoritmo utilizado. Muitas perguntas poderiam ser feitas como "O que faz *pathR*?", "O que é o vértice v? E quanto ao w? Eles tem algo em comum?", "O que significa adj(v, w) ser igual a 1?" e muitas outras.

A seguir está o mesmo código com alterações somente nos nomes das variáveis e métodos, sem nenhuma alteração nos detalhes da implementação.

```
passeieComOrigemEm(Vértice origem):
	Vértice próximo;
	estado[origem] = JÁ_VISITADO
	for(próximo = 0; próximo < numero_de_vértices(); próximo++)
		if(são_adjacentes?(origem, próximo))
			if(estado[próximo] == NÃO_VISITADO)
				passeioComOrigemEm(próximo)
```

Muitas questões podem ser respondidas depois da leitura deste trecho. O nome da função deixa claro que sua atividade é passear pelos vértices do grafo começando pelo vértice origem passado como parâmetro. O vetor *est* foi renomeado para *estado* e as constante *0* e *-1* receberam um nome, revelando o significado das operação em que constam. Até mesmo mudança de *tamanho()* para *numeroDeVertices()* provê um contexto mais completo para o entendimento do laço.

Claramente o exemplo poderia ser melhorado para ficar mais alinhado com o paradigma da orientação a objetos e com os conceitos que serão apresentados mais adiante. Porém, a importância da escolha de nomes não pode ser questionada. É possível revelar o significado de elementos dando nomes para as constantes, por exemplo. Nomes como *origem* e *próximo* podem nomear objetos da mesma classe para mostrar o contexto e semântica de como estão sendo usados. Métodos podem trazer mais expressividade e fluência à leitura se forem nomeados levando em conta o código cliente e nas circunstâncias em que será chamado.

Sessão - Funções/Métodos

As funções são o núcleo fundamental para a criação de um código expressivo. Pensando nas ideias levantadas por Grady Booch e Kent Beck, queremos que nosso código seja lido como um texto em prosa e, nessa analogia, a intenção é utilizar os nomes dos métodos para narrar a execução.

Tendo em mente os princípios já abordados anteriormente, buscamos minimizar as repetições e queremos que as mudanças sejam localizadas, de forma que a lógica esteja perto dos dados que manipula. Nesse sentido, as funções serão muito úteis para encapsular um trecho de código, provendo um escopo fechado para variáveis locais e permitindo chamadas como um maneira de evitar duplicações.

Fundamentação Teórica

O primeiro fundamento levantado no livro *Clean Code* quanto aos métodos é uma preocupação quanto ao tamanho. Segundo Robert Martin, métodos devem ser pequenos (e se possível menores do que isso). O autor também cita que não existe uma fundamentação científica para tal afirmação, além de não podermos afirmar o que é “pequeno” e definir limitantes quanto ao número de linhas.

A proposta é que cada função seja pequena o suficiente para facilitar sua leitura e compreensão. Devemos ter em vista a dificuldade de assimilação de grandes porções de informação durante a leitura e o fato de que nem sempre uma instrução é clara. A partir dessas ideias, uma função será uma porção curta de código que trabalha com poucas variáveis e tem um nome explicativo que espelha a sua funcionalidade.

Uma maneira bastante interessante de pensar sobre o tamanho das funções não é simplesmente contar seu número de linhas, mas compreender bem a atividade que realiza. Nas palavras de R. Martin, “Funções deveriam ter uma única atividade. Deveriam fazê-la bem. E fazê-la somente.”

Isso significa que para criar uma lista de números primos usando o algoritmo do Crivo de Erastótenes, não queremos ter uma única função que contenha todos os detalhes da criação de uma lista de inteiros e a marcação de múltiplos que não tem chance de serem primos. Seria muito difícil entendê-la e modificar alguma das minúcias da implementação. Queremos várias pequenas funções que fazem cada uma das atividades necessárias para essa computação, de forma que a leitura seja suficiente e uma documentação externa redundante.

Outra questão teórica importante e enfatizada por Kent Beck são os níveis de abstrações e a simetria de operações dentro de um método. Um incremento de uma variável de instância está fundamentalmente em outro nível de abstração do que a chamada de uma outra função. Não queremos uma função com essas duas instruções. Isso possivelmente faz com que o leitor não saiba se uma operação é um detalhe de implementação ou um conceito importante dentro da lógica, além de abrir as portas para mais ocorrências deste mesmo tipo, aumentando a complexidade do código a cada alteração.

Pensando em outros fatores que podem levar em dificuldades para a leitura do nosso código, temos que considerar a afirmação de Ward Cunningham relativa à um código limpo: “Sabemos que estamos trabalhando em um código limpo quando cada rotina que lemos faz o que esperávamos”. Refletindo esse pensamento para as funções, como seus nomes são a documentação da atividade que fazem, não queremos que, ao olhar o corpo de uma delas, nos deparemos com uma operação que não imaginávamos. Ao criar um método, temos que considerar que os leitores não terão a mesma linha de pensamento que temos naquele momento e, diante disso, temos que ter uma crescente preocupação com todas as nossas decisões.

O último fundamento teórico quanto as funções também diz respeito ao programa como um todo. Queremos ter um fluxo normal bem estabelecido, deixando-o simples e o tratamento de erros separado. Ou seja, não queremos que o código fiquei cheio de verificações de erro misturados com a lógica sem erros.

Técnicas

Tendo em vista a fundamentação teórica, a seguir estão algumas técnicas que nos possibilitam criar funções que se encaixem no estilo de programação buscado.

Composição de Métodos

A composição de método é a base para a criação de um código limpo. A proposta é compor nossos métodos em chamadas para outros métodos rigorosamente no mesmo nível de abstração.

Usando novamente o exemplo do algoritmo do Crivo de Erastótenes, para encontrar os primos de 1 a N devemos criar um conjunto de elementos que represente os inteiros de 1 a N, marcar todos números múltiplos de outros e depois coletar todos aqueles que não estão marcados (os primos). Para isso criamos um método da classe *GeradorDePrimos*:

```
primos_ate(N):
      cria_inteiros_desmarcados_ate(N)
      marca_os_múltiplos()
      coloca_os_não_marcados_no_resultado()
      retorna resultado
```

Nesse exemplo, compusemos o método *primos_ate(N)* em chamadas para outros métodos, cada um com uma atividade e um nome explicativo. Dessa forma, o leitor pode entender facilmente como funciona o algoritmo de uma maneira geral.

Também ficam bastante claros os níveis de abstrações criados. A função *primos_ate(n)* está em um nível e os métodos que invoca estão todos no mesmo patamar logo abaixo. Queremos criar essa estrutura em cadeia, tornando possível que o leitor só precise entender os níveis que interessem no momento.

A composição de métodos não é uma técnica que desconhecemos, uma vez que a todo momento criamos estruturas dessa maneira sem o ter conhecimento da sua efetividade. Porém, segundo o autor de *Implementation Patterns* em uma entrevista para a InfoQ, a facilidade de leitura propiciada faz com que esse seja o padrão mais relevante do livro.

Métodos Explicativos (*Explaining Methods*)

Para auxiliar na expressividade de um trecho de código, podemos criar uma função com nome específico do contexto em que será invocada. A operação pode ser bastante simples como um incremento de variável ou a chamada de um método em um objeto guardado em uma variável de instância, mas o nome dado será de grande valor para a documentação do código. Além disso, a função pode ser reutilizada posteriormente evitando duplicações e encapsulando a operação.

Essa ideia pode ser empregada em diferentes contextos e níveis de abstração. Uma circunstância pode ser ilustrada pela afirmação de Kent Beck: "Considero a criação de um método explicativo quando fico tentado a comentar uma única linha de código". Abaixo está um exemplo em Java extraído do livro do mesmo.

```
flags |= LOADED_BIT; //Set the loaded bit
```

O comentário pede uma mudança usando um método explicativo.

```
void setLoadedBit() {
      flags |= LOADED_BIT;
}
```

Em outro exemplo, criamos um método chamado *highlight()* para dar o contexto necessário para uma operação de nível mais baixo.

```
def highlight(word)
      word.background_color(Color.yellow)
end
```

Considerando a região de código que chama highlight(), as melhorias são diversas com essa alteração. Se a operação fosse simplesmente colocada sem nenhum comentário, provavelmente o leitor se perguntaria porque a cor de fundo da palavra está sendo alterada naquele contexto. O comentário seria algo como "destacar a palavra no texto", o que se torna obsoleto com o método explicativo.

Por fim, ainda há formas mais implícitas de utilizar métodos explicativos. Quando utilizamos o padrão de projeto *Wrapper* para criar uma classe que abstrai uma coleção de elementos, estamos de uma maneira criando uma interface cujos métodos façam mais sentido para o domínio de aplicação trabalhado.

Funções como Condicionais

Um caso específico de método explicativo bastante utilizado é uma chamada para uma função que contenha uma expressão booleana como valor de retorno.

```
if(dobra[inicio] != dobra[inicio+1] && inicio < n)
```

Um condicional como o mostrado acima não é muito claro e deixa o próprio autor confuso quanto ao seu significado depois de certo tempo após sua redação. A criação de uma função que contenha a expressão

booleana nos permite escolher um nome apropriado para a circunstância, deixando o condicional claro.

```
ainda_ha_dobras_a_serem_feitas():
      return dobra[inicio] != dobra[inicio+1] && inicio < n
```

Evitar Estruturas Encadeadas

Tendo em vista a busca por funções pequenas e com uma única atividade, um aspecto relevante é a criação de estruturas encadeadas. Se um método tem uma cadeia de *if*s e *else*s, o leitor terá dificuldades para compreender todos os casos e fluxos possíveis.

Em ambos os livros trabalhados, os autores enfatizam sobre o uso de chamadas de funções logo em seguida de um condicional. Novamente a operação que será encapsulada estará dentro de um método com nome expressivo, deixando a função que a continha curta e expressiva.

Novamente utilizando o exemplo do Crivo de Eristótenes, é necessário marcar todos os números que sejam múltiplos de um primo, representado por um número não marcado.

```
marca_os_múltiplos():
      for(candidato = 0; candidato <= limite; candidato++)
            marca_múltiplos_se_não_esta_marcado(candidato)

marca_múltiplos_se_não_esta_marcado(candidato):
      if(não_marcado(candidato))
            marca_múltiplos_de(candidato)

marca_múltiplos_de(primo):
      for(múltiplo = 2*primo; múltiplo < limite; múltiplo += i)
            números[múltiplo].marca()
```

Cada função encapsula um nível na cadeia de estruturas encadeadas. Ao invés de muitos *for*s e *if*s encadeados, obtivemos funções pequenas e muito focadas em uma única atividade que podem ser mais facilmente testadas de forma independente.

Cláusulas Guarda (*Guard Clauses*)

Outra técnica para evitar o uso de estruturas complexas de condicionais são as cláusulas guarda. Quando criamos uma expressão condicional (*if*), o leitor naturalmente espera um bloco com a contrapartida (*else*). A ideia é criar um retorno breve para expressar que uma das partes dessa contraposição é mais relevante. Portanto, buscamos a estrutura à direita sobre à esquerda no exemplo abaixo, pois estamos dizendo ao leitor que o fluxo mais relevante é o caso em que ocorre a inicialização.

```
inicializa():
      if(!já_inicializado)
            ...
```

```
inicializa():
      return if(já_inicializado)
            ...
```

Objeto Método (*Method Object*)

Ao dividir os métodos em outros menores com uma única atividade, podemos criar métodos com grande complexidade e difícil refatoração: muitas variáveis, muito parâmetros necessários e muitas estruturas

encadeadas. Diante dessa situação, podemos criar um objeto que encapsule toda essa lógica.

Suponha que temos um método complexo chamado *operaçãoComplexa()*. Criamos uma classe com nome relacionado como *OperadorComplexo* cujas variáveis de instância são os parâmetros que a função recebia anteriormente. Em um método chamado *calcula()*, por exemplo, colocamos a funcionalidade que o método original continha. Com essa mudança, todos os detalhes dessa operação complexa foram encapsulados em uma classe que terá testes, aumentando nossa confiança e flexibilidade para refatorações.

Tal solução está bem inserida diante do paradigma da orientação a objetos uma vez que a nova classe será coesa e conterá uma estrutura complexa cujos clientes não terão que conhecer. Portanto, os clientes terão seu código simplificado através de uma delegação.

Minimizar os Argumentos

O número de argumentos se torna bastante importante quando queremos métodos pequenos e com apenas uma atividade. Se uma função recebe muitos argumentos, provavelmente os utiliza para um conjunto de operações e não uma somente. Diante disso, nosso objetivo é sempre minimizá-los por algumas razões.

Primeiramente, testes de funções com poucos ou nenhum argumento requerem menores esforços uma vez que as combinações de casos de teste são limitadas. Além disso, teremos um maior acoplamento da classe em que a função está contida com todas as classes dos objetos que recebe, já que é necessário conhecer suas interfaces para utilizá-los. E por fim, o leitor terá que entender todos esses casos de uso.

Com essa motivação queremos evitar parâmetros desnecessários e que confundem o leitor. Abaixo estão algumas clarificações citadas pelos autores.

Evitar *Flags* como Argumentos

Se passamos uma Flag (booleana ou proveniente de uma enumeração) como argumento para gerar dois tipos de comportamentos de um método, claramente esse último faz mais do que uma atividade. O natural seria criar um método para cada uma das atividades como mostrado à direita no exemplo de uma classe *Figura* abaixo.

```
rotaciona(ângulo, sentido_horário):
     if(sentido_horário == true)
          this.ângulo += ângulo
     else
          this.ângulo -= ângulo
```

```
rotaciona_sentido_horário(ângulo):
     this.ângulo += ângulo

rotaciona_sentido_anti_horário(ângulo):
     this.ângulo -= ângulo
```

Objeto como Parâmetro

Diante de uma lista grande de argumentos, devemos questionar se não existe alguma abstração que une esses elementos. Por exemplo, considere uma classe *FábricaDeMáquinas* cujo construtor é FábricaDeMáquinas(número_de_máquinas, número_de_funcionários, horas_de_funcionamento). Uma possibilidade seria criar um objeto *EspecificaçãoDeFábrica* que contivesse esses argumentos como variáveis de instância e métodos que fossem úteis para que o código desta *FábricaDeMáquinas* fosse o mais claro possível.

Parâmetros com Variável de Instância

Na utilização da composição de métodos, freqüentemente será preciso passar um argumento para diversos métodos, que possivelmente não o utilizam, somente para satisfazer a necessidade de um deles. Uma solução plausível é transformar esse argumento em uma variável de instância acessível por todos os métodos.

Esse tipo de alteração tem grande importância no design do sistema como um todo. Na sessão de Classes abordaremos esse tópico e veremos o papel dessa técnica em refatorações de maior escala.

Uso de Exceções

Como foi dito nos fundamentos teóricos dessa sessão, queremos ter um fluxo normal bem definido e sem interferências do código de tratamento de erros. Nesse contexto, devemos preferir o uso de exceções sobre retornar códigos de erro e valores nulos.

O grande problema com o retorno de códigos de erros e valores nulos está na limpeza do código cliente. Se, ao chamar um método, é necessário utilizar condicionais para verificar qual foi o erro causado ou certificar que não chamaremos um método sobre uma referência nula, o código cliente provavelmente terá que lidar com estruturas encadeadas e muitos casos de teste. Essa decisão viola os princípios de um código expressivo para o leitor uma vez que, durante a leitura, terá que compreender cada um destes erros mesmo que os mesmos não sejam interessantes no momento.

Sessão X - Classes

Entre os princípios propostos por Kent Beck estão a proximidade da lógica com os dados na qual trabalha e a preocupação quanto às conseqüências locais. Esses conceitos além de estarem inseridos na justificativa para a adoção do paradigma da orientação a objetos, também estão intimamente relacionados com as classes que compõe o sistema. Queremos que as classes encapsulem dados e operações e tenham uma interface que permita um código cliente com o mínimo de dependências, sempre visando a simplicidade do design e do conteúdo das classes.

Diante da concepção ágil de tomar boas decisões durante o desenvolvimento que proporcionem melhorias imediatas, nessa sessão serão discutidas algumas preocupações quanto as classes, buscando facilitar as mudanças no sistema.

Muitas Classes Pequenas

Do mesmo modo que consideramos importante limitar a quantidade de informação que o leitor tem que se deparar quando lê funções, queremos que as classes sejam o menores possível. Além de facilitar a leitura e entendimento, programar buscando minimizar o tamanho das classes nos auxilia a criar unidades coesas e a evitar duplicações.

Logicamente, se nossas classes são pequenas, teremos que reunir uma grande quantidade delas, tornando nossos sistemas compostos de muitas classes pequenas. Em uma analogia bastante simples, é mais fácil encontrar um objeto em muitas gavetas pequenas do que em poucas gavetas grandes e lotadas.

Princípio da Única Responsabilidade (Single Responsibility Principle)

A questão que surge após a afirmação que as classes deveriam ser pequenas é como definir o que é ser pequena. O número de métodos pode nos dar um bom indicativo quanto a quantidade de operações diferentes que a classe pode executar, mas podemos encontrar classes com poucos métodos e que possuem contextos totalmente diferentes. Por exemplo, uma classe Controlador com um método os captura_entrada() cria_corpo_html e cria_cabeçalho_html() possui dois tipos de atividades completamente diferentes.

A forma de medir o tamanho de uma classe proposto por Robert Martin está atrelado com a quantidade de responsabilidades que a mesma possui. Podemos pensar em uma responsabilidade como uma razão para mudar. No exemplo acima, para que o sistema seja adaptado para utilizar um novo padrão HTML, é necessário alterar a classe Controlador. O mesmo teria que ser feito caso um novo tipo de entrada do usuário fosse concebido. Dessa forma, dizemos que o Controlador tem pelo menos duas responsabilidades: lidar com a criação de HTML e processar a entrada do usuário.

Dentro do estilo de programação buscando, queremos que nossas classes sigam o Princípio da Única Responsabilidade que, de maneira geral, afirma: as classes deveriam ter uma única responsabilidade, ou seja, ter uma única razão para mudar. Seguindo o princípio SOLID acima exposto por Robert Martin em Agile *Software Development, Principles, Patterns, and Practices*, queremos evitar classes como o Controlador.

Coesão

De um ponto de vista técnico, qual crítica poderia ser feita sobre a classe Controlador?

A classe Controlador não é coesa. Se tal classe tem a responsabilidade de lidar com a criação de HTML, logicamente terá um conjunto de variáveis de instância e métodos que as utilizam capazes de executar o trabalho em conjunto. O mesmo pode ser dito para o processamento da entrada do usuário. Um leitor que precisa entender apenas um desses contextos se deparará com inúmeros detalhes de implementação tanto da responsabilidade que deseja trabalhar, quanto da que não está interessado.

Nas palavras do autor de *Clean Code*, “Quando a coesão de uma classe é alta seus métodos e variáveis são co-dependentes e se unem como uma unidade lógica”. De maneira mais prática, uma classe é totalmente coesa quando todos os seus métodos usam todas as variáveis de instância.

Dessa forma, a coesão da classe está intimamente relacionada com as responsabilidades que assume. Se uma classe tem muitas responsabilidades, provavelmente terá um conjunto de métodos que pouco se comunicam e usam poucas variáveis em comum, tendo baixa coesão. Quando a coesão é baixa, provavelmente será uma boa ideia criar uma nova classe.

Acoplamento

Outro aspecto muito relevante do posto de vista de orientação a objetos é o acoplamento entre as classes. Essa medida está atrelada a quanto as classes do sistema dependem uma das outras. As dependências podem se expressar de diversas formas desde o simples uso de um método de outra classe até a perigosa manipulação e modificação de dados de outra classe.

Tendo em vista a busca por conseqüências locais e a simplicidade e expressividade do código, não queremos que as classes dependam largamente umas das outras de forma que o entendimento e a modificação de uma leve a alterações em outras classes. Além disso, a cada dependência gerada, os testes unitários criados provavelmente precisaram da utilização de artifícios para isolar cada uma das partes sendo mais difíceis de serem compreendidos.

Abaixo seguem fatores relacionados com o acoplamento que devem ser considerados para o desenvolvimento de um código limpo.

A Lei de Demeter *(The Law of Demeter)*

A Lei de Demeter diz que um método M de uma classe C só deveria chamar um dos seguintes métodos:

- Da própria classe;
- De um objeto criado por M;
- De um objeto passado como argumento para M;
- De um objeto guardado em uma variável de instância de C.

O cumprimento desta heurística visa evitar que uma classe conheça os detalhes de implementação dos objetos aos quais manipula. A invocação de métodos da própria classe não causa nenhum tipo de dependência pela óbvia ausência de algo sobre a qual depender. Os outros tipos de chamadas especificados na Lei causam uma dependência, mas que se limita as interfaces dos objetos trabalhados. A classe deve conhecer quais os métodos dos objetos, os argumentos que aceitam e o tipo do valor de retorno, caso houver.

Um exemplo claro de violação são os chamados *train wrecks*. Suponha que temos uma classe *JogoDeFutebol* e uma classe cliente C executa a seguinte operação:

```
jogoDeFutebol.getTimes().primeiro().getCartoes().count()
```

A chamada para *getTimes()* está de acordo com a Lei de Demeter pois é uma invocação de um método de um objeto guardado em um variável de instância *jogoDeFutebol* da classe *JogoDeFutebol*. O valor de retorno dessa chamada é uma lista contendo os dois times que não é objeto criado e/ou guardado dentro da classe C, nem um argumento passado para M. Dessa forma, a chamada para *primeiro()* sobre essa lista é a primeira das violações à heurística.

Considerando os problemas dessa abordagem e o acoplamento gerado, que partes do sistema a classe C conhece? Primeiramente, conhece a classe *JogoDeFutebol*, já que possui um destes objetos guardados em uma variável de instância e chama um método *getTimes()*. Além disso, conhece o valor de retorno desta chamada e sabe que é uma lista composta por objetos da classe Time e que, portanto, poderia chamar o método *primeiro()* sobre a lista. Ainda conhece que um *Time* tem uma coleção de cartões acessada através de *getCartões()*, que aceita a mensagem *count()*.

Esse exemplo mostra bem que tipo de problemas podem ocorrer quando acoplamento entre as partes do sistema está alta. Claramente, a classe C conhece muitas partes do sistema e, a partir desse momento, qualquer mudança nessa estrutura pode quebrar seu código. Quando temos que criar estruturas como a exemplificada, é necessário repensar o design e nos perguntar porque a classe C precisava ter acesso a todas

aquelas informações. Provavelmente é o caso de fazer mais delegações ao invés de depender tanto dos dados de outros objetos.

Princípio da Inversão de Dependência (Dependency Inversion Principle)

O objetivo da minimização do acoplamento entre nossas classes é facilitar as alterações que serão feitas no sistema. Queremos conseguir estender a infra-estrutura de classes que temos para criar novas funcionalidades.

No exemplo utilizado por Robert Martin em sua coluna para a *The C++ Report* sobre o assunto discutido, temos uma classe *Copiador* que utiliza um *LeitorDeTeclado* para coletar inteiros digitados pelo usuário e um *EscritorDeArquivo* para escrever o dado em um arquivo. Que alterações precisam ser feitas para adicionar a funcionalidade de ler os inteiros de um arquivo? Infelizmente será necessário criar mudanças na classe Copiador para que possa receber dados coletados de dois tipos diferentes de objetos que lidam com a entrada de dados. Tal alteração só se torna necessária devido a dependência que Copiador tem com classes como EscritorDeArquivos e LeitorDoTeclado.

A solução ideal para o problema seria que Copiador dependesse de abstrações como Leitor e Escritor que tivesse uma interface definida de forma que subclasses como LeitorDeTeclado ou LeitorDeArquivo pudessem ser utilizadas polimorficamente. Todos os Leitores teriam que seguir uma abstração ao implementar o método *lerInteiro()* e *Copiador* só precisaria depender dessa interface. Essa mudança seguiria o Princípio da Inversão de Dependência que diz:

- Classes de alto nível não devem depender de classes de baixo nível. Ambas deveriam depender de abstrações.
- Abstrações não devem depender de detalhes. Detalhes deveriam depender de abstrações.

Durante o desenvolvimento, freqüentemente essa solução pode não ser clara ou, em um cenário oposto, os programadores tentam prever todas as partes do código que enfrentam esse problema e utilizam a Inversão de Dependência. Porém, nenhum dos casos pode ser o ideal para o código. Como afirmado por Kent Beck, só devemos investir em flexibilidade em partes do sistema que realmente precisam de flexibilidade. Dessa forma, talvez a melhor solução não precise ser implementada em um primeiro momento, mas quando a necessidade surgir. Provavelmente nesse instante temos que ter certeza que fizemos uma boa escolha para as próximas alterações.

Outro fato importante a ser considerado quanto a Inversão de Dependência está no isolamento dos testes automatizados. Ao testar classes que dependem de outras classes, queremos isolar a testada de forma que os testes sejam de fato unitários. Se a classe em teste depender de abstrações, o trabalho será simplificado. No exemplo do *Copiador* acima, poderíamos criar um *EscritorTeste* e um *LeitorTeste* que tivesse operações convenientes para que os testes de Copiador fosse isolados.

Sessão – Reunindo os Conceitos