**Refactoring - White paper**

**Quando refatorar**

**Regra dos três**

Quando for fazer alguma coisa pela primeira vez, conclua-a. Get it done!

Quando estiver fazendo algo similiar pela segunda vez, repita, mas com um sinal de alerta ligado.

Quando estiver fazendo algo pela terceira vez, é hora de refatorar!

**Quando adicionar uma feature**

A refatoração ajuda a entender o código das outras pessoas. Quando você tem que lidar com o “dirty code” de alguém, tente refatorar antes. Clean code é muito mais fácil de lidar, assim ele não será só mais fácil para o seu desenvolvimento, mas também para os demais que irão mantê-lo futuramente.

E na inclusão de uma nova feature, a refatoração tornará este missão muito mais fácil.

**Quando eliminar um bug**

Bugs em código funcionam basicamente como na vida real: vivem nos mais escuros e sujos lugares do código. Limpe o código e os erros serão praticamente descobertos por si só.

Gestores apreciam refatoração proativa, pois isso elimina a necessidade de criar tarefas específicas para este fim, bem como, para diminuir o débito técnico.

**Durante o code review**

Code review é a última chance de averiguar o código antes de torna-lo público.

Uma das melhores formas de fazê-lo é através das pull request que ferramentas como o Bitbucket permitem. É uma poderosa maneira de corrigir problemas simples rapidamente e agilizar a correção de problemas maiores.

**Quando não refatorar**

1. Quando o código está muito confuso e que a melhor opção seria criá-lo do zero.
2. Quando o código está cheio de falhas. Ao tentar consertar as falhas pode-se gerar ainda mais problemas. A melhor solução neste caso é fazer novamente. Bem feito.
3. Enquanto cria os testes. Antes de refatorar, crie testes sólidos para o trecho a ser alterado, garantindo o resultado esperado. Aí então refatore e verifique nos testes se o resultado não foi alterado

**Como refatorar**

Refatoração deve ser feita em pequenas séries de mudanças, tornando o código cada vez melhor enquanto se realiza o verdadeiro propósito da alteração no código, tais como, o conserto de um bug ou uma nova feature.

**O código precisa se tornar cada vez mais limpo em pequenos passos.**

Se o código refatorado continua tão sujo quanto antes da refatoração, me desculpa, mas você está desperdiçando seu tempo.

Tente descobrir por isso acontece, talvez seja uma ou mais das situações abaixo:

* commits de muitos códigos juntos e enviando uma grande alteração;
* código complexo. Faça simples!
* Não esteja criando testes para cada refatoração;
* Não esteja refatorando os testes;
* Tempo muito limitado para efetuar uma alteração ou refatoração no código que seja consistente.

**Nova funcionalidade não deve ser criada durante a refatoração.**

Não misture o código da refatoração e com o código da nova funcionalidade. Tente, pelo menos, realizar commits atômicos individuais, sendo um para o código refatorado e outro para o novo. (DICA: refatore primeiro, depois adicione a nova feature).

**Todos os testes anteriores devem continuar passando depois da refatoração**.

Há duas situações quando os testes quebram após a refatoração:

* **Você cometeu um erro durante a codificação:** Aqui não tem o que pensar, vai lá e corrija!
* **Os testes estão muito baixo nível:** Por exemplo, há testes unitários para métodos privados. Sendo assim, ou você pode refatorar estes testes ou escrever um novo conjunto de testes mais alto nível. Uma forma de evitar este tipo de situação é escrevendo BDD style-tests.

**Propósito e semântica**

Utilizar bons nomes que demonstrem a intenção do método ou da classe, assim fica fácil identificar qual seria a função e objetivo do mesmo. Com bons nomes não é necessário decifrar o código para ver o que a classe ou método faz, o seu nome já diz, evitando assim escrever qualquer comentário explicativo.

**Baseie-se em métricas de código e qualidade.**

Qualquer percurso a ser trilhado é sempre melhor e mais eficaz quando guiado de forma consistente. E se basear em métricas de código e qualidade trarão luz e direção para o seu caminho.

**NUNCA SE ESQUEÇA: Cada mudança deve vir sempre acompanhada de testes e, sempre que possível, testes automatizados.**

**Técnicas de refatoração**

**Composing Methods**

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand – and even harder to change.

The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

1. Extract Method
2. Inline Method
3. Extract Variable
4. Inline Temp
5. Replace Temp with Query
6. Split Temporary Variable
7. Remove Assignments to Parameters
8. Replace Method with Method Object
9. Substitute Algorithm

**Moving Features between Objects**

Even if you have distributed functionality among different classes in a less-than-perfect way, there is still hope.

These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.

1. Move Method
2. Move Field
3. Extract Class
4. Inline Class
5. Hide Delegate
6. Remove Middle Man
7. Introduce Foreign Method
8. Introduce Local Extension

**Organizing Data**

These refactoring techniques help with data handling, replacing primitives with rich class functionality. Another important result is untangling of class associations, which makes classes more portable and reusable.

1. Change Value to Reference
2. Change Reference to Value
3. Duplicate Observed Data
4. Self-Encapsulate Field
5. Replace Data Value with Object
6. Replace Array with Object
7. Change Unidirectional Association to Bidirectional
8. Change Bidirectional Association to Unidirectional
9. Encapsulate Field
10. Encapsulate Collection
11. Replace Magic Number with Symbolic Constant
12. Replace Type Code with Class
13. Replace Type Code with Subclasses
14. Replace Type Code with State/Strategy
15. Replace Subclass with Fields

**Simplifying Conditional Expressions**

Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.

1. Consolidate Conditional Expression
2. Consolidate Duplicate Conditional Fragments
3. Decompose Conditional
4. Replace Conditional with Polymorphism
5. Remove Control Flag
6. Replace Nested Conditional with Guard Clauses
7. Introduce Null Object
8. Introduce Assertion

**Simplifying Method Calls**

These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.

1. Add Parameter
2. Remove Parameter
3. Rename Method
4. Separate Query from Modifier
5. Parameterize Method
6. Introduce Parameter Object
7. Preserve Whole Object
8. Remove Setting Method
9. Replace Parameter with Explicit Methods
10. Replace Parameter with Method Call
11. Hide Method
12. Replace Constructor with Factory Method
13. Replace Error Code with Exception
14. Replace Exception with Test

**Dealing with Generalization**

Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.

1. Pull Up Field
2. Pull Up Method
3. Pull Up Constructor Body
4. Push Down Field
5. Push Down Method
6. Extract Subclass
7. Extract Superclass
8. Extract Interface
9. Collapse Hierarchy
10. Form Template Method
11. Replace Inheritance with Delegation
12. Replace Delegation with Inheritance