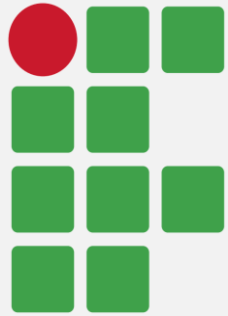
ENGENHARIA DE SOFTWARE

Refatoração, TDD e Código Legado

Prof. Giovani Fonseca Ravagnani Disperati
IFSP – Câmpus Guarulhos

CONTEÚDO PROGRAMÁTICO



- Desenvolvimento dirigido a testes
- Refatoração
- Código legado
- Débito técnico
- Smells de design

Introdução

- Conforme mencionamos anteriormente, a sequência de disciplinas práticas do XP, em particular o Desenvolvimento Dirigido a Testes e a Refatoração, são ferramentas importantes para que possamos abraçar os valores ágeis

- Refatoração e TDD são disciplinas técnicas complicadas de se dominar, porém que resultam em uma série de benefícios no longo prazo

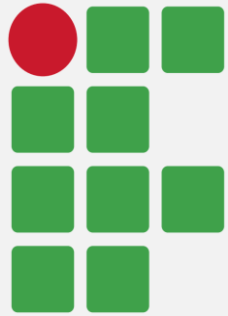
DESENVOLVIMENTO DIRIGIDO A TESTES

TDD

- O que é TDD?

- O desenvolvimento orientado a testes (TDD) é um processo de desenvolvimento de software que depende da repetição de um ciclo de desenvolvimento muito curto: os requisitos são transformados em casos de teste muito específicos e, em seguida, o software é construído para que os testes passem;

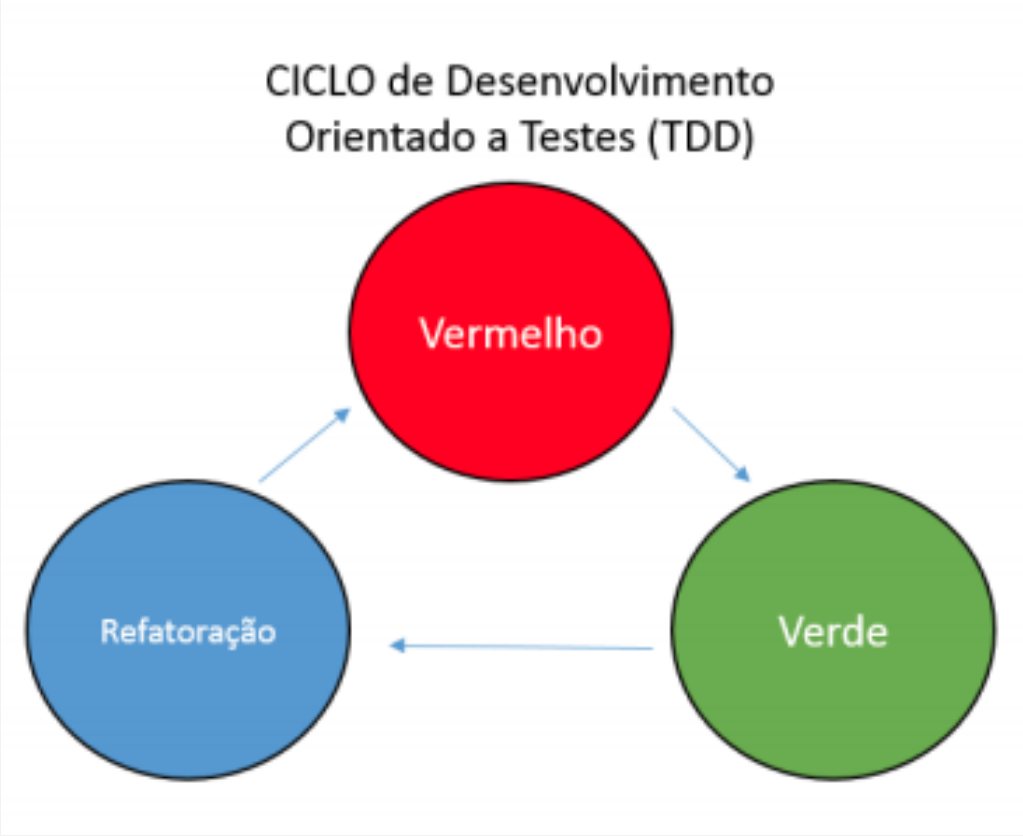
TDD



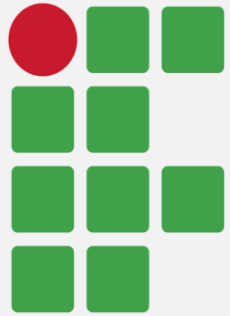
- O engenheiro de software norte-americano Kent Beck, a quem se credita ter desenvolvido ou "redescoberto" a técnica, afirmou em 2003 que TDD encoraja designs simples e inspira confiança.

TDD

- O ciclo de desenvolvimento orientado a testes é sintetizado na Figura abaixo

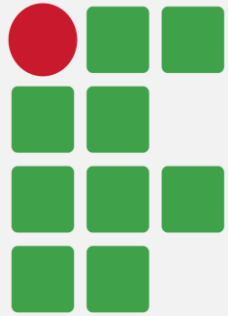


TDD



- Um ciclo comum de desenvolvimento com TDD é representado da seguinte forma
  - O programador escreve um número (muito) pequeno de casos de testes automatizados;
  - Executa-se, então, o novo caso de teste para garantir que ele falhe (pois não há código para ser executado ainda);
  - O código que faz com que o novo teste passe é implementado;

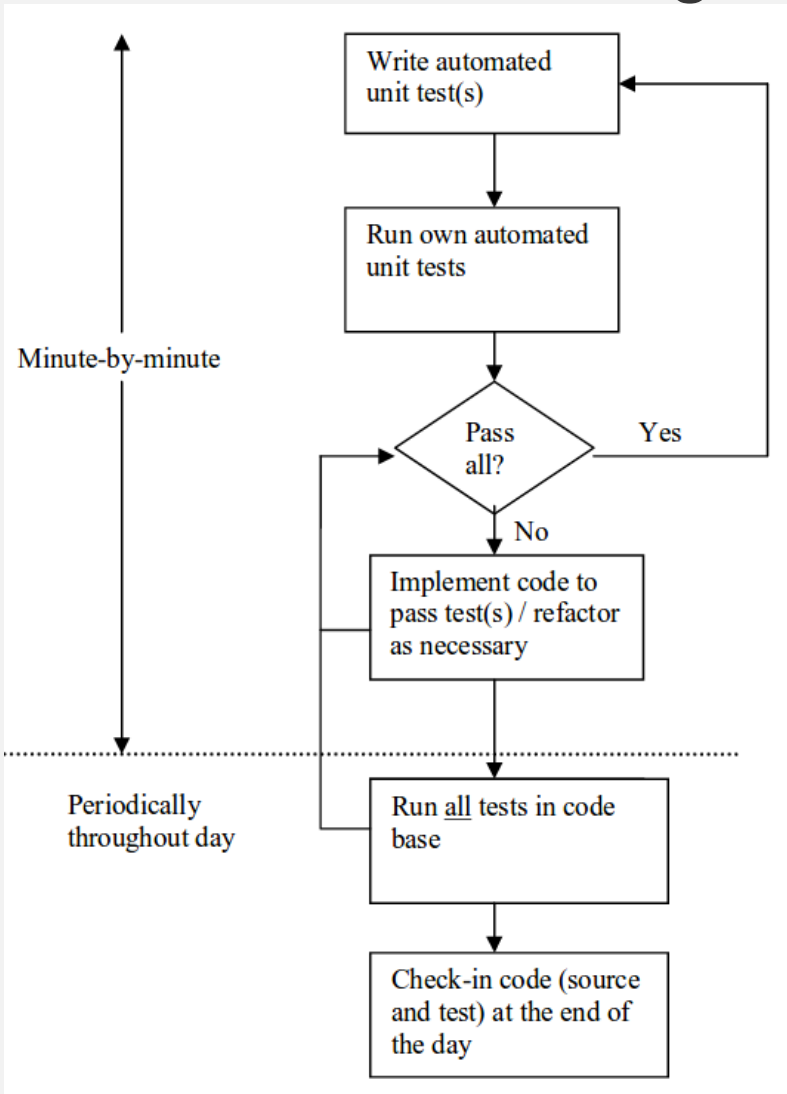
TDD



- O teste é re-executado, de modo a garantir que ele passa com o novo código implementado;
- Com o teste passando, se refatora o código de implementação ou de teste, como necessário, e
- Periodicamente (de preferência, uma vez por dia ou mais) todos os casos de teste na base de código são reexecutados para garantir que o novo código não quebra nenhum caso de teste que já foi executado.

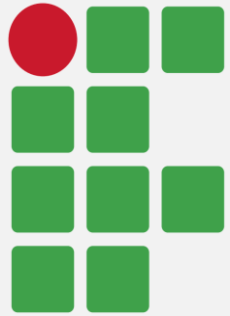
TDD

- Esse processo pode ser visto no diagrama



*Bhat, Thirumalesh; Nagappan, Nachiappan. Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. International Symposium on Empirical Software Engineering (ISESE 2006), September 21-22, 2006, Rio de Janeiro, Brazil

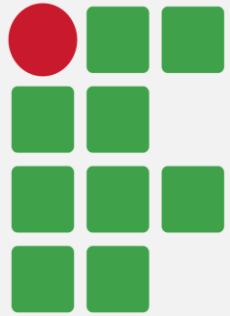
TDD



- Alguns dos benefícios reportados a partir do uso de TDD são

- Eficiência e Feedback:
  - A granularidade do ciclo de “test-then-code” fornece feedback contínuo ao desenvolvedor;
- Design de baixo nível:
  - Os testes fornecem uma especificação das decisões de design de baixo nível em termos de classes, métodos e interfaces criadas.

TDD



- Alguns dos benefícios reportados a partir do uso de TDD são

- Eficiência e Feedback:
  - A granularidade do ciclo de “test-then-code” fornece feedback contínuo ao desenvolvedor;

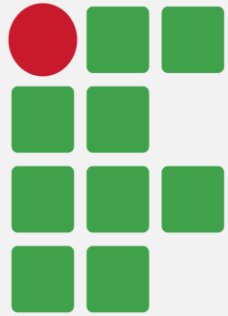
- Design de baixo nível:
  - Os testes fornecem uma especificação das decisões de design de baixo nível em termos de classes, métodos e interfaces criadas.

TDD

- Redução da Injeção de defeitos
  - Muitas vezes a partir da depuração e manutenção de software, o código de produção é "patcheado" para alterar sua propriedades e especificações e, por vezes, os designs não são examinados nem atualizados.
  - Infelizmente tais correções e “pequenas" mudanças de código podem ser quase 40 vezes mais propensas a erros do que novos desenvolvimento*. Ao executar continuamente estes casos de teste automatizados, pode-se descobrir se uma mudança quebra o sistema existente.

*W. S. Humphrey, Managing the Software Process. Reading, Massachusetts: Addison-Wesley, 1989.
**L. Williams, E. M. Maximilien, and M. Vouk, “Test-Driven Development as a Defect-Reduction Practice”, Proceedings of IEEE International Symposium on Software Reliability Engineering, Denver, CO, pp. 34-45, 2003.

TDD



- Os tipos mais comuns de testes são

- Unitários - Testes unitários dizem respeito ao comportamento de uma funcionalidade específica. Testes unitários tendem a ser os mais rápidos.

- De integração / Funcionais - testes funcionais realizam interação entre as diversas áreas do sistema. Testes funcionais tendem a ser mais lentos que os testes unitários.

- De sistema / ponta a ponta: Testes end to end demonstram o funcionamento de uma parte significativa do sistema

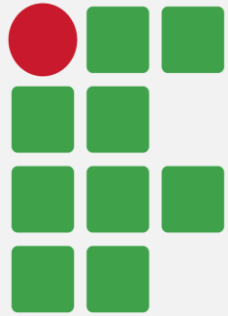
TDD

- Um estudo empírico de um ano realizado na IBM* usando programadores profissionais verificou que a prática de TDD auxiliou os programadores a produzirem um código de maior qualidade.

- O time da IBM produziu um conjunto completo de casos de teste automatizados após o design UML.

L. Williams, E. M. Maximilien, and M. Vouk, “Test-Driven Development as a Defect-Reduction Practice”, Proceedings of IEEE International Symposium on Software Reliability Engineering, Denver, CO, pp. 34-45, 2003.

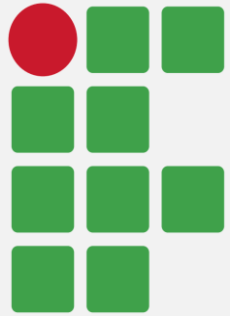
TDD



- O código desenvolvido com a prática TDD mostrou, a partir de testes de verificação e regressão, aproximadamente 40% menos defeitos do que um produto anterior desenvolvido de forma mais tradicional.

- A produtividade da equipe não foi afetada pela foco adicional na produção de casos de teste automatizados.

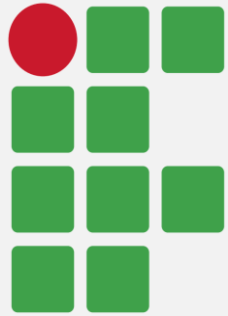
TDD



- Uma experiência envolvendo 24 programadores profissionais da John Deere, Rolemodel Software e Ericsson foi realizado para investigar a eficácia do TDD.

- Um grupo desenvolveu um programa Java usando TDD enquanto o outro grupo de controle usou uma abordagem cascata.

TDD



- Os resultados indicaram que os programadores TDD produziram código de maior qualidade, porque passaram em 18% mais casos de testes funcionais de caixa preta.

- No entanto, os programadores TDD levaram 16% mais tempo. Contudo, os programadores no grupo de controle cascata não escreveram casos de teste automatizados após completar seu código.

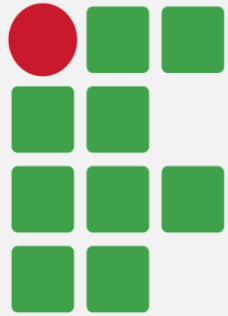
REFATORAÇÃO

Refatoração

- Refatoração
  - ...é uma técnica disciplinada para reestruturar um corpo de código existente, alterando sua estrutura interna sem alterar seu comportamento externo.

- O coração da refatoração consiste de uma série de pequenas transformações, preservando-se o comportamento. Cada transformação (chamada de "refatoração") faz pouco, mas uma sequência de transformações pode produzir uma reestruturação significativa.

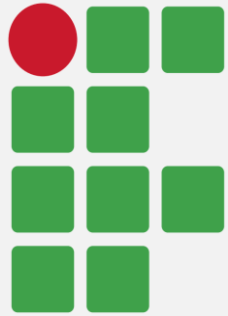
Refatoração



- Uma vez que cada refatoração é pequena, é menos provável que ela dê errado.

- O sistema é mantido funcionando completamente após cada pequena refatoração, reduzindo as chances de que um sistema possa ser gravemente quebrado durante a reestruturação.

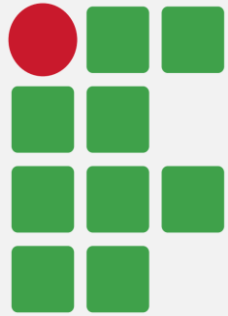
Refatoração



- Martin Fowler, em seu livro Refactoring, afirma

  - “I've been asked, "Is refactoring just cleaning up code?" In a way the answer is yes, but I think refactoring goes further because it provides a technique for cleaning up code in a more efficient and controlled manner. Since I've been using refactoring, I've noticed that I clean code far more effectively than I did before. This is because I know which refactorings to use, I know how to use them in a manner that minimizes bugs, and I test at every possible opportunity.” (Fowler, 2004. p. 46)

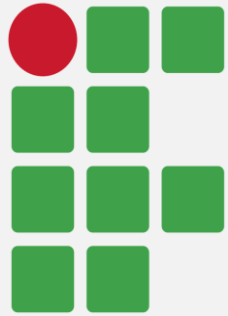
Refatoração



- Refatoração é uma ferramenta que pode ser usada para diversas finalidades;

- Refatorar tende a melhorar o design do código: sem refatoração, o design do programa irá decair.

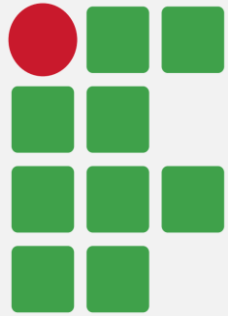
Refatoração



- Conforme as pessoas mudam o código, este tende a perder sua estrutura.

- A perda de estrutura do código tem efeito cumulativo: quanto mais difícil visualizar a estrutura, mais difícil mantê-la. Um design ruim usualmente toma mais tempo para manutenção do que um bom design.

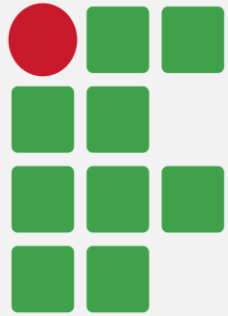
Refatoração



- Refatorar torna o software mais simples de entender: programar é como “conversar” com o computador. O problema é que, em um primeiro momento, não costumamos pensar em outros desenvolvedores.

- Refatorar nos ajuda a melhorar a legibilidade do código e aprimorar nosso entendimento do que ocorre.

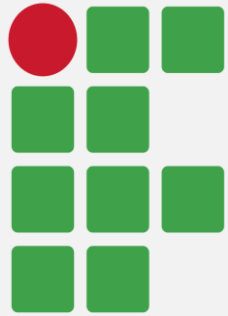
Refatoração



- Refatoração te ajuda a encontrar bugs: ao aprimorar seu entendimento sobre o código, torna-se mais fácil identificar bugs.
  - Kent Beck, "I'm not a great programmer; I'm just a good programmer with great habits."

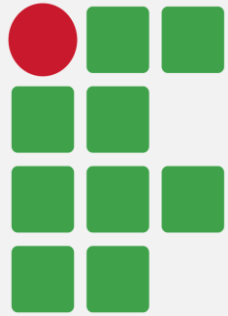
- Refatorar te ajuda a programar mais rápido - apesar disso soar contraintuitivo

Refatoração



- Fowler afirma que
  - I strongly believe that a good design is essential for rapid software development. Indeed, the whole point of having a good design is to allow rapid development. Without a good design, you can progress quickly for a while, but soon the poor design starts to slow you down. You spend time finding and fixing bugs instead of adding new function. Changes take longer as you try to understand the system and find the duplicate code. New features need more coding as you patch over a patch that patches a patch on the original code base. A good design is essential to maintaining speed in software development. Refactoring helps you develop software more rapidly, because it stops the design of the system from decaying. It can even improve a design. (Fowler, 2004. p. 49)

Refatoração



- Para Fowler, devemos refatorar na terceira vez que fazemos algo similar. “Three strikes and you refactor.”.

- Refatorar quando você adiciona uma função;

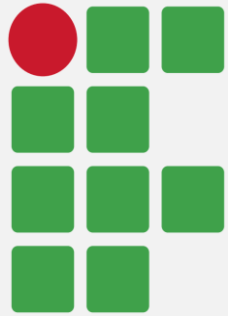
- Refatorar quando você precisa arrumar um bug;

- Refatorar quando você faz revisão de código;

Refatoração

- De acordo com Kent Beck
  - Programs have two kinds of value: what they can do for you today and what they can do for you tomorrow. Most times when we are programming, we are focused on what we want the program to do today. Whether we are fixing a bug or adding a feature, we are making today's program more valuable by making it more capable. You can't program long without realizing that what the system does today is only a part of the story. If you can get today's work done today, but you do it in such a way that you can't possibly get tomorrow's work done tomorrow, then you lose. Notice, though, that you know what you need to do today, but you're not quite sure about tomorrow.

Refatoração



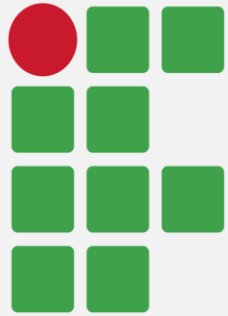
- Ademais, Kent Beck afirma que
  - Maybe you'll do this, maybe that, maybe something you haven't imagined yet. I know enough to do today's work. I don't know enough to do tomorrow's. But if I only work for today, I won't be able to work tomorrow at all. Refactoring is one way out of the bind. When you find that yesterday's decision doesn't make sense today, you change the decision. Now you can do today's work. Tomorrow, some of your understanding as of today will seem naive, so you'll change that, too. What is it that makes programs hard to work with? Four things I can think of as I am typing this are as follows: Programs that are hard to read are hard to modify; Programs that have duplicated logic are hard to modify.

Refatoração

- E ainda de acordo com Kent Beck
  - Programs that require additional behavior that requires you to change running code are hard to modify; Programs with complex conditional logic are hard to modify; So, we want programs that are easy to read, that have all logic specified in one and only one place, that do not allow changes to endanger existing behavior, and that allow conditional logic to be expressed as simply as possible. Refactoring is the process of taking a running program and adding to its value, not by changing its behavior but by giving it more of these qualities that enable us to continue developing at speed.

CÓDIGO LEGADO

Trabalhando com código legado



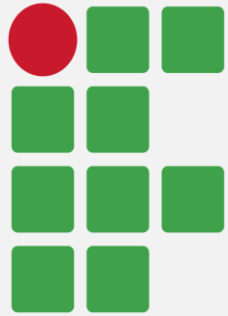
- Sistemas de software tendem a se degradar

- O que começa como um design cristalino nas mentes dos programadores *apodrece*, ao longo do tempo.

-

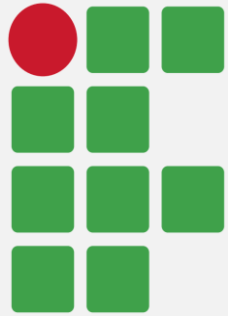
- O belo sistema que construímos no ano passado se transforma em um pântano horrível de funções e variáveis emaranhadas no próximo ano.

Trabalhando com código legado



- Por qual motivo isso acontece?
  - Usualmente a degradação do código fonte se dá por mudanças de requisitos de formas não previstas pelo design inicial.

Código legado

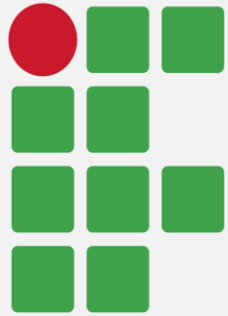


- Para Michael Feathers, código legado é simplesmente um código sem testes.

- O que os testes têm a ver com o código ser ruim?
  - Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.

DÉBITO TÉCNICO

Débito técnico



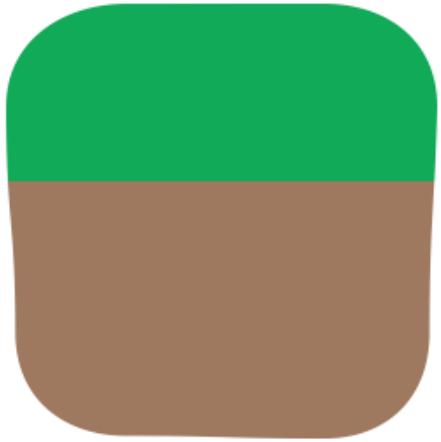
- Débito técnico é uma metáfora, cunhada por Ward Cunningham, que descreve como pensar em lidar com esse *cruft*, pensando nele como uma dívida financeira. O esforço extra necessário para adicionar novos recursos é o juro pago sobre a dívida.

https://dev.to/pullrequest/the-85-billion-cost-of-bad-code-3dmc
https://www.cnbc.com/2018/09/06/companies-worry-more-about-access-to-software-developers-than-capital.html

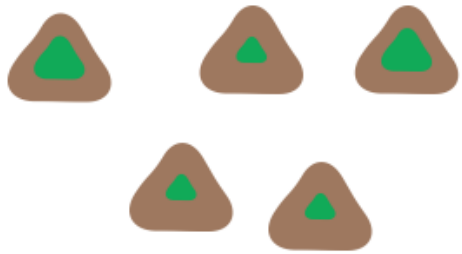
Débito técnico

*Any software system has a certain amount of **essential** complexity required to do its job...*

*... but most systems contain **cruft** that makes it harder to understand.*

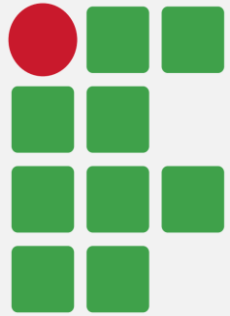


*Cruft causes changes to take **more effort***



The technical debt metaphor treats the cruft as a debt, whose interest payments are the extra effort these changes require.

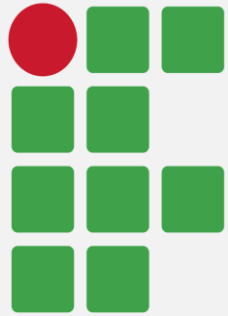
Débito técnico



- Imagine que eu tenho uma estrutura de módulo confusa na minha base de código. Eu preciso adicionar um novo recurso;

- Se a estrutura do módulo fosse clara, levaria quatro dias para adicionar o recurso. com esse *cruft*, entretanto, levaria seis dias. A diferença de dois dias é o juro da dívida;

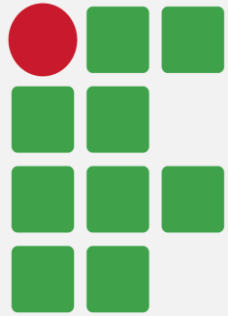
Débito técnico



- Eu poderia levar cinco dias para limpar a estrutura modular, removendo o cruft, pagando metaforicamente o principal da dívida;

- Se eu fizer isso apenas para esse recurso, não há ganho, pois levaria nove dias em vez de seis. Mas se eu tiver mais dois recursos semelhantes chegando, acabarei o serviço de forma mais rápida removendo o *cruft* primeiro;

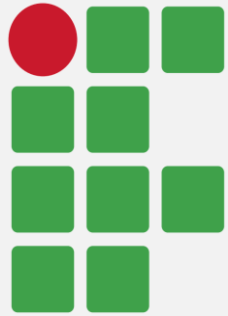
Débito técnico



- Dito isso, parece uma simples questão de trabalhar os números, e qualquer gerente com uma planilha deve descobrir as opções;

- Infelizmente, como é muito difícil medir produtividade, nenhum desses custos costuma ser objetivamente mensurável.

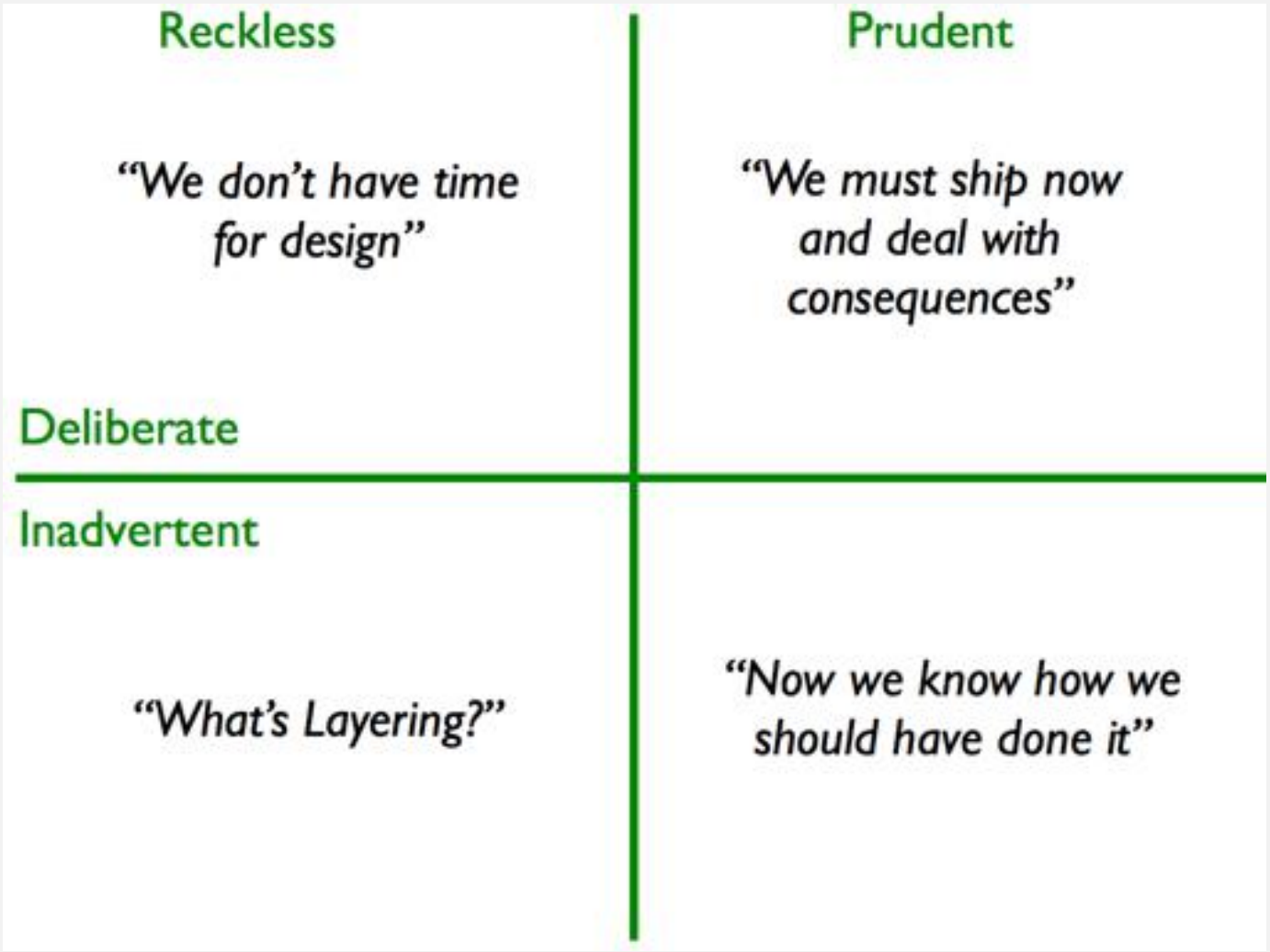
Débito técnico



- Podemos estimar quanto tempo leva para executar um recurso, estimar como seria se o cruft foi removido e estimar o custo de remoção do cruft.

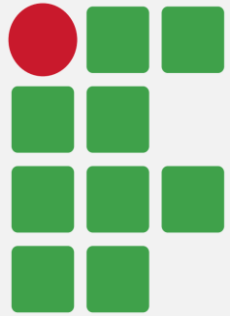
- Entretanto, nossa precisão para tais estimativas é usualmente baixa.

Débito técnico



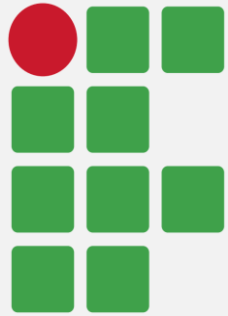
DESIGN SMELLS

Design smells



- Bob Martin aponta os seguintes Smells de Design
  - Rigidez
  - Fragilidade
  - Imobilidade
  - Viscosidade
  - Complexidade desnecessária
  - Repetição desnecessária
  - Opacidade

Rigidez



- Rigidez é a tendência do software ser difícil de mudar, mesmo de maneiras simples.
    - Um design é rígido se uma única alteração causa uma cascata de alterações subsequentes nos módulos dependentes.
    - Quanto mais módulos devem ser alterado, mais rígido o design.

Rigidez

- A maioria dos desenvolvedores já enfrentou essa situação de uma maneira ou de outra

- Eles são convidados a fazer o que parece ser uma mudança simples, examinam a mudança e fazem uma estimativa razoável do trabalho necessário

Rigidez

- Porém, mais tarde, ao trabalharem com a mudança, descobrem que há repercussões imprevistas à mudança.

- Os desenvolvedores se veem perseguindo a mudança através de enormes partes do código, modificando muito mais módulos do que haviam estimado inicialmente e descobrindo outras alterações que eles devem se lembrar de fazer.

Rigidez

- No final, as mudanças levar muito mais tempo que a estimativa inicial.

- Quando perguntados por que sua estimativa era tão ruim, eles repetem o lamento tradicional dos desenvolvedores de software: "Foi muito mais complicado do que eu pensava!”

Fragilidade

- Fragilidade é a tendência do Software quebrar em muitos locais quando uma única mudança é feita; frequentemente, os novos problemas se dão em áreas sem relação conceitual com a área que foi modificada.

- O conserto destes problemas subsequentes conduz à mais problemas e o comportamento do time de desenvolvimento passa a lembrar o de um cachorro correndo atrás do próprio rabo.

Fragilidade

- Conforme a fragilidade aumenta, a probabilidade de que uma mudança introduza problemas inesperados aproxima-se cada vez mais da certeza;

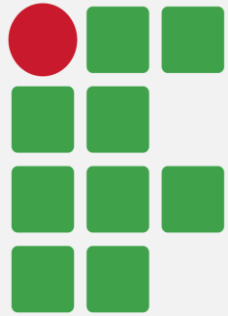
- Isso soa absurdo, mas a existência deste tipo de problema não é incomum;

Fragilidade

- Normalmente isso se dá nos módulos que estão em contínua necessidade de reparo, que nunca saem das listas de bugs.

- Usualmente os desenvolvedores sabem que tais módulos precisam de um redesign - são os módulos que ficam piores quanto mais você os conserta.

Imobilidade



- Um design é imóvel quando sua base de código contém diversas partes que poderiam ser úteis em outros sistemas, mas o risco e esforço envolvido na extração destas partes são grandes demais.

- Isso é lamentável, porém bastante comum.

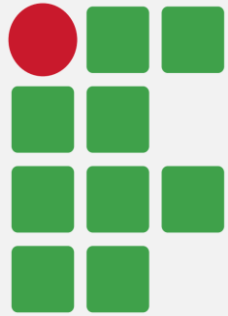
Viscosidade

- Viscosidade vem em duas formas: viscosidade de software e viscosidade de ambiente.

- Quando confrontados com uma mudança, usualmente os desenvolvedores encontram mais de uma forma de implementá-la. Algumas das formas preservam o design; outras não preservam - são hacks.

- Quando as formas de implementação que preservam o design são mais difíceis do que os hacks, a viscosidade do design é alta. Ou seja, é fácil fazer a coisa errada mas difícil fazer a coisa certa.

Viscosidade



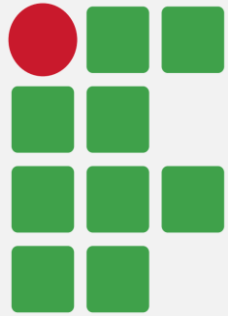
- Já a viscosidade do ambiente se mostra quando o ambiente de desenvolvimento é lento e ineficiente.
  - Por exemplo, se os tempos de compilação são excessivamente longos, os desenvolvedores tenderão a fazer mudanças que não forcem grandes recompilações - ainda que tais mudanças não preservem o design
- Em ambos os casos, um projeto viscoso é um em que o design do software é difícil de preservar.

Complexidade desnecessária

- Um sistema desnecessariamente complexo é quando este contém elementos que não são atualmente necessários.

- Isso frequentemente ocorre quando os desenvolvedores antecipam mudanças nos requisitos e introduzem recursos no software para lidar com estas potenciais mudanças.

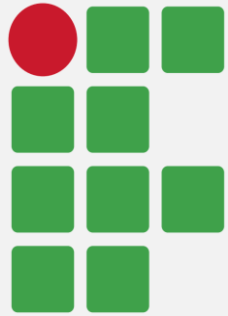
Complexidade desnecessária



- Em um primeiro momento, isto pode parecer algo positivo a ser feito.

- Afinal de contas, se preparar para mudanças futuras mantém nosso código flexível e previne mudanças assustadoras posteriormente.

Complexidade desnecessária



- Infelizmente, o efeito é normalmente o oposto. Por se preparar para muitas contingências, o design se torna desarrumado com construções que nunca são utilizadas. Alguns dos preparos podem se pagar no futuro, mas muitos outros não.

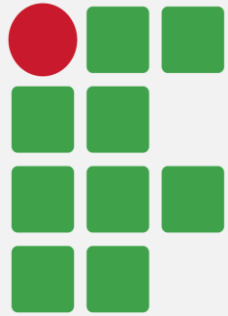
- Enquanto isso, o software carrega o peso desses elementos de design não utilizados, tornando-o complexo e de difícil entendimento.

Repetição desnecessária

- “Copiar e colar” é uma operação desastrosa quando se trata de criação de código fonte; Não é incomum sistemas de software construídos sob centenas de elementos de códigos repetidos.

- Normalmente acontece da seguinte forma: Ralph precisa de um código para realizar operação X; ele procura no base código locais onde acredita que operação semelhante já foi feita; ele encontra tal código, copia sua estrutura e faz as alterações necessárias para que ele funcione

Repetição desnecessária



- Mal suspeita Ralph que o código que ele copiou foi escrito por Todd, que o encontrou em um módulo escrito por Lilly

- Lilly foi a primeira a escrever a operação X, mas reparou que ela era muito parecida com a operação Y; então ela encontrou operação Y em outro módulo, a copiou e fez as alterações necessárias.

- Quando o mesmo código aparece diversas vezes de formas levemente distintas os desenvolvedores não estão percebendo uma abstração. Código redundante torna o trabalho de manutenção terrivelmente árduo.

Opacidade

- Opacidade é a tendência do software ser de difícil compreensão.

- Código pode ser escrito de maneira clara e expressiva ou pode ser escrito de forma “opaca” e convoluta. Quando os desenvolvedores escrevem um módulo pela primeira vez, o código pode parecer claro para eles. Mais tarde eles podem voltar ao módulo e se perguntar como poderiam ter escrito algo tão terrível.

- Para evitar isso, os desenvolvedores precisam se colocar no lugar de seus leitores e fazer um esforço conjunto refatorar seu código para que seus leitores possam entendê-lo.

Referências

- Brutal Refactoring: More Working Effectively with Legacy Code
- Working Effectively with Legacy Code