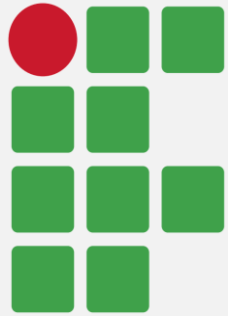
ENGENHARIA DE SOFTWARE

Clean Code

Prof. Giovani Fonseca Ravagnani Disperati
IFSP – Câmpus Guarulhos

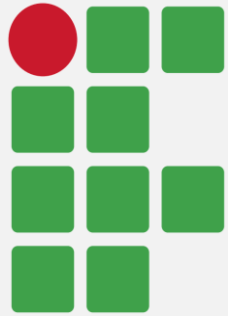
CONTEÚDO PROGRAMÁTICO



- Fundamentos do Clean Code
- Nomes significativos
- Funções / Métodos
- Comentários
- Forma
- Objetos e estruturas de dados
- Gerenciamento de erros
- Limites
- Testes unitários
- Classes
- Sistemas
- Emergência de design

FUNDAMENTOS DO CLEAN CODE

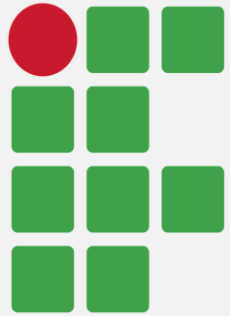
Fundamentos do Clean Code



- Nós sempre desenvolvemos código em baixo nível, porque todos os detalhes são importantes.

- Código bom e limpo importa! Código ruim eventualmente arruína um produto, porque durante o desenvolvimento a produtividade aproxima-se gradualmente de zero.

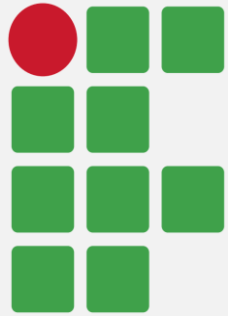
Fundamentos do Clean Code



- Os programadores devem defender o código limpo, assim como os gerentes defendem os requisitos e prazos.

- Mas os gerentes dependem dos programadores, não vice-versa: a fim de ir rápido, devemos ter um código limpo.

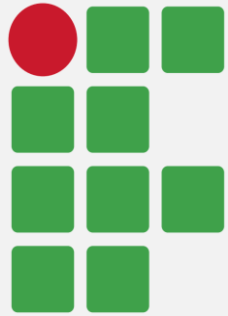
Fundamentos do Clean Code



- Se você é programador há algum tempo, provavelmente já foi significativamente desacelerado pelo código bagunçado de alguém
    - O grau de desaceleração pode ser alarmante!

- Ao decorrer de um ou dois anos, times que estavam indo rápido no começo podem se encontrar em ritmo extremamente lento;

Fundamentos do Clean Code



- Toda mudança feita no sistema quebra alguma parte inesperada;

- Cada modificação no sistema requer o entendimento de “hacks” no código, a fim de que se possa gerar mais “hacks”

Fundamentos do Clean Code

- O gráfico abaixo (Clean Code, p. 04) mostra a situação descrita

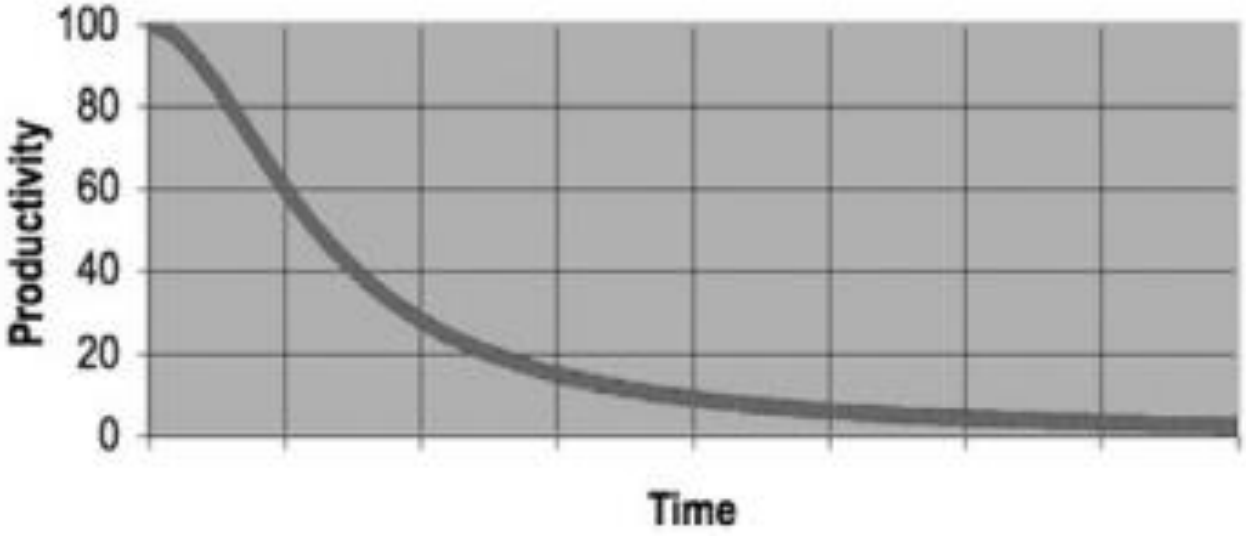
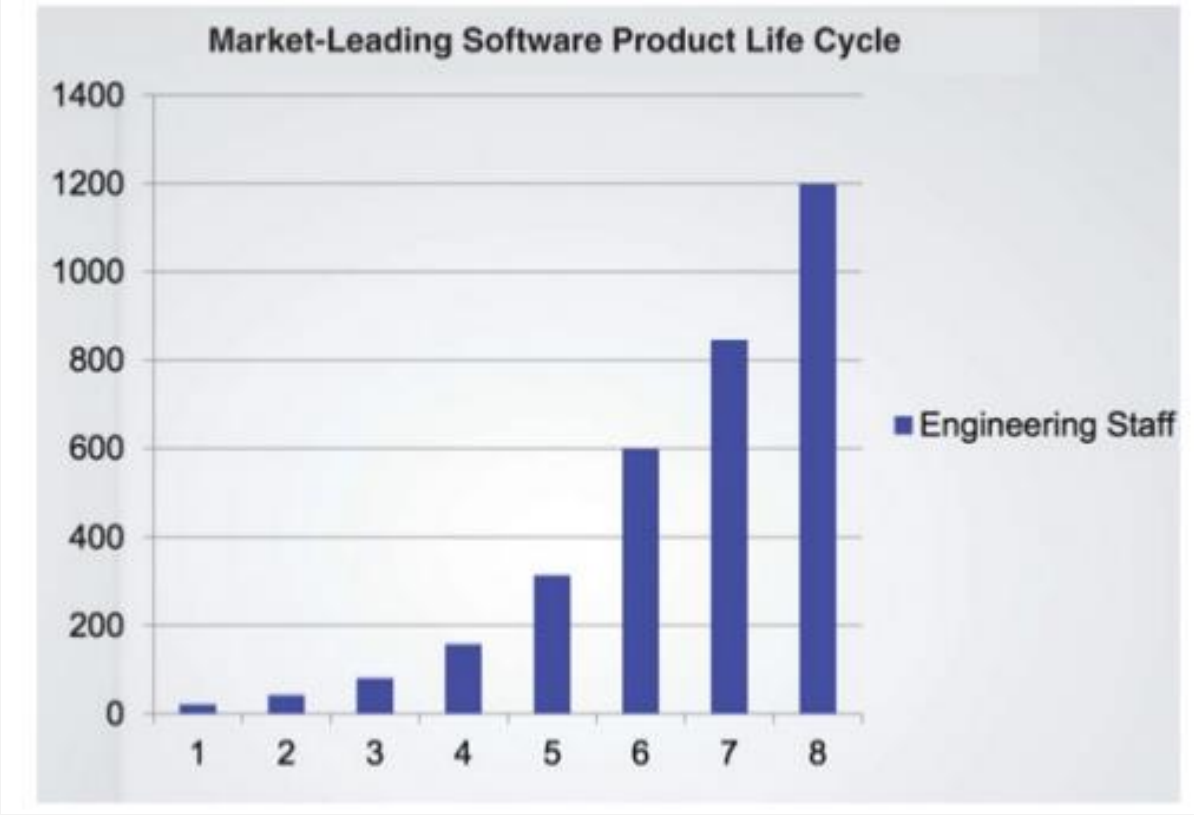


Figure 1-1
Productivity vs. time

Fundamentos do Clean Code

- Em seu livro “Clean Architecture”, o autor Robert C. Martin mostra seguinte estudo de caso. O gráfico abaixo mostra a evolução do número de funcionários a cada release

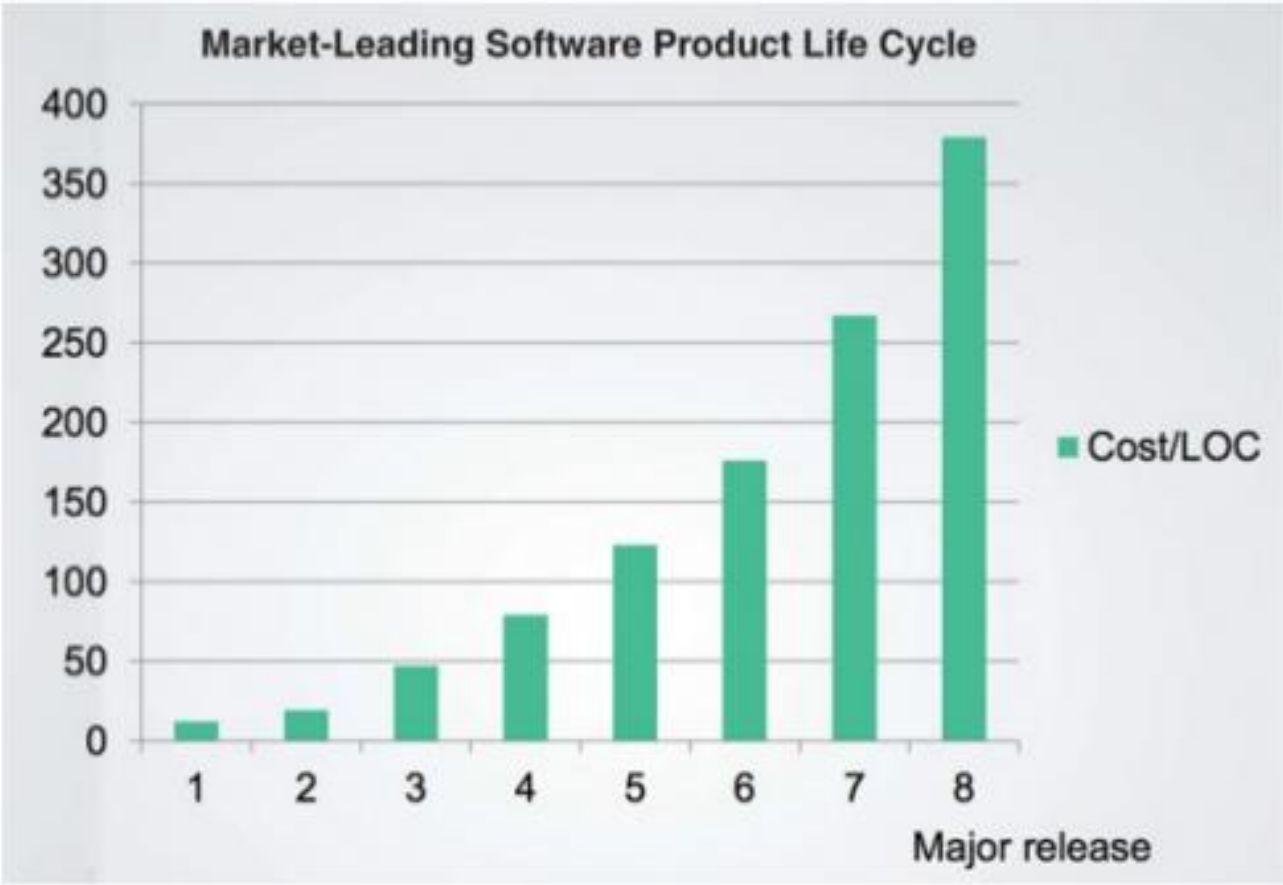


Fundamentos do Clean Code

- O gráfico abaixo mostra a evolução do sistema medida em linhas de código

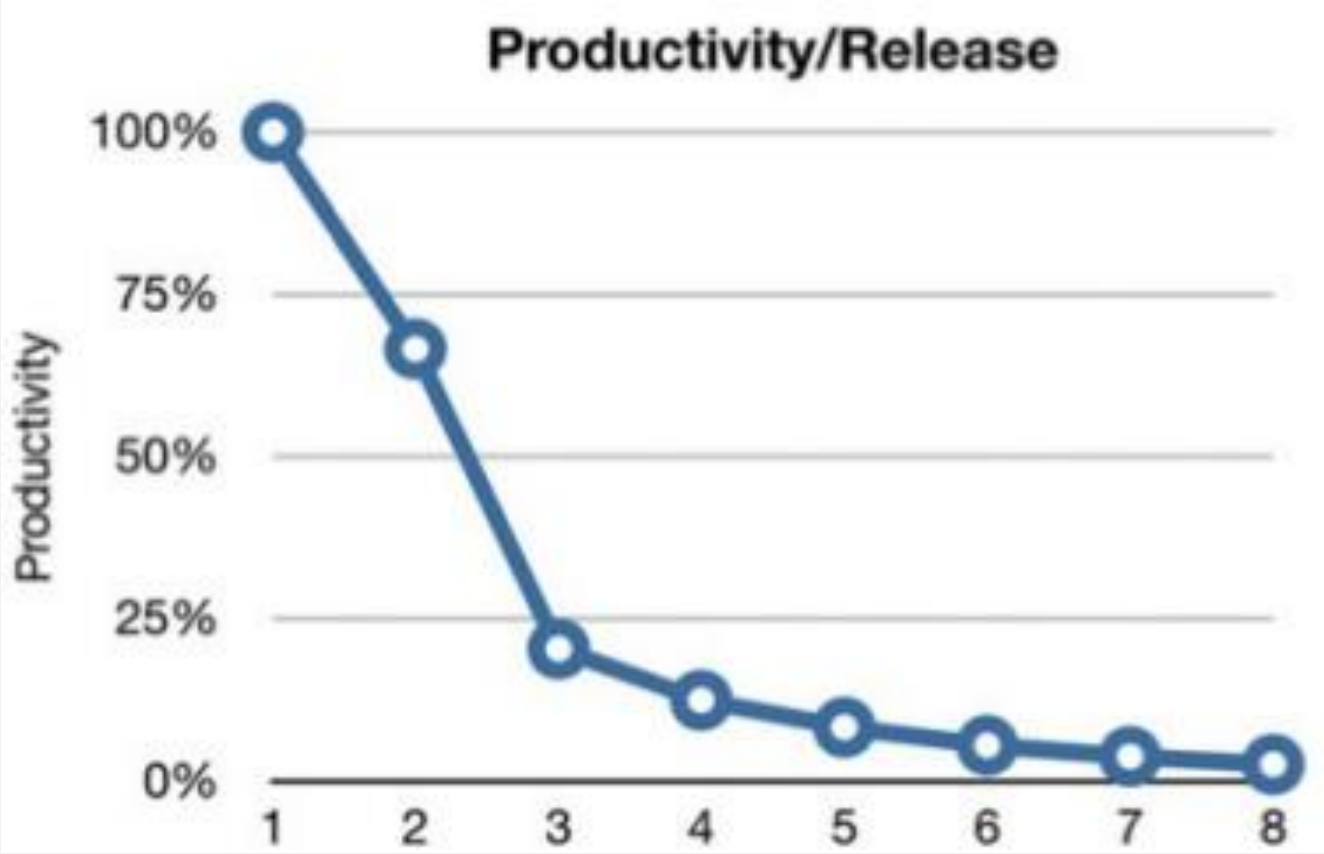
Fundamentos do Clean Code

- O seguinte gráfico mostra o custo por linhas de código



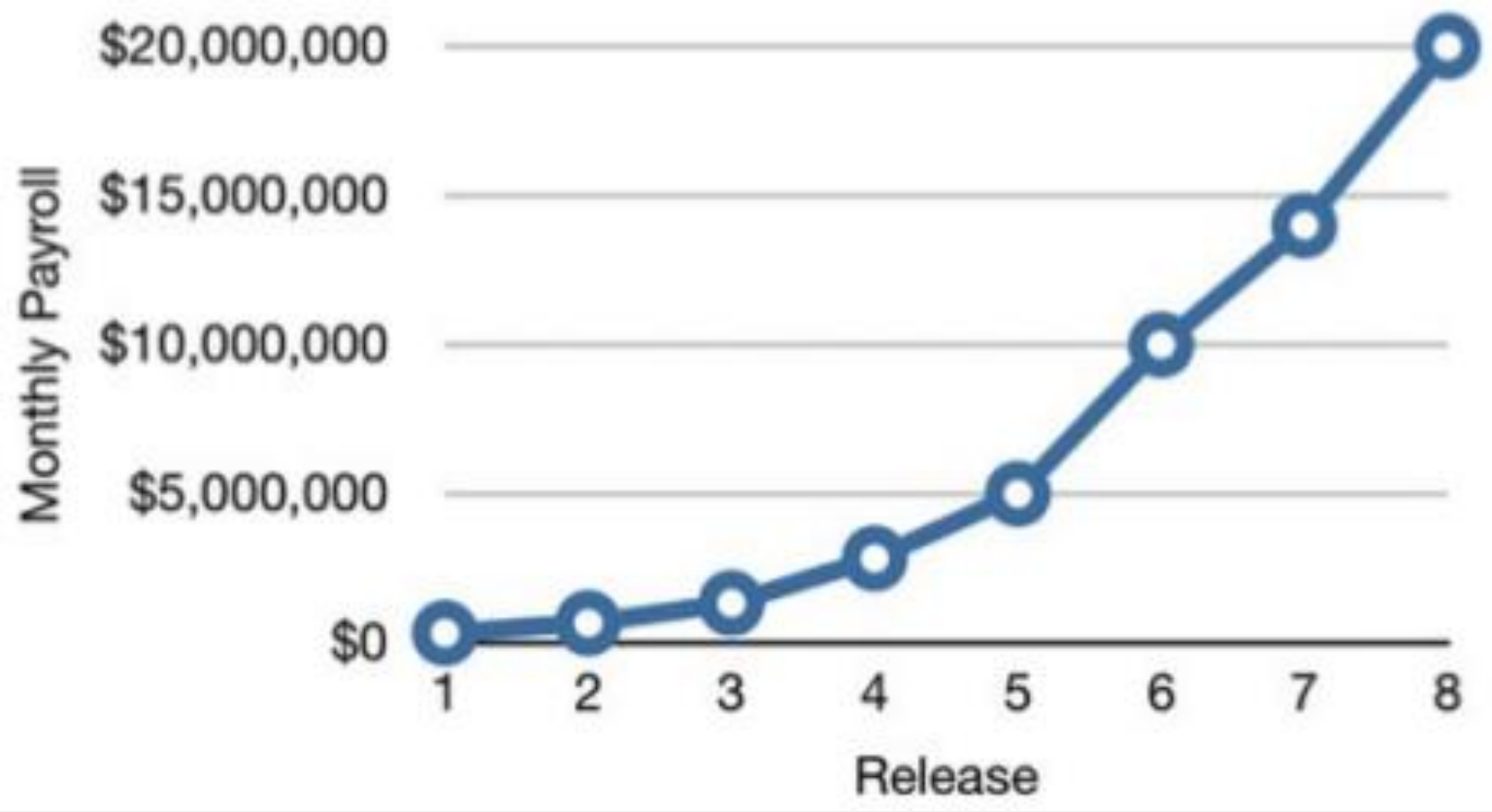
Fundamentos do Clean Code

- O gráfico abaixo mostra a produtividade por release

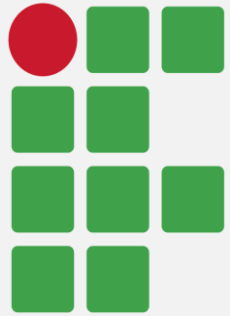


Fundamentos do Clean Code

- E este gráfico mostra a evolução da folha de pagamento



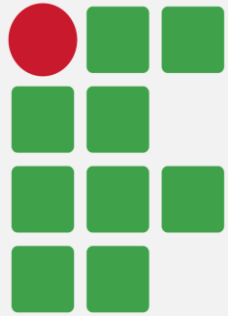
Fundamentos do Clean Code



- Em 1951, uma filosofia de qualidade chamada “Total Productive Maintenance” (TPM) surgiu na cena automobilística japonesa.

- Um dos principais pilares do TPM era seu foco em manutenção ao invés da produção, através do princípios chamados “5S”;

Fundamentos do Clean Code

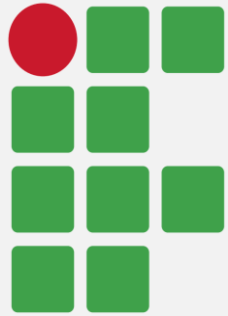


- Estes princípios são:

- Seiri, ou organização: saber onde as coisas estão é crucial; identificadores como nomes significativos desempenham um importante papel;

- Seiton, ou sistematização: cada coisa exatamente em seu lugar. Um trecho de código deve estar onde for mais óbvio e, e caso não esteja, é necessário refatorar o código para colocá-lo lá;

Fundamentos do Clean Code

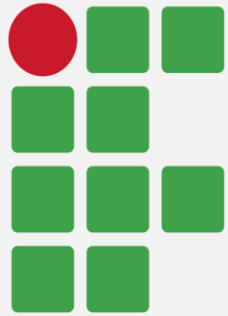


- Seiso, ou limpeza: o código deve ser limpo de comentários inúteis, código comentado e todo tipo de “lixo” que o esteja poluindo;

- Seiketsu, ou padronização: o grupo deve concordar em como manter o ambiente limpo;

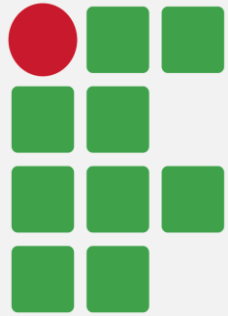
- Shutsuke, ou disciplina (auto-disciplina): é necessário ter disciplina para seguir as práticas acordadas e estar disposto a constantemente refletir e melhorar o trabalho;

Fundamentos do Clean Code



- Definições de código limpo por Bjarne Stroustrup (C++), Grady Booch (UML), Dave Thomas, Michael Feathers, Ron Jeffries (XP), Ward Cunningham (XP, Wiki, Design Patterns):
  - O código limpo é elegante, simples, eficiente, direto, nítido, claro, legível por outros, não surpreendente, tem dependências mínimas e explícitas, testes automatizados, minimiza o número de classes e métodos, expressa suas ideias de design, trata erros, não tem nada de óbvio que se possa fazer para melhorá-lo: parece que o autor teve cuidado.

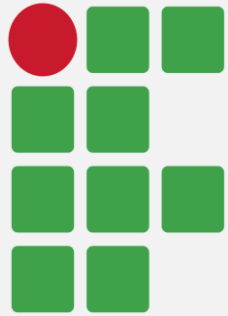
Fundamentos do Clean Code



- O código é lido muito (pelo menos sempre que alguém está escrevendo mais código), então qualquer escola de código limpo deve enfatizar a legibilidade. Limpar um pouco aonde quer que você vá é necessário para manter o código limpo.

EIXOS DO CLEAN CODE

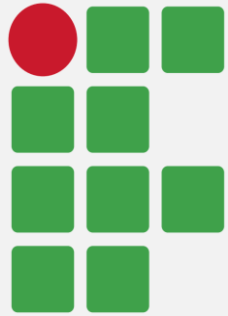
Eixos do Clean Code



- Nomes significativos
- Funções/Métodos;
- Comentários;
- Forma;
- Objetos e estruturas de dados;
- Gerenciamento de erros;
- Limites;
- Testes unitários;
- Classes;
- Sistemas;
- Emergência de design;
- Concorrência

NOMES SIGNIFICATIVOS

Nomes Significativos



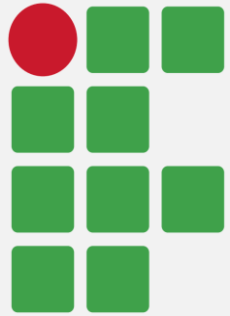
- Use nomes significativos, que revelem intenção;

- Use nomes pronunciáveis;

- Evite a desinformação (semelhanças acidentais com outra coisa ou diferenças de nomes muito sutis) e trocadilhos;

- Escopos maiores exigem nomes mais longos (para pesquisas bem sucedidas);

Nomes Significativos

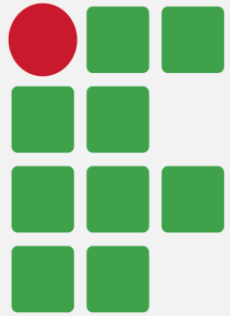


- Nomes de classe devem ser substantivos, nomes de métodos verbos;

- Use consistentemente a mesma palavra para um mesmo conceito.

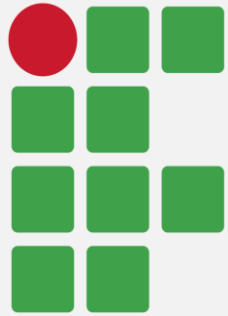
- Não tenha medo de mudar globalmente nomes ruins (incluindo os seus usos, é claro!)

Nomes Significativos



- Ruim:
  - int d; // elapsed time in days
- Bom:
  - int elapsedTimeInDays;
  - int daysSinceCreation;
  - int daysSinceModification;
  - int fileAgeInDays;

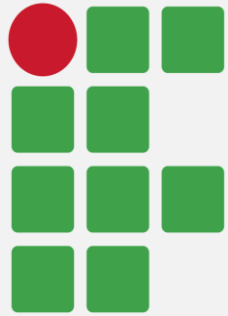
Nomes Significativos



- Ruim

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Nomes Significativos

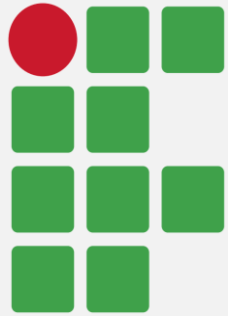


- Bom

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

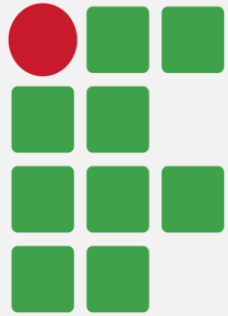
FUNÇÕES / MÉTODOS

Funções / Métodos



- Funções (métodos) devem ser pequenos e fazer apenas uma coisa;
- Todas as declarações devem estar um nível de abstração abaixo do conceito representado pelo método;
- Isso implica evitar estruturas de controle aninhadas, instruções switch, e a maioria dos if-else-if;
- Ordenar as funções em ordem de profundidade, de modo que o código geral possa ser lido de cima para baixo;

Funções / Métodos



- Funções (métodos) devem ser pequenos e fazer apenas uma coisa;

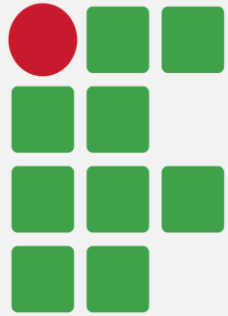
- Todas as declarações devem estar um nível de abstração abaixo do conceito representado pelo método;

- Isso implica evitar estruturas de controle aninhadas, instruções switch, e a maioria dos if-else-if;

- Ordenar as funções em ordem de profundidade, de modo que o código geral possa ser lido de cima para baixo;

```php
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

Funções / Métodos

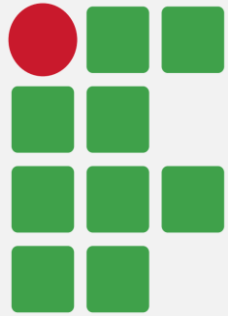


- É difícil superestimar a importância de nomes consistentes e descritivos, bem como da ausência de efeitos secundários surpreendentes;

- Os parâmetros tornam as funções mais difíceis de entender e muitas vezes estão em um nível mais baixo de abstração, então evite-os o tanto quanto possível;

- Evite a duplicação;

Funções / Métodos



- Tudo isso descreve um bom resultado final.

- Mas, inicialmente, você pode ter métodos com muito código, mal nomeado, complexo, cheio de parâmetros que fazem muitas coisas: isso não é problema, desde que você vá e refatore, refatore e refatore!

Funções / Métodos

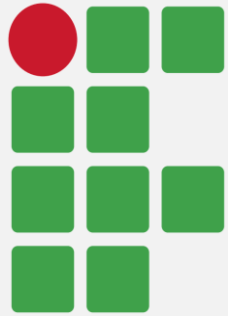
- Exemplo de efeitos colaterais

Listing 3-6

UserValidator.java

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

Funções / Métodos



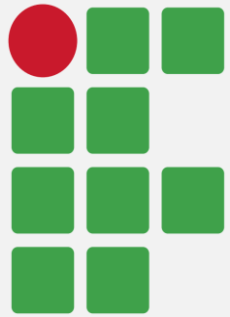
- CQRS: Command and Query Responsibility Separation

- Funções devem devem retornar algo ou fazer algo, mas não ambos;

- Ou seu método deve alterar o estado de um objeto, ou retornar alguma informação sobre esse objeto. Fazer ambos geralmente leva a confusão;

COMENTÁRIOS

Comentários



- "O uso adequado dos comentários é para compensar a nossa incapacidade de nos expressar através do código";

- Comentários não compensam o mau código. Em vez disso, devemos nos expressar através do código.

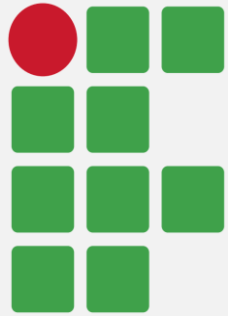
Comentários

- Tipos de bons comentários são: comentários legais, comentários informativos, explicações de intenção, aviso de consequências, TODO, marcando como importante, documentando a API pública;

Comentários

- Tipos de comentários ruins são: “murmúrios”, comentários redundantes, comentários enganosos, comentários obrigatórios, comentários de changelog, comentário em vez de colocar código função separada, “bandeiras”, comentários de encerramento, código comentado, comentários HTMLificados, informações não-locais, informações demais ou irrelevantes, comentários que precisam explicação, documentação de API não pública.

Comentários



- Isso

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

- Ou isso?

```
if (employee.isEligibleForFullBenefits())
```

FORMATO

Formato

- Formato adequado e uniforme é necessário se você pretende comunicar ordem para os leitores do seu código e fornecer legibilidade;

- Use uma ferramenta de formatação;

- Evite arquivos muito longos; ~200 linhas está ok;

- Os bons arquivos são como artigos de jornal: um bom cabeçalho, as coisas importantes primeiro e os detalhes mais tarde;

Formato

- Não deixe as linhas ficarem muito longas (80 ou 120 caracteres é OK);

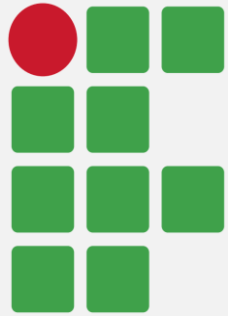
- Use espaços em branco horizontais para indicar relacionamento e separação;

- Use indentação corretamente;

- Use regras de formatação para toda a equipe

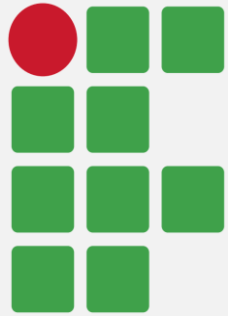
OBJETOS E ESTRUTURAS DE DADOS

Objetos e estruturas de dados



- Decida conscientemente o que esconder nos seu objetos: isso depende de quais mudanças são esperadas (e, por vezes, estruturas de dados públicas são ok);

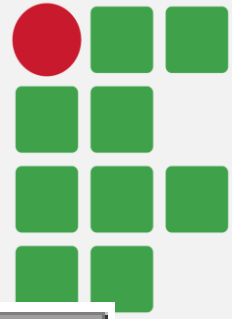
Objetos e estruturas de dados



- Existe uma razão pelas quais as variáveis são mantidas privadas: queremos ter a liberdade de mudar o tipo ou implementação delas sem quebrar o código.

- Porque, então, alguns programadores automaticamente adicionam getters/setters em suas classes assim que elas são criadas, expondo suas variáveis privadas como se fossem públicas?

Objetos e estruturas de dados



Listing 6-1

Concrete Point

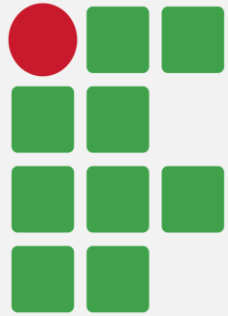
```
public class Point {
  public double x;
  public double y;
}
```

Listing 6-2

Abstract Point

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Objetos e estruturas de dados



- Objetos escondem seus dados através de abstrações e expõe métodos que operam sobre os dados;

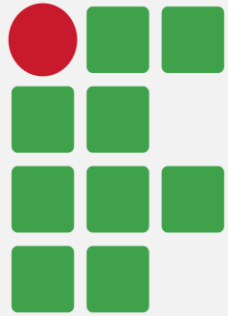
- Estruturas de dados expõe seus dados e não tem funções significativas;

Listing 6-5

Procedural Shape

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
```

Objetos e estruturas de dados

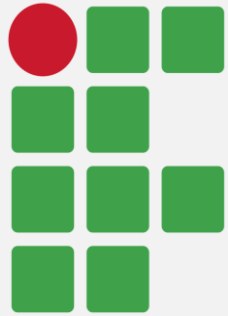


Listing 6-5 (continued)

Procedural Shape

```
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Objetos e estruturas de dados

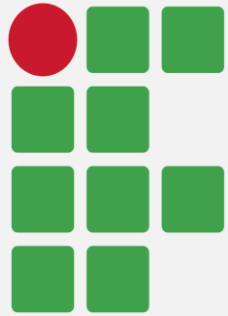


Listing 6-6

Polymorphic Shapes

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

Objetos e estruturas de dados

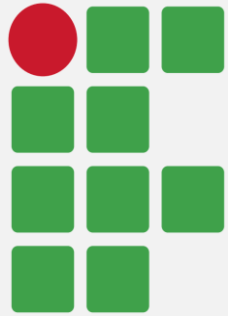


Listing 6-6 (continued)

Polymorphic Shapes

```
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

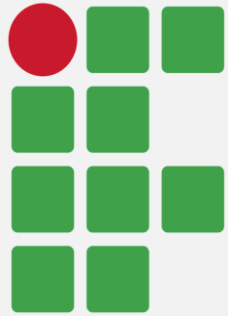
Objetos e estruturas de dados



- Código procedural (usando estruturas de dados) torna fácil adicionarmos novas funções sem modificar estruturas de dados existentes;

- Código orientado a objetos, por outro lado, torna fácil adicionar novas classes sem mudar funções existentes;

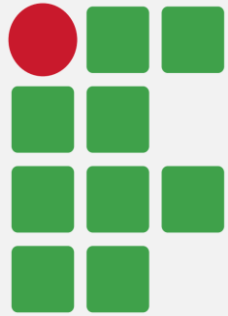
Objetos e estruturas de dados



- Código procedural torna difícil adicionar novas estruturas de dados porque as funções devem mudar;

- Código OO torna difícil adicionar novas funções porque as classes devem mudar;

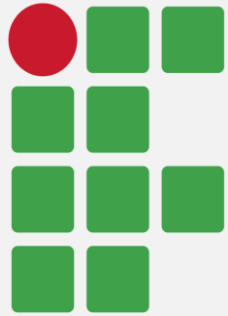
Objetos e estruturas de dados



- Código procedural torna difícil adicionar novas estruturas de dados porque as funções devem mudar;

- Código OO torna difícil adicionar novas funções porque as classes devem mudar;

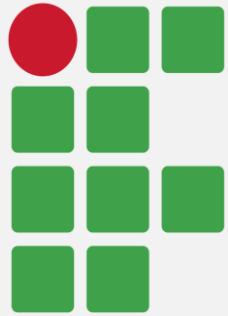
Objetos e estruturas de dados



- De preferência, chame apenas métodos próprios da classe, de objetos que você acabou de criar, de parâmetros e variáveis de instância, não outros métodos acessíveis através destes objetos (lei de Demeter!);

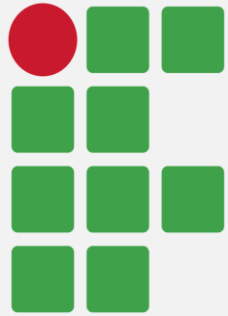
- Para todos os métodos que estão no objeto X, estes somente podem se comunicar com :

Objetos e estruturas de dados



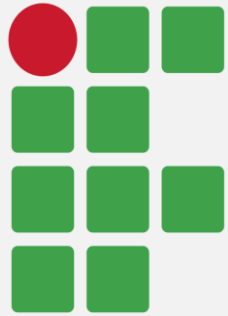
- Para todos os métodos que estão no objeto X, estes somente podem se comunicar com:
  - Métodos de X;
  - Parâmetros do próprio método;
  - Por objetos criados ou instanciados pelo próprio método;
  - Atributos de X;

Objetos e estruturas de dados



```java
package rs.com.teste.model;

import java.util.ArrayList;
import java.util.List;

public class CarrinhoCompras {

    private List<Itens> itens;

    public CarrinhoCompras() {
        this.itens = new ArrayList<Itens>();
    }

    public Double valorTotalDosItens(){
        Double valorTotal = 0.0;
        for (Itens item : itens) {
            valorTotal += item.getProduto().getValor();
        }
        return valorTotal;
    }

}
```

Objetos e estruturas de dados

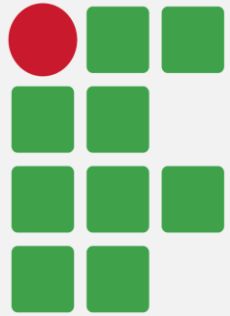


```java
package rs.com.teste.model;

import java.util.ArrayList;
import java.util.List;

public class CarrinhoCompras {

    private List<Itens> itens;

    public CarrinhoCompras() {
        this.itens = new ArrayList<Itens>();
    }

    public Double valorTotalDosItens(){
        Double valorTotal = 0.0;
        for (Itens item : itens) {
            valorTotal += item.getValorDoProduto();
        }
        return valorTotal;
    }

}

```

Retirado de: http://joleques.blogspot.com.br/2013/08/law-of-demeterlei-de-demeter.html

TRATAMENTO DE ERROS

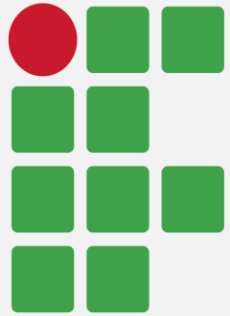
Tratamento de erros



- O tratamento de erros é importante e muitas vezes há muito tratamento de erro em nossa aplicação; entretanto, este não deve obscurecer as intenções do código, então use exceções (não códigos de retorno) e trate os blocos try-catch como transações.

- Suas exceções devem fornecer intenção, contexto e detalhes do tipo de erro.

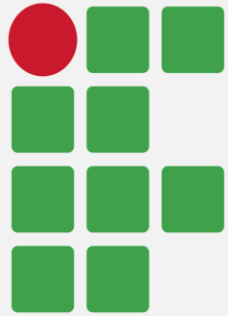
Tratamento de erros



- Envolva APIs de terceiros para remapear suas exceções, conforme necessário.

- Verificações de nulos são incômodas, assim como códigos de retorno; use exceções ou “null objects” em vez de retornar ou aceitar nulo.

Tratamento de erros

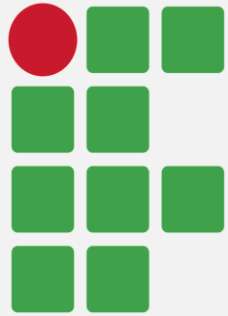


Listing 7-1

DeviceController.java

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Check the state of the device
    if (handle != DeviceHandle.INVALID) {
      // Save the device status to the record field
      retrieveDeviceRecord(handle);
      // If not suspended, shut down
      if (record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Device suspended.  Unable to shut down");
      }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  ...
}
```

Tratamento de erros

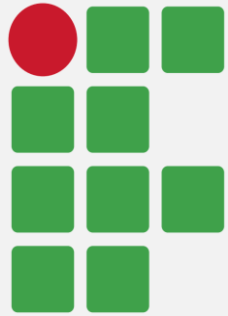


Listing 7-2

DeviceController.java (with exceptions)

```
public class DeviceController {
  ...

  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }
```

Tratamento de erros



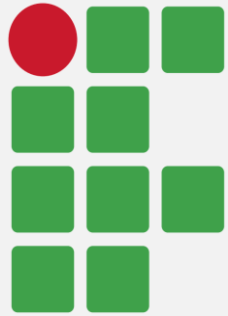
Listing 7-2 (continued)

DeviceController.java (with exceptions)

```
  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }

  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
  }

  ...
}
```

LIMITES

Limites



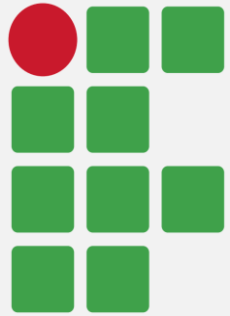
- Mantenha os limites limpos entre o código proveniente de diferentes equipes, open source ou de terceiros;

- Não passe amplamente em sua base de código objetos facilmente suscetíveis a mudanças por terceiros;

- Aprenda e documente bibliotecas de terceiros através da escrita de “testes de aprendizagem” para estes;

TESTES UNITÁRIOS

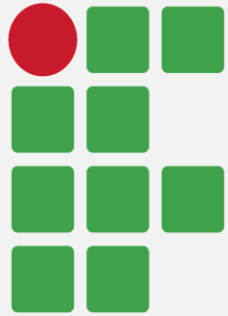
Testes Unitários



- Testes automatizados devem abranger todos os detalhes de funcionalidade do nosso código e devem ser fáceis de executar.

- Eles devem ser criados com o Test-Driven Development (TDD).

Testes Unitários

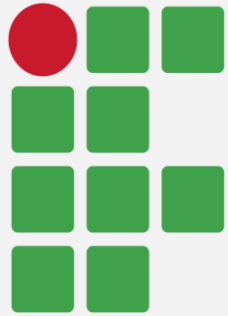


- Você não pode escrever código de produção até que você tenha escrito um teste falhando;

- Você não pode escrever mais testes do que é suficiente para falhar (não compilar significa falhar);

- Você não pode escrever mais código de produção do que é suficiente para passar o teste falhando no momento.

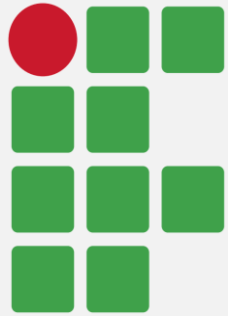
Testes Unitários



- Este estilo de codificação produz ciclos curtos em que se desenvolve o código de produção.

- Os testes devem ser tão limpos quanto o código, tendo em vista que estes irão mudar, também, conforme necessário; então sempre refatore ambos.

Testes Unitários

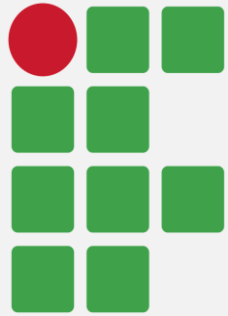


- Os testes devem, cada um, verificar um conceito único;

- Test F.I.R.S.T
  - F: fast (rápido)
  - I: independentes
  - R: repetíveis
  - S: self-validating (auto validantes)
  - T: timely (devem ser escritos imediatamente antes do código que testam)

CLASSES

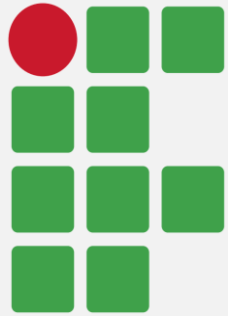
Classes



- Ordem: constantes antes das variáveis; variáveis antes dos métodos (público antes de privado, mas métodos privados usados apenas uma vez seguem logo após seu uso);

- Mantenha as variáveis e os métodos de utilidade privados a menos que isso interfira no teste;

Classes



- As classes devem ser pequenas: ter apenas uma responsabilidade (Princípio de Responsabilidade Única)

- (SRP): tenha apenas um motivo para mudar.

- Se uma descrição de 25 palavras da classe e suas responsabilidades usa o termo "e", seja cauteloso.

Classes

- Coesão: um método que acessa mais do que uma variável da classe é mais coeso com a classe;

- Coesão geral-baixa tende a ser ruim e pode indicar que a classe deve ser dividida;

- Divisão das classes também pode ser necessária para suportar o princípio Aberto-Fechado, ou seja, evitar modificar uma classe quando se estende a funcionalidade do sistema;

Classes

- Também é importante seguir o princípio da inversão de dependência: ao invés de fazer chamadas a serviços através de “hard-code”, dependa de uma abstração (interface) e passe o serviço concreto (objeto) como um parâmetro;

SISTEMAS

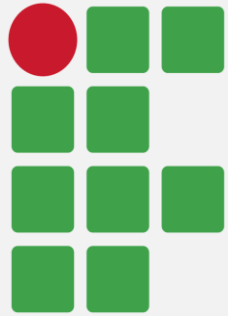
Sistemas

- Obedeça ao princípio da separação das responsabilidades;

- Nunca permita que conveniências levem a ruptura de modularidade, por exemplo o “hard-code” de dependências;

- O processo de instanciamento é uma importante preocupação: use fábricas e Injeção de Dependência (DI), que aplica a Inversão de Controle (IoC);

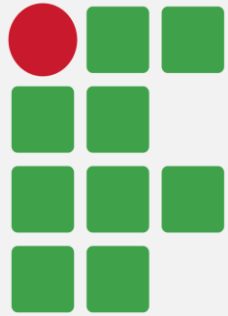
Sistemas



- Delegue a criação de dependências para objetos que são especializado para essa tarefa (ou explicitamente ou, de preferência, através de um construtor adequado ou de métodos setter);

- Isso também apoia o princípio da responsabilidade única!

Sistemas

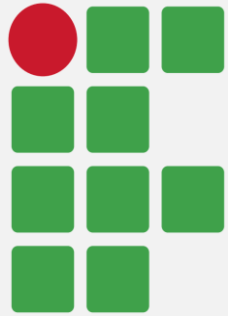


- A adequada separação das responsabilidades permitirá crescer até a estrutura arquitetônica de um sistema;

- Desacoplamento total (através, por exemplo, de POJOs) permitirá mudanças arquitetônicas (por exemplo, trocar tecnologias de persistência) facilmente;

- Isso também simplifica descentralizar ou adiar decisões;

Sistemas

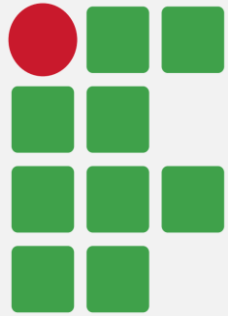


- DSLs ajudam a manter a lógica da aplicação concisa, legível e modificável;

- Nunca se esqueça de usar a coisa mais simples que pode funcionar!

EMERGÊNCIA DE DESIGN

Emergência de design



- Bons designs podem ser produzidos os deixando emergir do uso das quatro regras do Design Simples de Kent Beck;

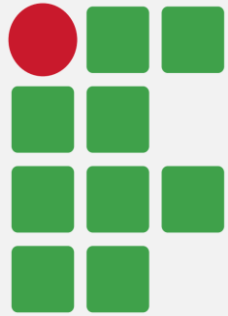
- Um Design Simples é um design que
  - Roda todos os testes - tudo é testado e nada falha;
  - Não contém duplicações;
  - Expressa a intenção do programador;
  - Minimiza o número de classes e métodos

Emergência de design

- Lembrando: não há um conjunto simples de práticas que substituam a experiência. Entretanto, as práticas propostas são o resultado dos anos de experiência dos autores: segui-lás encoraja os desenvolvedores a aderir a bons princípios e padrões que, de outra forma, levam anos para aprender.

CONCORRÊNCIA

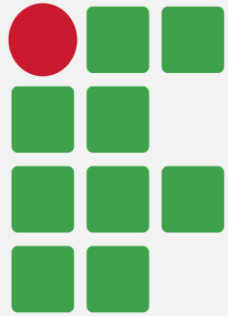
Concorrência



- Concorrência desacopla o que é feito de quando isto é feito, podendo melhorar - ou complicar - a estrutura, compreensibilidade e eficiência de um sistema;

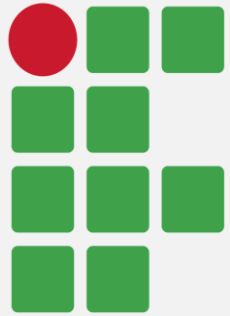
- A evolução do estado do programa, de qualquer forma, se torna muito mais complexa. Concorrência é complexa, ainda que para problemas simples.

Concorrência



- Portanto:
  - Obedeça estritamente o SRP: mantenha código de gerenciamento de concorrência separado de outros códigos;
  - Conheça suas bibliotecas: quais são thread safe vs não thread safe, blocantes vs não blocantes, etc;
  - Conheça conceitos: limites de recursos, exclusão mútua, deadlock

Concorrência

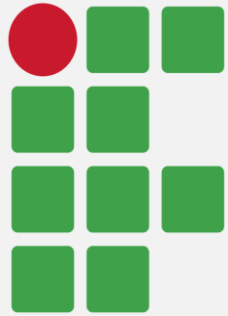


- Conheça modelos de programação: produtor-consumir, jantar dos filósofos, etc;

- Teste com variações - número de threads, etc - e rastreie cada falha; faça o código não-threaded funcionar confiavelmente primeiro!

REFINAMENTOS SUCESSIVOS

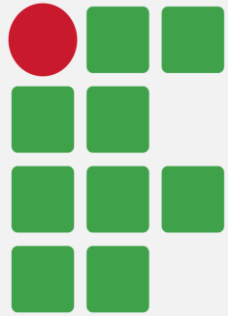
Refinamentos sucessivos



- Os último capítulos do livro, “refinamentos sucessivos”, “JUnit internals” e “Refatorando SerialDate” são dedicados a estudos de caso demonstrando as técnicas e conceitos abordados no decorrer do livro;

- O último capítulo “Smells and Heuristics” é uma lista compilada das heurísticas e “odores” descritos no decorrer do livro;

Leituras sugeridas



- Clean Code: A Handbook of Agile Software Craftsmanship
- Clean Architecture: A Craftsman's Guide to Software Structure and Design
- https://martinfowler.com/bliki/CQRS.html
- https://martinfowler.com/bliki/BeckDesignRules.html
- https://martinfowler.com/bliki/TechnicalDebt.html
- https://martinfowler.com/bliki/CodeSmell.html
- https://martinfowler.com/ieeeSoftware/explicit.pdf