Curso Java Básico

Uma introdução prática usando BlueJ

Design de classes

Como escrever classes compreensíveis, manuteníveis e reutilizáveis

Principais conceitos a serem abordados

- Acoplamento
- Coesão
- Refatoração
- Design baseado em responsabilidade
- Tipos enumerados

Mudar ou morrer

- Um software não é como um romance que é escrito uma vez e então permanece inalterado.
- Um software é estendido, corrigido, mantido, portado, adaptado…
- O trabalho é feito por diferentes pessoas ao longo do tempo (frequentemente em décadas).

Mudar ou morrer

- Há somente duas opções para um software:
  - ser mantido continuamente; ou
  - morrer.
- Um software que não pode ser mantido será descartado.

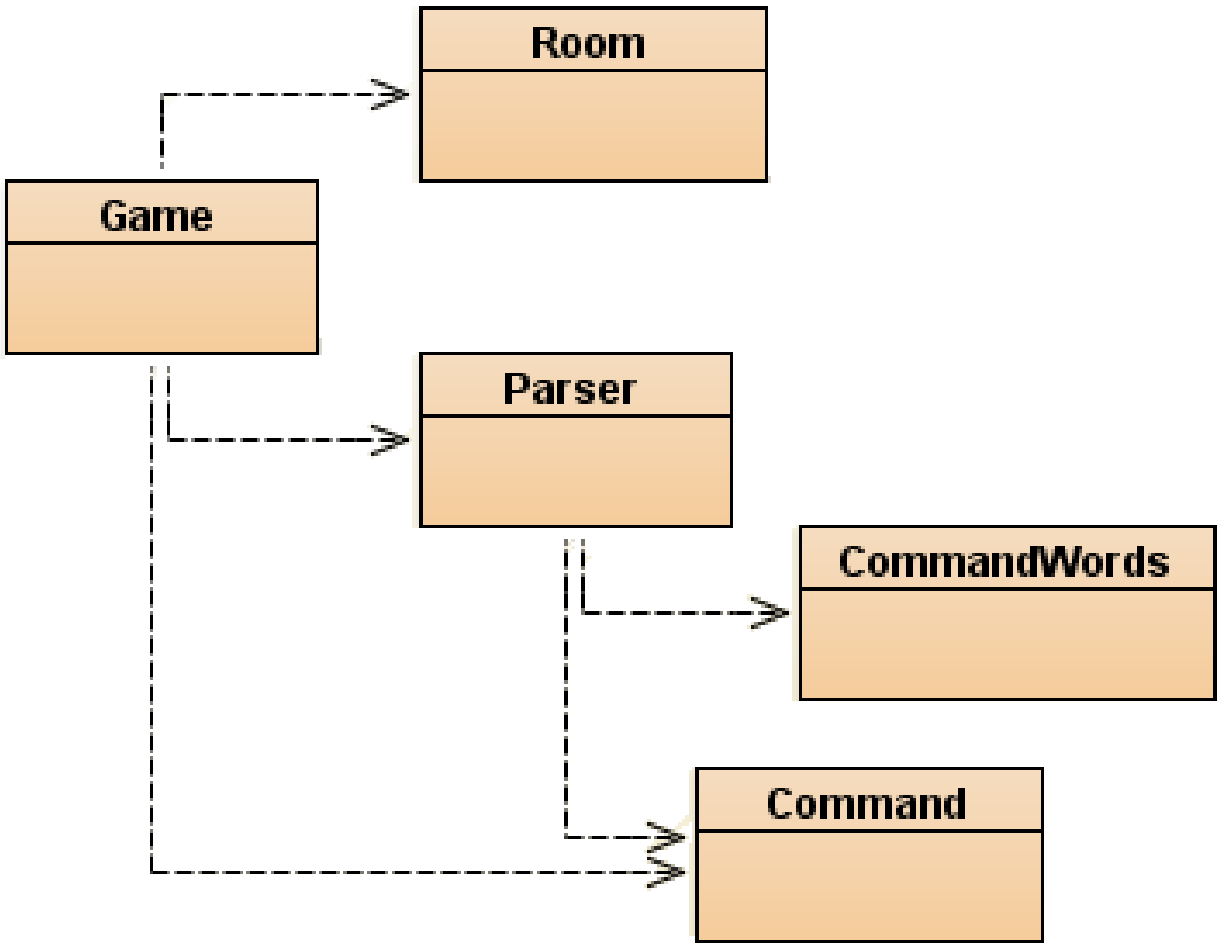
Design bom ou ruim

- Estudaremos uma implementação bastante primitiva de um jogo baseado em texto.
- Apesar do jogo funcionar bem, ele tem problemas claros de design ruim.
- Estas decisões ruins de design, muitas vezes, só ficam óbvias durante a manutenção ou o reuso.
- Enquanto extendemos a aplicação original, discutiremos aspectos do design das classes existentes.

World-of-Zuul

- Nosso jogo World-of-Zuul tem como modelo o jogo original *Adventure*, que foi desenvolvido no início da década de 1970 por Will Crowther e expandido por Don Woods.
- Este jogo envolvia a busca, através de cavernas, de um tesouro escondido.
- Utilizando palavras secretas e outros mistérios, tinha como objetivo marcar o maior número de pontos.

World-of-Zuul



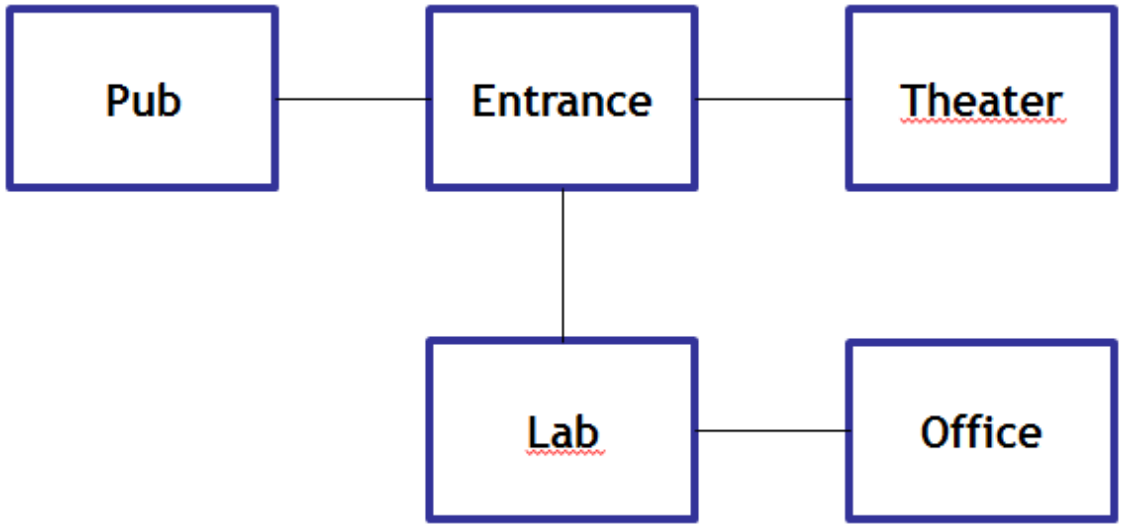
Classes do World-of-Zuul

- **`Game`**: controlador do jogo.
- **`Room`**: um local do jogo, que pode ter saídas para outros locais.
- **`Parser`**: leitor da entrada do usuário.
- **`Command`**: um comando do usuário.
- **`CommandWord`**: palavra reconhecida como um comando.

Exercício

- **World-of-Zuul**
  - Inicie o **BlueJ**, abra o projeto *zuul-bad* e crie uma instância de *Game*.
  - Chame o método *play* e explore o jogo:
    - o que essa aplicação faz ?
    - que comandos o jogo aceita ?
    - o que cada comando faz ?
    - quantas salas estão no cenário ?
  - Desenhe um mapa das salas do jogo.

Salas do World-of-Zuul



Qualidade do código

- Dois conceitos importantes quanto à qualidade do código:
  - Acoplamento
    - vinculação entre as unidades
    - bom design tem ***baixo acoplamento***
  - Coesão
    - afinidade de tarefas de uma unidade
    - bom design tem ***alta coesão***

Acoplamento

- O acoplamento se refere à vinculação de unidades separadas de um programa.
- Se duas classes dependerem intimamente de muitos detalhes entre si, dizemos que elas estão *totalmente integradas*.
- Nosso objetivo é um ***baixo acoplamento***.

Acoplamento

- O baixo acoplamento torna possível:
  - entender uma classe sem ler outras
  - alterar uma classe sem afetar outras
- Assim, melhora a manutenibilidade.

Coesão

- Coesão se refere ao número e à diversidade de tarefas pelos quais uma única unidade é responsável.
- Se cada unidade for responsável por uma ***única tarefa lógica***, dizemos que ela tem ***alta coesão***.
- A coesão se aplica a classes e a métodos.
- Nosso objetivo é uma ***coesão alta***.

Coesão

- A coesão alta torna mais fácil :
  - utilizar nomes descritivos para cada classe ou método
  - entender o que faz cada classe ou método
  - reutilizar classes ou métodos

Coesão

- Coesão de classes :
  - cada classe deve representar uma única e bem definida entidade lógica.
- Coesão de métodos:
  - cada método deve ser responsável por uma única e bem definida tarefa lógica.

Código-fonte: Game

```
public class Game
{
    private Parser parser;
    private Room currentRoom;

    public Game()
    {
        createRooms();
        parser = new Parser();
    }
    ...
}
```

Código-fonte: Game

```
private void createRooms()
{
    Room outside, theatre, pub, lab, office;
    outside = new Room("outside the main entrance " +
            "of the university");
    theatre = new Room("in a lecture theatre");
    pub = new Room("in the campus pub");
    lab = new Room("in a computing lab");
    office = new Room("in the computing admin office");

    outside.setExits(null, theatre, lab, pub);
    theatre.setExits(null, null, null, outside);
    pub.setExits(null, outside, null, null);
    lab.setExits(outside, office, null, null);
    office.setExits(null, null, null, lab);

    currentRoom = outside;  // start game outside
}
```

Código-fonte: Game

```
public void play()
{
    printWelcome();

    boolean finished = false;
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);
    }

    System.out.println("Thank you for playing." +
        " Good bye.");
}
```

Código-fonte: Game

```
private boolean processCommand(Command command)
{
    boolean wantToQuit = false;
    if(command.isUnknown()) {
      System.out.println("I don't know what you mean...");
      return false;
    }
    String commandWord = command.getCommandWord();
    if (commandWord.equals("help")) {
        printHelp();
    } else if (commandWord.equals("go")) {
        goRoom(command);
    } else if (commandWord.equals("quit")) {
        wantToQuit = quit(command);
    }
    return wantToQuit; // else command not recognised.
}
```

Código-fonte: Game

```
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to the World of Zuul!");
    System.out.println("World of Zuul is a new," +
                       " incredibly boring adventure game.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
```

Código-fonte: Game

```
        System.out.println("You are " +
                currentRoom.getDescription());
        System.out.print("Exits: ");
        if(currentRoom.northExit != null) {
            System.out.print("north ");
        }
        if(currentRoom.eastExit != null) {
            System.out.print("east ");
        }
        if(currentRoom.southExit != null) {
            System.out.print("south ");
        }
        if(currentRoom.westExit != null) {
            System.out.print("west ");
        }
        System.out.println();
}
```

Código-fonte: Game

```
private void goRoom(Command command)
{
    if(!command.hasSecondWord()) {
        System.out.println("Go where?");
        return;
    }
    String direction = command.getSecondWord();

    Room nextRoom = null;
    if(direction.equals("north")) {
        nextRoom = currentRoom.northExit;
    }
    if(direction.equals("east")) {
        nextRoom = currentRoom.eastExit;
    }
```

Código-fonte: Game

```
if(direction.equals("south")) {
    nextRoom = currentRoom.southExit;
}
if(direction.equals("west")) {
    nextRoom = currentRoom.westExit;
}

if (nextRoom == null) {
    System.out.println("There is no door!");
}
else {
    currentRoom = nextRoom;
```

Código-fonte: Game

```
        System.out.println("You are " +
                currentRoom.getDescription());
        System.out.print("Exits: ");
        if(currentRoom.northExit != null) {
            System.out.print("north ");
        }
        if(currentRoom.eastExit != null) {
            System.out.print("east ");
        }
        if(currentRoom.southExit != null) {
            System.out.print("south ");
        }
        if(currentRoom.westExit != null) {
            System.out.print("west ");
        }
        System.out.println();
}
```

Duplicação do código

- A duplicação do código:
  - é um indicador de design ruim
  - torna a manutenção mais difícil
  - pode levar à introdução de erros durante a manutenção
- A duplicação do código normalmente é um sinal de baixa coesão.

Duplicação do código

- Os métodos **`printWelcome`** e **`goRoom`** contém o seguinte código:

```
System.out.println("You are " + currentRoom.getDescription());
System.out.print("Exits: ");
if(currentRoom.northExit != null) {
    System.out.print("north ");
}
if(currentRoom.eastExit != null) {
    System.out.print("east ");
}
if(currentRoom.southExit != null) {
    System.out.print("south ");
}
if(currentRoom.westExit != null) {
    System.out.print("west ");
}
System.out.println();
```

Duplicação do código

- A duplicação de código deve-se ao fato de que ambos os métodos fazem duas coisas:
  - **`printWelcome:`**
    - imprime a mensagem de boas-vindas, e
    - imprime as informações sobre a localização atual
  - **`goRoom:`**
    - altera a localização atual, e
    - imprime as informações sobre a localização atual

Exercício

- **World-of-Zuul**
  - Salve o projeto atual como *zuul*.
  - Substitua o código duplicado em *printWelcome* e *goRoom* por uma chamada ao método *printLocationInfo*, que você deve criar com o código extraído daqueles métodos.

Fazendo extensões

- Apesar do jogo funcionar bem, ele apresenta um design ruim.
- Ao tentarmos fazer modificações no projeto, notaremos que a quantidade de trabalho necessária é muito maior do que se ele tivesse um bom design.

Fazendo extensões

- Considere que os requisitos do jogo mudaram e agora devemos adicionar mais duas direções de movimento: *para cima* e *para baixo*.
- Duas classes estão envolvidas nesta alteração:
  - Room (armazena as saídas de cada sala)
  - Game (processa a mudança de sala)

Código-fonte: Room

```
public class Room
{
    public String description;
    public Room northExit;
    public Room southExit;
    public Room eastExit;
    public Room westExit;

    public Room(String description)
    {
        this.description = description;
    }
    ...
}
```

Código-fonte: Room

```
public void setExits(Room north, Room east,
                     Room south, Room west)
{
    if(north != null) {northExit = north;}
    if(east != null)  {eastExit = east;}
    if(south != null) {southExit = south;}
    if(west != null)  {westExit = west;}
}

public String getDescription()
{
    return description;
}
```

Melhorando a estrutura interna

- Acrescentar mais saídas na classe **`Room`** atualmente envolve:
  - mais campos
  - mais parâmetros em **`setExists`**
- Seria melhor se a classe **`Room`**, ao invés de usar campos individuais para cada saída, usasse um mapa para vincular cada saída à sala correspondente.

Acoplamento

- Alterar a estrutura interna da classe **`Room`** não deveria ter impacto em outras classes.
- Entretanto, fazendo isso a classe **`Game`** não compilará mais.
- A existência deste impacto é um sinal de alto acoplamento.

Acoplamento

- A classe **`Room`** usa campos públicos que são acessados diretamente pela classe **`Game`**.
- O design atual não segue a diretriz de ocultamento de informações:
  - dados que pertencem a um objeto devem ser ocultados de outros objetos
- Devemos encapsular os campos de **`Room`**, tornando-os privados e fornecendo um único método de acesso parametrizado.

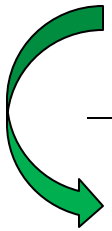
Código-fonte: Room

```
public class Room
{
    private String description;
    private Room northExit;
    private Room southExit;
    private Room eastExit;
    private Room westExit;
    ...
    public Room getExit(String direction)
    {
        if(direction.equals("north")) {return northExit;}
        if(direction.equals("east"))  {return eastExit;}
        if(direction.equals("south")) {return southExit;}
        if(direction.equals("west"))  {return westExit;}
        return null;
    }
}
```

Campos privados

Método de acesso

Código-fonte: Game

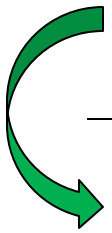


```
if(currentRoom.northExit != null)
```

```
if(currentRoom.getExit("north") != null)
```

Código-fonte: Game

```
Room nextRoom = null;
if(direction.equals("north")) {
    nextRoom = currentRoom.northExit;
}
if(direction.equals("east")) {
    nextRoom = currentRoom.eastExit;
}
if(direction.equals("south")) {
    nextRoom = currentRoom.southExit;
}
if(direction.equals("west")) {
    nextRoom = currentRoom.westExit;
}
```



```
Room nextRoom = currentRoom.getExit(direction);
```

Exercício

- **World-of-Zuul**
  - Altere a classe *Room* tornando seus campos privados e fornecendo um método de acesso que receba a direção da saída e informe a sala correspondente.
  - Altere os métodos *printLocationInfo* e *goRoom* da classe *Game* para que usem o método de acesso de *Room*.

Melhorando a estrutura interna

- Tendo ocultado a estrutura interna da classe **`Room`**, podemos alterá-la sem impactar a classe **`Game`**.
- Agora podemos substituir em **`Room`** seus diversos campos com saídas individuais por um único campo do tipo **`Map`** com todas as saídas.

Código-fonte: Room

```
public class Room
{
    private String description;
    private HashMap<String, Room> exits;

    public Room(String description)
    {
        this.description = description;
        exits = new HashMap<String, Room>();
    }

    public String getDescription()
    {
        return description;
    }
    ...
}
```

Campos Privados (simplificados)

Código-fonte: Room

```
public void setExits(Room north, Room east,
                     Room south, Room west)
{
    if(north != null) {exits.put("north", north);}
    if(east != null)  {exits.put("east", east);}
    if(south != null) {exits.put("south", south);}
    if(west != null)  {exits.put("west", west);}
}

public Room getExit(String direction)
{
    return exits.get(direction);
}
```

Método de acesso (simplificado)

Exercício

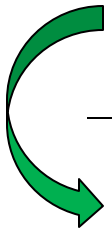
- **World-of-Zuul**
  - Altere a classe *Room* substituindo seus campos individuais de saída por um campo do tipo HashMap<String, Room>, ajustando o construtor e os métodos setExits e getExit.

Melhorando a interface pública

- Fizemos diversas alterações nas classes **`Room`** e **`Game`** para incluir mais direções de movimento e seu design melhorou muito.
- Entretanto, o método **`setExits`** de **`Room`** configura todas as saídas possíveis.
- Seria melhor um método modificador que não dependesse da totalidade de saídas.
- Fazendo isso, alteramos a interface pública de **`Room`** (causando impacto em **`Game`**), mas melhoramos seu design.

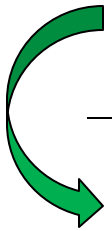
Código-fonte: Room

```
public void setExits(Room north, Room east,
                     Room south, Room west)
{
    if(north != null) {exits.put("north", north);}
    if(east != null)  {exits.put("east", east);}
    if(south != null) {exits.put("south", south);}
    if(west != null)  {exits.put("west", west);}
}
```



```
public void setExit(String direction, Room neighbor)
{
    if (neighbor != null) {
        exits.put(direction, neighbor);
    }
}
```

Código-fonte: Game



```
lab.setExits(outside, office, null, null);
```

```
lab.setExit("north", outside);
lab.setExit("east", office);
```

Exercício

- **World-of-Zuul**
  - Altere a classe *Room* substituindo o método que configura todas as saídas possíveis por um método que configura uma por vez recebendo a direção da saída e a sala correspondente.
  - Altere o método *createRooms* da classe *Game* para que use o método acima.

Design baseado em responsabilidade

- Questão: onde (qual classe) devemos adicionar um novo método ?
- Cada classe deve ser responsável por manipular seus próprios dados.
- A classe que possui os dados deve ser responsável por processá-los.
- Design baseado em responsabilidade leva a uma alta coesão e a um baixo acoplamento.

Melhorando a interface pública

- Já podemos incluir mais direções ?
- O método **`createRooms`**, responsável pela inicialização das salas, deveria ser o único impactado pela inclusão de mais direções.
- Entretanto, **`printLocationInfo`** também conhece as direções possíveis.
- Alem disso, **`printLocationInfo`** sabe que há duas coisas a informar sobre **`Room`**: sua descrição e suas saídas.

Melhorando a interface pública

- Seria melhor se **`Room`** tivesse uma descrição longa, que informasse tudo sobre si: sua descrição curta, suas saídas e qualquer outra coisa relevante.
- A classe **`Game`** poderia eliminar o método **`printLocationInfo`** e usar a descrição longa de **`Room`** onde o método era chamado.
- Fazendo isso, alteramos a interface entre **`Room`** e **`Game`**, mas melhoramos ainda mais o design destas classes.

Código-fonte: Room

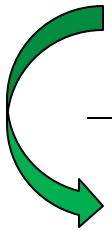
```
private String getExitString()
{
    StringBuilder sBuilder = new StringBuilder();
    sBuilder.append("Exits:");
    Set<String> keys = exits.keySet();
    for(String exit : keys) {
        sBuilder.append(" " + exit);
    }
    return sBuilder.toString();
}
```

Código-fonte: Room

```
public String getShortDescription()
{
    return description;
}

public String getLongDescription()
{
    StringBuilder sBuilder = new StringBuilder();
    sBuilder.append("You are " + description);
    sBuilder.append('\n');
    sBuilder.append(getExitString());
    return sBuilder.toString();
}
```

Código-fonte: Game



```
printLocationInfo();
```

```
System.out.println(currentRoom.getLongDescription());
```

Exercício

- **World-of-Zuul**
  - Altere a classe *Room* criando os métodos getExitString e getLongDescription e renomeando o método getDescription para getShortDescription.
  - Altere a classe *Game* eliminando o método printLocationInfo e substituindo suas chamadas pelo uso do método getLongDescription de *Room*, além de substituir as chamadas a getDescription por chamadas a getShortDescription.

Minimizando alterações

- Um dos objetivos do baixo acoplamento e do design baseado em responsabilidade é minimizar as alterações.
- Quando uma alteração é necessária, o número de classes afetadas deve ser o menor possível.
- Além disso, as alterações devem ser óbvias, fáceis de detectar e fáceis de executar.

Fazendo extensões

- Agora as classes **`Game`** e **`Room`** estão prontas para a adição de mais duas direções de movimento: *para cima* e *para baixo*.
- Suponha que os requisitos do jogo mudaram e agora devemos incluir um novo local (o porão) sob o escritório.
- Quais classes precisam ser alteradas ? Em quais métodos ?

Código-fonte: Game

```
private void createRooms()
{
    Room outside, theatre, pub, lab, office, cellar;
    ...
    cellar = new Room("in the cellar");
    ...
    office.setExit("down", cellar);
    cellar.setExit("up", office);
    ...
}
```

Exercício

- **World-of-Zuul**
  - Compare o estágio atual do projeto em uso com *zuul-better*.
  - Altere a classe *Game* adicionando um porão(cellar) sob o escritório(office).

Tipos Enumerados

- Um tipo *enum* é um tipo que define um conjunto fixo de constantes; pode ter campos, métodos e construtores.
- Deve ser usado sempre que for necessário representar um conjunto fixo de constantes.

Tipos Enumerados

- Não criamos objetos a partir de tipos enumerados; cada nome dentro de sua definição é uma instância do tipo criada automaticamente para uso.
- Um tipo enumerado pode ser usado como tipo de variável, como classes e interfaces.

Um tipo enumerado mais simples

```
public enum CommandWord
{
    GO, QUIT, HELP, UNKNOWN;
}
```

Um tipo enumerado mais elaborado

```
public enum CommandWord
{
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?");

    private String commandString;

    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    public String toString()
    {
        return commandString;
    }
}
```

Questões de design

Questões comuns:

- Qual deve ser o comprimento de uma classe?
- Qual deve ser o comprimento de um método?

- Agora elas podem ser respondidas em termos de coesão e acoplamento.

Diretrizes de design

- Um método será muito longo se realizar mais de uma tarefa lógica.
- Uma classe será muito complexa se representar mais de uma entidade lógica.

- Nota: essas são as *diretrizes* — elas ainda deixam muita coisa em aberto ao designer.

Refatoração

- Quando classes e métodos são mantidos, frequentemente algum código é adicionado.
- Classes e métodos tendem a se tornar mais longos.
- Às vezes, classes e métodos devem ser ***refatorados*** para manter a alta coesão e o baixo acoplamento.

Refatorando e testando

- Ao refatorar o código, não acrescente outras funcionalidades.
- Primeiro, faça apenas a refatoração, sem alterar as funcionalidades.
- Teste antes e depois da refatoração para se assegurar de que nada foi estragado.

Refatoração

http://www.refactoring.com/catalog/

- Encapsulate Field
- Rename Method
- Parameterize Method
- Extract Method & Extract Class
- Move Field & Move Method
- Pull Up Field & Pull Up Method
- Push Down Field & Push Down Method

Revisão (1)

- Programas são continuamente alterados.
- É importante tornar essas alterações possíveis.
- A qualidade do código requer muito mais do que simplesmente realizar uma correção em um dado momento.
- O código precisa ser compreensível e manutenível.

Revisão (2)

- Um código de boa qualidade evita a duplicação e exibe alta coesão e baixo acoplamento.
- O estilo de codificação (comentários, atribuição de nomes, layout etc.) também é importante.
- Há uma grande diferença quanto ao volume de trabalho exigido para fazer uma alteração em um código mal estruturado e em um bem estruturado.

Contatos

Câmara dos Deputados

CENIN - Centro de Informática

Carlos Renato S. Ramos

carlosrenato.ramos@camara.gov.br