FreeCol

Phase 3

Elementos do Grupo:

José Fernandes	52970	**GitHub:** JoseBFernandes
Brás Ramos	57800	**GitHub:** BritaPng
Eduardo Dias	62466	**GitHub: ej-dias**
Guilherme Carvalhão	62675	**GitHub:** g-carvalhao
Wagna Banar	64157	**GitHub:** Wagna-Banar

GitHub Repository

https://github.com/JoseBFernandes/SE2324_52970_57800_62466_62675_64157

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Índice

Fase 1

User Stories

User Storie 1:

Como utilizador, eu quero que o mapa do jogo esteja visível assim que se cria um novo jogo singlePlayer, para que assim posso conseguir ver onde fica o continente.

Consideramos esta “User Storie” visto que ao ter o mapa todo disponibilizado, teremos uma maior facilidade em começar o jogo. Ora ao jogar o jogo, notamos que tínhamos alguma dificuldade na nossa jogabilidade, isto porque não sabíamos para onde havíamos de mover o barco.

User Storie 2:

Como utilizador, eu quero ter uma forma melhor de mover o cenário do jogo, para que consiga navegar melhor pelo mapa.

Consideramos esta “User Storie”, visto que quando estávamos a jogar tivemos alguma dificuldade em mover o cenário. Queríamos assim colocar um “arrasto do rato” (drag) para mover o mapa do jogo.

User Storie 3:

Como utilizador, eu quero ter a possibilidade de adicionar mais ouro ao meu player utilizando uma Action, para desta forma ter uma vantagem perante o meu adversário no jogo singlePlayer.

Consideramos esta “User Storie”, principalmente pela dificuldade em começar o jogo com zero ouro e também porque podemos desta forma apreciar mais a jogabilidade deste.

https://www.youtube.com/watch?v=r_nZpbqX5gI

Fase 2

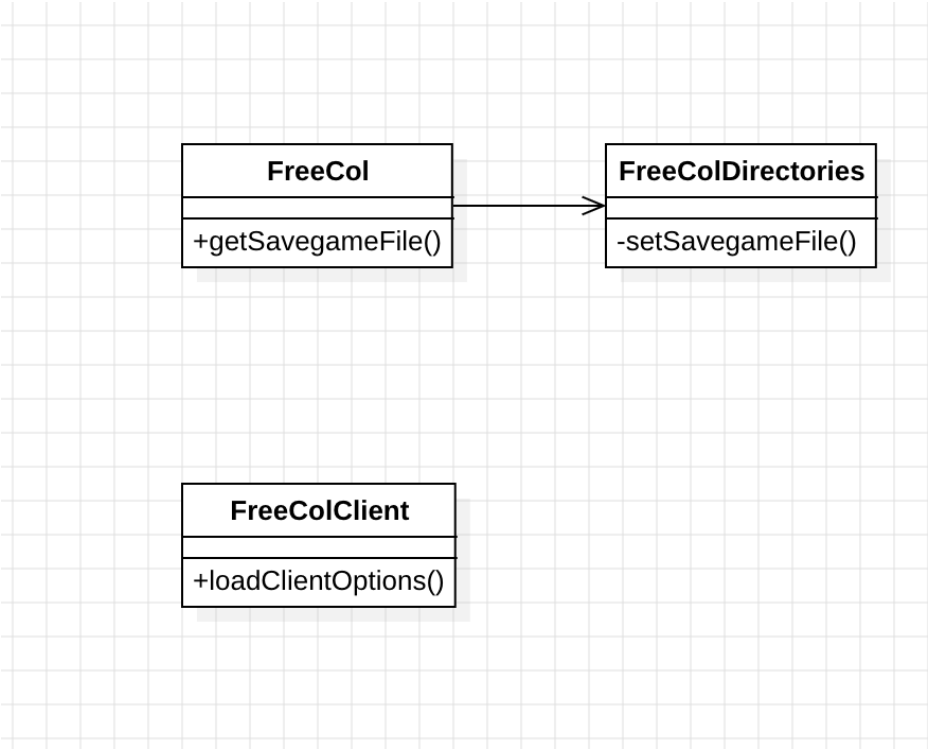
Design Patterns

José Fernandes - 52970

Memento Pattern

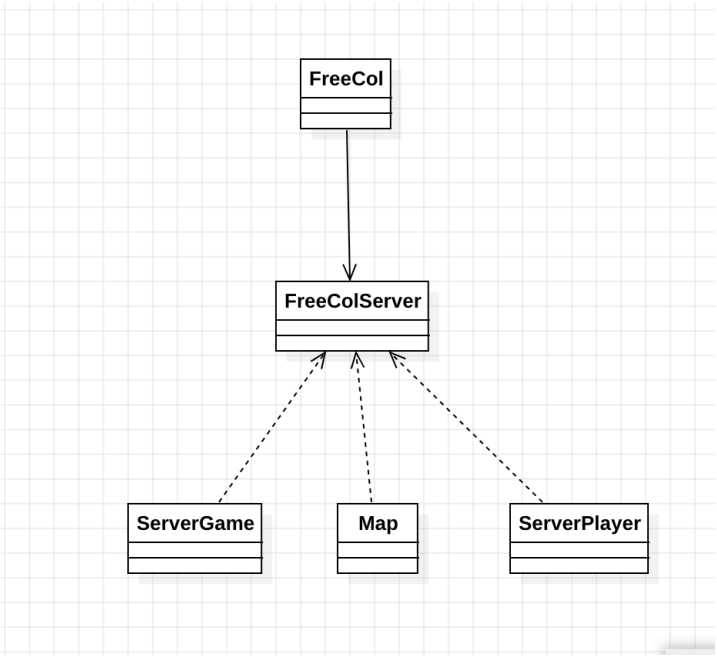
Usado para guardar o jogo e depois poder recuperá-lo numa outra altura, por exemplo:

- Na classe FreeCol onde se utiliza a classe FreeCol Directories para gravar e carregar de novo o jogo (guardar o estado do jogo).



Interpreter Pattern

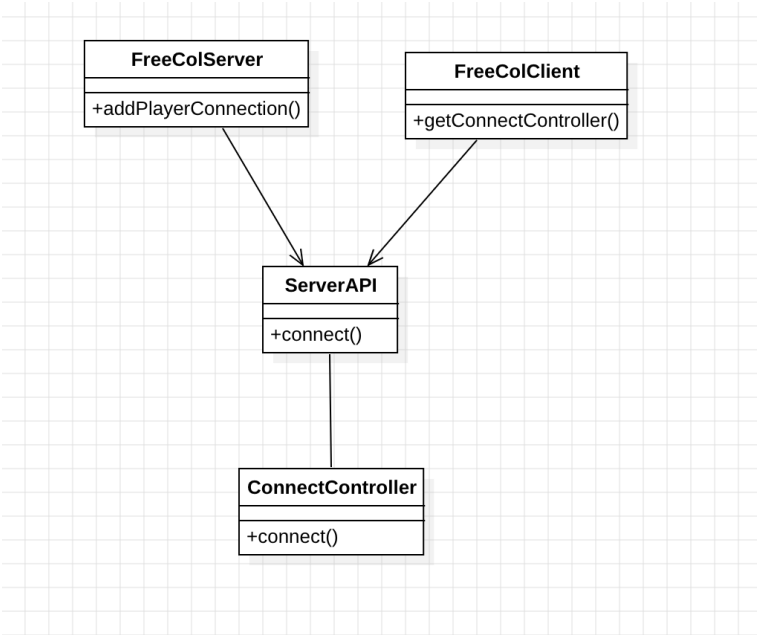
Usado para traduzir informação entre partes do projeto neste caso através da UserServerAPI é tornada possível a comunicação entre o utilizador e o servidor.



Façade Pattern

Usado para esconder a complexidade de um grupo de classes com uma classe envolvente:

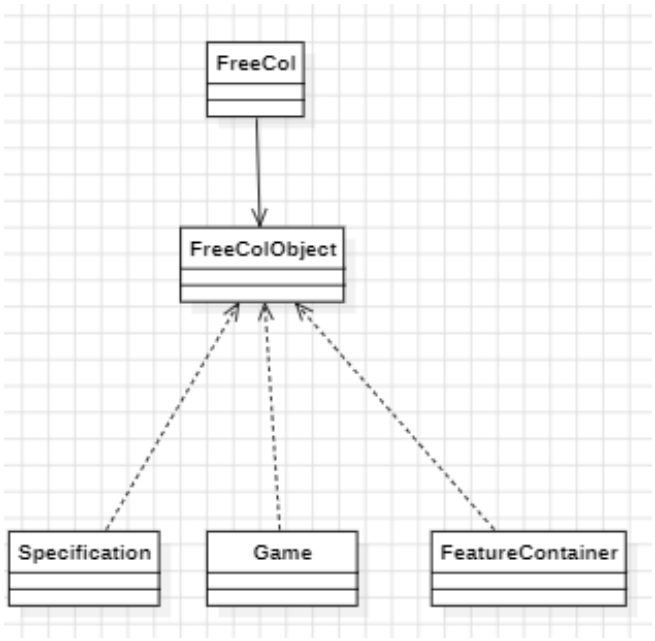
- Na classe FreeCol é utilizada a classe FreeColServer como “wrapper class” de todas as operações a nível do server.



Brás Ramos - 57800

Facade Pattern

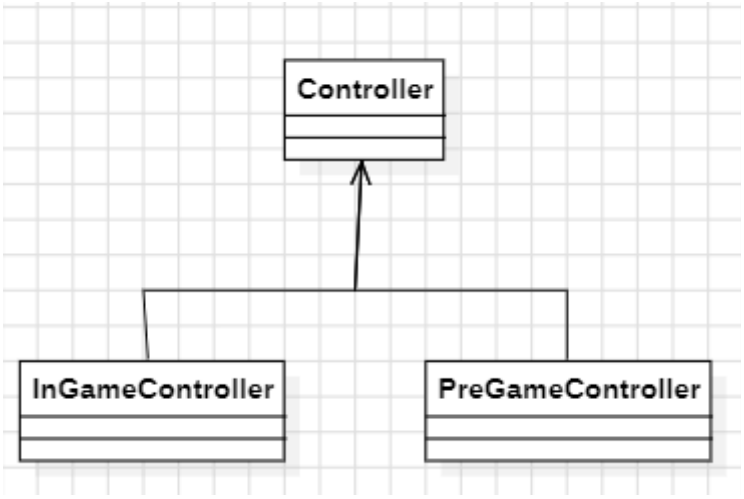
A classe FreeColObject é usada como “wrapper class”, encaplusa as outras classes de maneira a simplificar e esconder a complexidade do substistema.



Template Method Pattern

Usado para fornecer uma classe genérica que são completadas por sub classes específicas:

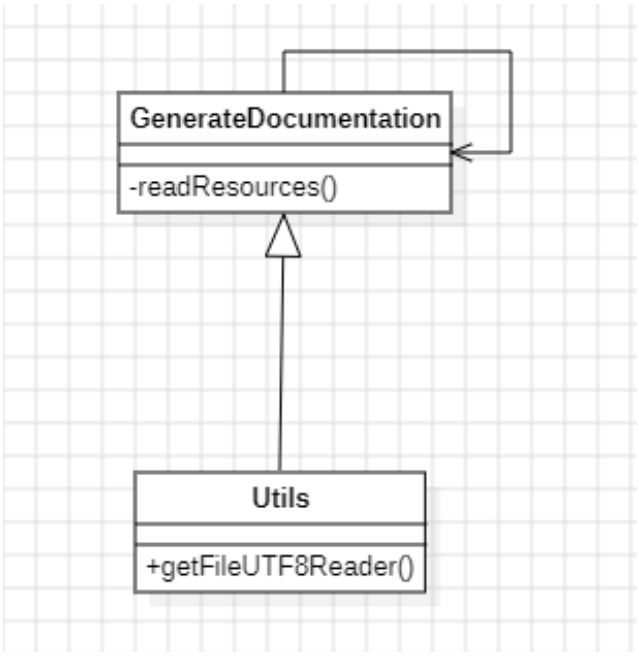
- A classe Controller, serve de base para as classes InGameController e PreGameController que são implementações específicas da mesma.



Chain of Responsability Pattern

Usado para delegar a responsabilidade de um método, por exemplo:

- na classe GenerateDocumentation no método readResources(), é feito um try/catch e passada a responsabilidade a métodos noutras classes de resolver a questão.

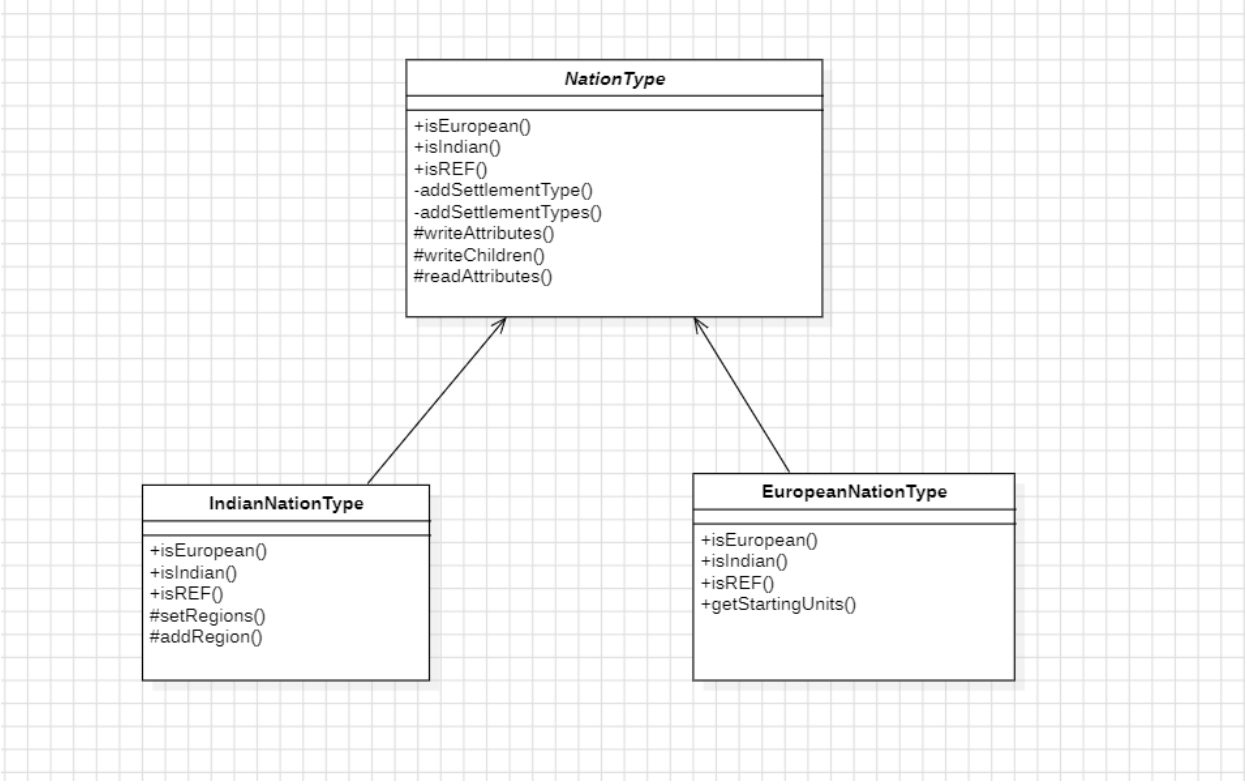


Eduardo Dias - 62466

Template Method Pattern

Usado para fornecer uma classe abstracta a duas ou mais classes, de forma a retirar redundância ao código e de forma a tornar o código mais simples.

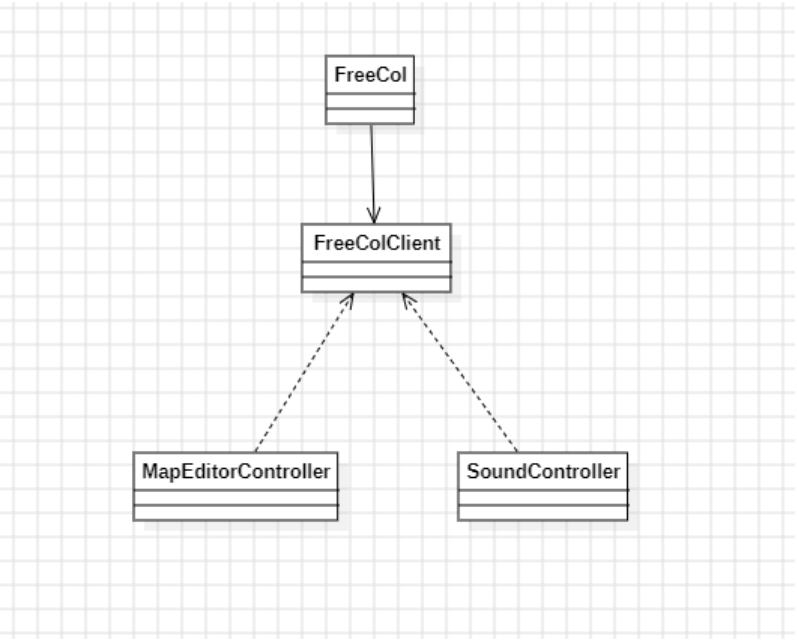
- A classe abstracta NationType possui duas classes que herdam os seus métodos, sendo estas classes IndianNationType e EuropeanNationType.



Facade Pattern

Este padrão é utilizado para simplificar/esconder um conjunto de classes de forma a simplificar todo o subsistema de classes envolventes

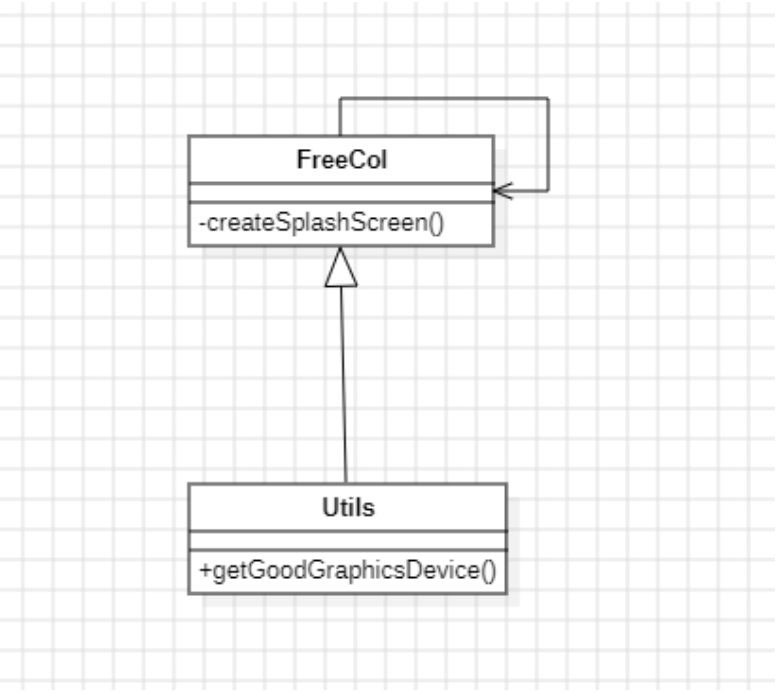
- A classe FreeCool é utilizada na classe FreeCoolClient como "*envelopamento*" de todas as operações a nível do cliente.



Chain of Responsability Pattern

Usado para delegar a responsabilidade de um método, por exemplo:

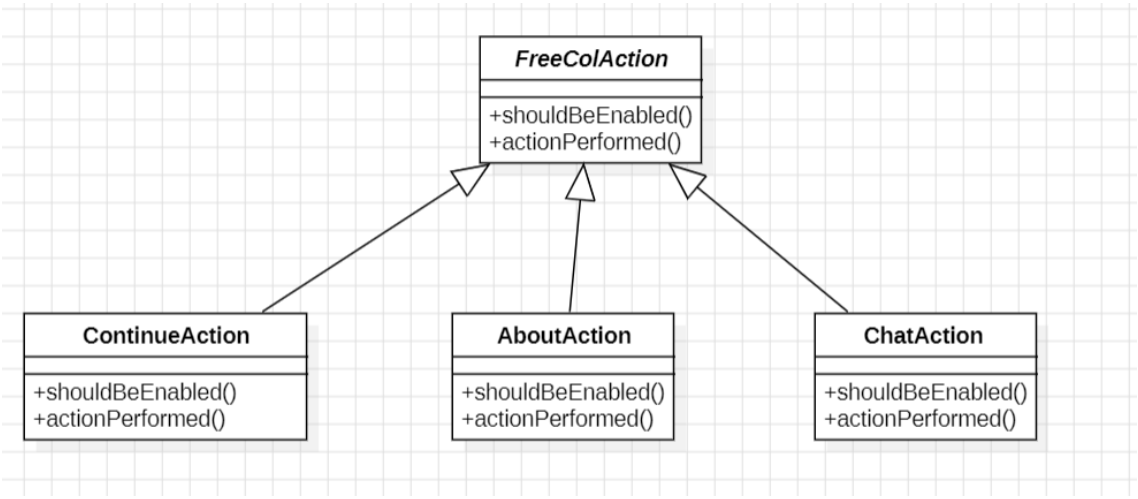
- A classe FreeCol em vários métodos como o createSplashScreen(), é feito um try/catch. Neste é passada a responsabilidade a métodos noutras classes de forma a resolver a questão do problema.



Guilherme Carvalhão - 62675

Template Pattern

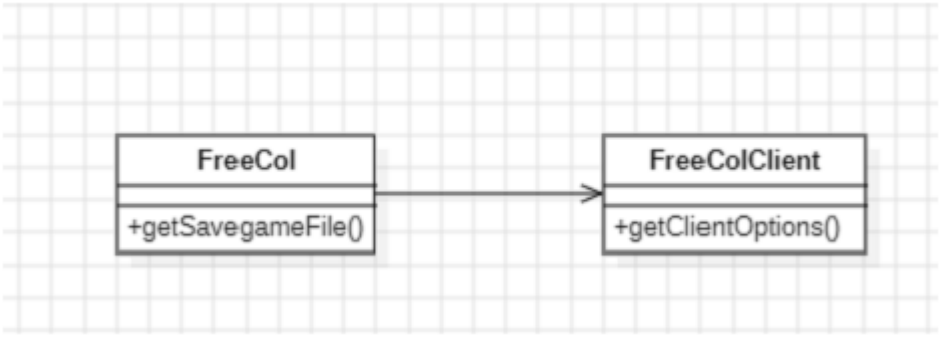
Este pattern é utilizado para podermos definir a base de um algoritmo e depois irmos acrescentando informações ao mesmo através de sub classes, por exemplo, a classe FreeColAction serve de base a várias classes como ContinueAction, AboutAction, ChatAction.



Memento Pattern

Usado para guardar o jogo e depois poder recuperá-lo numa outra altura no mesmo estado em que ficamos, ou seja se fizermos alterações depois da gravação podemos descartá-las ao sair do jogo. Isto verifica-se por exemplo:

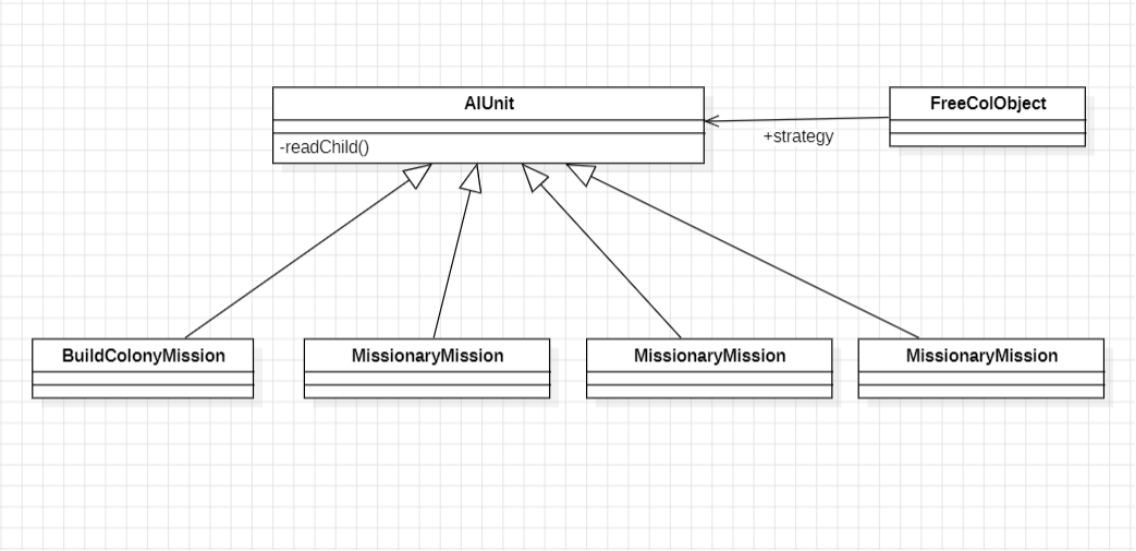
- Na classe FreeCool, é utilizada a classe FreeCoolClient para guardar as informações do jogo relativas ao cliente.



Strategy Pattern

Este pattern é utilizado para aplicar estratégias diferentes em várias situações, neste caso nas missões, por exemplo:

- Na classe AIUnit no método readChild é escolhido um tipo de missão que terá uma estratégia e dinâmica diferente.

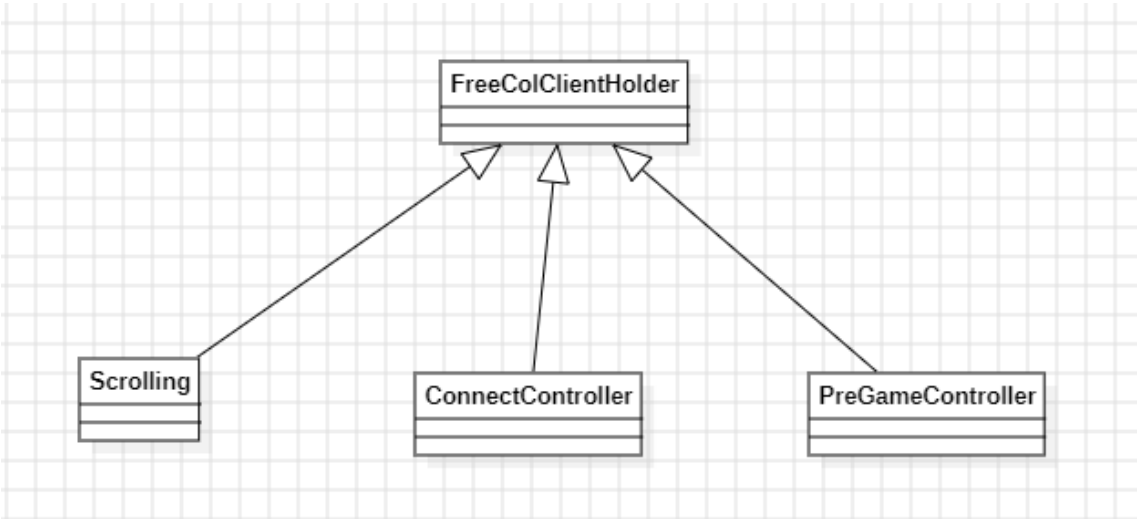


Wagna Banar - 64157

Template Method Pattern

Usado para fornecer uma interface genérica que são completadas por implementações específicas:

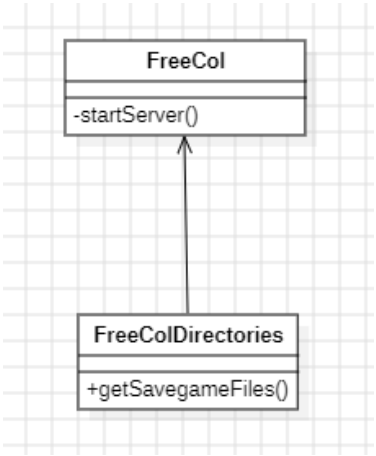
- A classe FreeColClientHolder serve de base a várias classes como Scrolling, ConnectController, PreGameController.



State Pattern

Este pattern é utilizado, por exemplo:

- Na classe FreeCol onde se utiliza a classe FreeColDirectories para guardar o estado do jogo.



Strategy Pattern

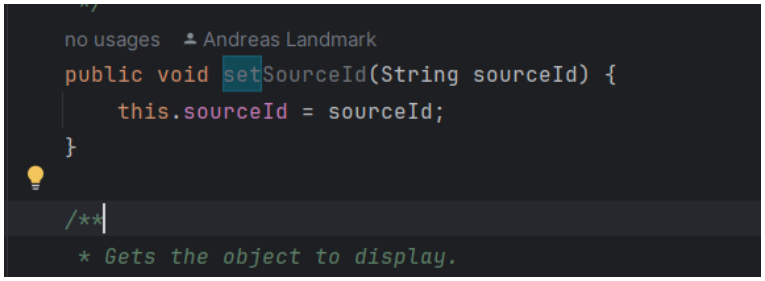
Este pattern é utilizado quando estamos em cenário de combate e são usadas as várias estratégias e etapas diferentes para lidar com essa disputa.

Code Smells

José Fernandes - 52970

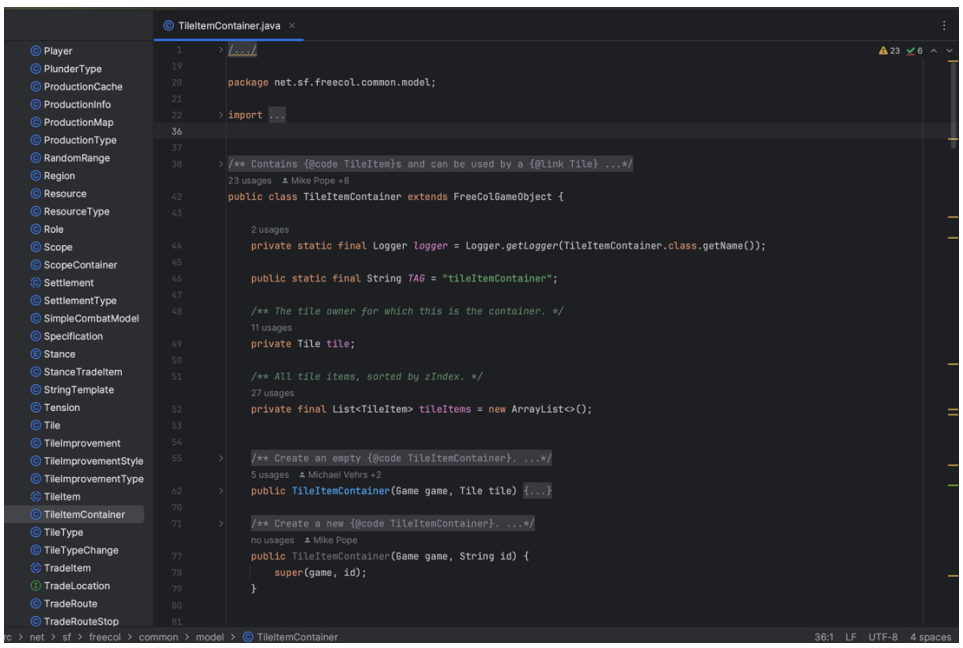
Speculative generality

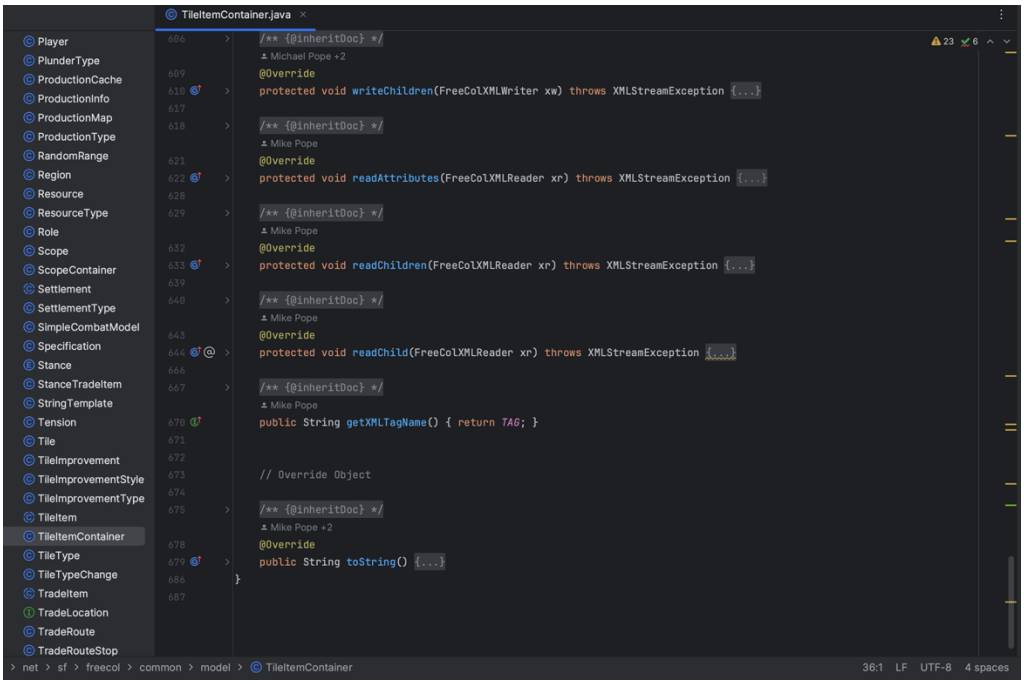
Na classe ModelMessage.java tem presente o método setSourceId que não tem nenhuma utilidade ou prepósito definido. Este método podia simplesmente ser removido ou então documentar sua finalidade e possíveis casos de uso futuro para esclarecer sua presença.



Large class

A classe TileItemContainer.java é muito extensa, sendo por isso muito complexa e difícil entender. Esta deveria ser reduzida, reavaliando que partes do código poderiam ser refeitas ou mesmo retiradas. Para ter uma leitura mais fácil, também poderiam ser adicionados mais comentários.





Excessive Coupling:

a classe LandMap.java , no método generate mantem muitas responsabilidades pelos seus extensos casos de switch-case. Esta responsabilidade deveria estar distribuída por outras classes ou por outros métodos podendo usar uma abordagem mais estruturada, como o factory pattern, para determinar o algoritmo de geração de mapa com base no tipo fornecido.

```
private final void generate(int type, int distanceToEdge,
                            int minNumberOfTiles) {
    switch (type) {
    case MapGeneratorOptions.LAND_GENERATOR_CLASSIC:
        createClassicLandMap(distanceToEdge, minNumberOfTiles);
        break;
    case MapGeneratorOptions.LAND_GENERATOR_CONTINENT:
        // Create one landmass of 75%, start it somewhere near the
        // center, then fill up with small islands.
        addPolarRegions();
        int contsize = (minNumberOfTiles * 75) / 100;
        int height = getHeight()/4 + this.cache.nextInt( tighterRange: getHeight()/2);
        addLandMass(contsize, contsize,  x: getWidth()/2, height,
                    distanceToEdge);
        while (this.numberOfLandTiles < minNumberOfTiles) {
            addLandMass( minSize: 15,  maxSize: 25,  x: -1,  y: -1, distanceToEdge);
        }
        break;
    case MapGeneratorOptions.LAND_GENERATOR_ARCHIPELAGO:
        // Create 5 islands of 10% each, then delegate to the Islands
        // generator.
        addPolarRegions();
        int archsize = (minNumberOfTiles * 10) / 100;
        for (int i = 0; i < 5; i++) {
            addLandMass( minSize: archsize - 5,  maxSize: archsize + 5,  x: -1,  y: -1,
                        distanceToEdge);
        }
        // Fall through
    case MapGeneratorOptions.LAND_GENERATOR_ISLANDS:
        // Create islands of 25..75 tiles.
        addPolarRegions();
```

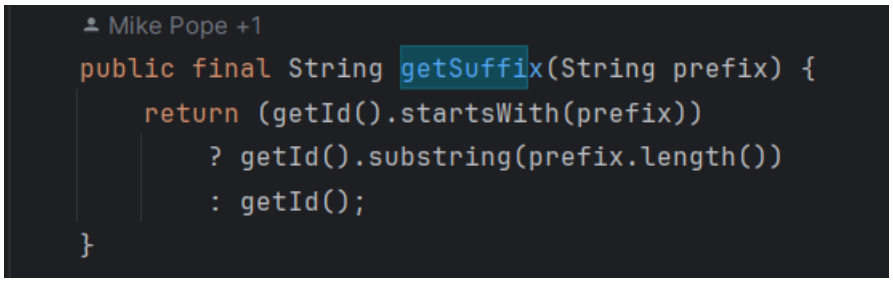
Brás Ramos - 57800

Large class

Algumas classes como a Colony.java são muito extensas. Esta classe possui muitas responsabilidades e muitos métodos. Além disso esta classe possui enumerados muito extensos que tornam todo o código muito mais extenso. Ao criar uma class Enum, poderíamos reduzir bastante o código desta classe. Esta classe possui 3268 linhas o que torna a sua leitura bastante difícil de ser analisada.

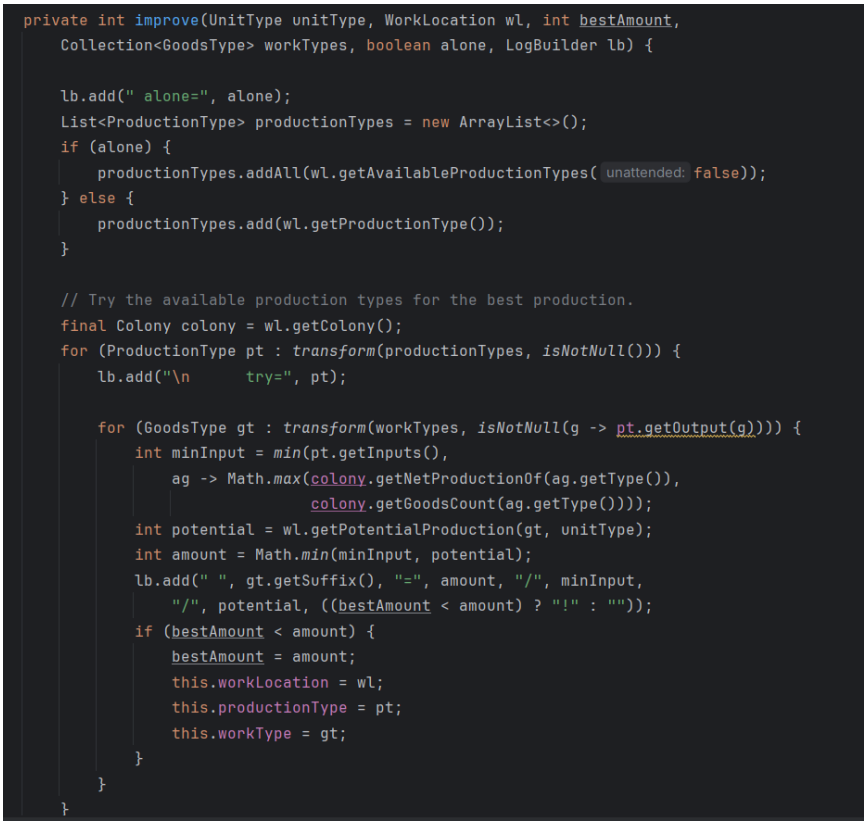
Speculative generality

A classe FreeColObject.java possui muitos métodos que não são utilizados, como por exemplo, o método getSuffix. Poderíamos remover este método.



Long method

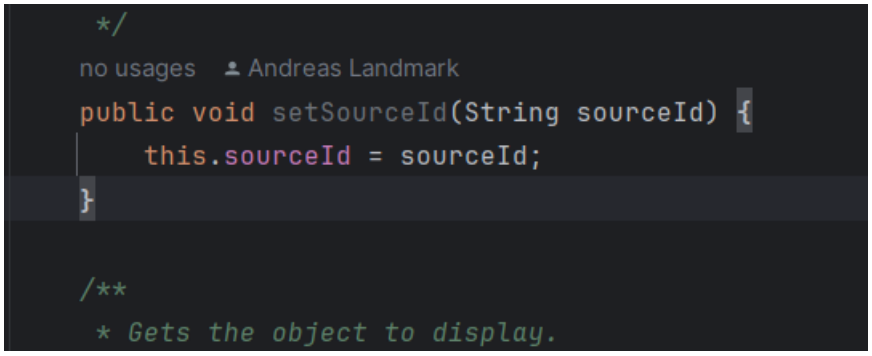
A classe Occupation.java possui o método improve que para além de possuir muitos parâmetros, é muito extensa e complexa. Possui muitos casos if-else e alguns ciclos 'for' muito extensos



Eduardo Dias - 62466

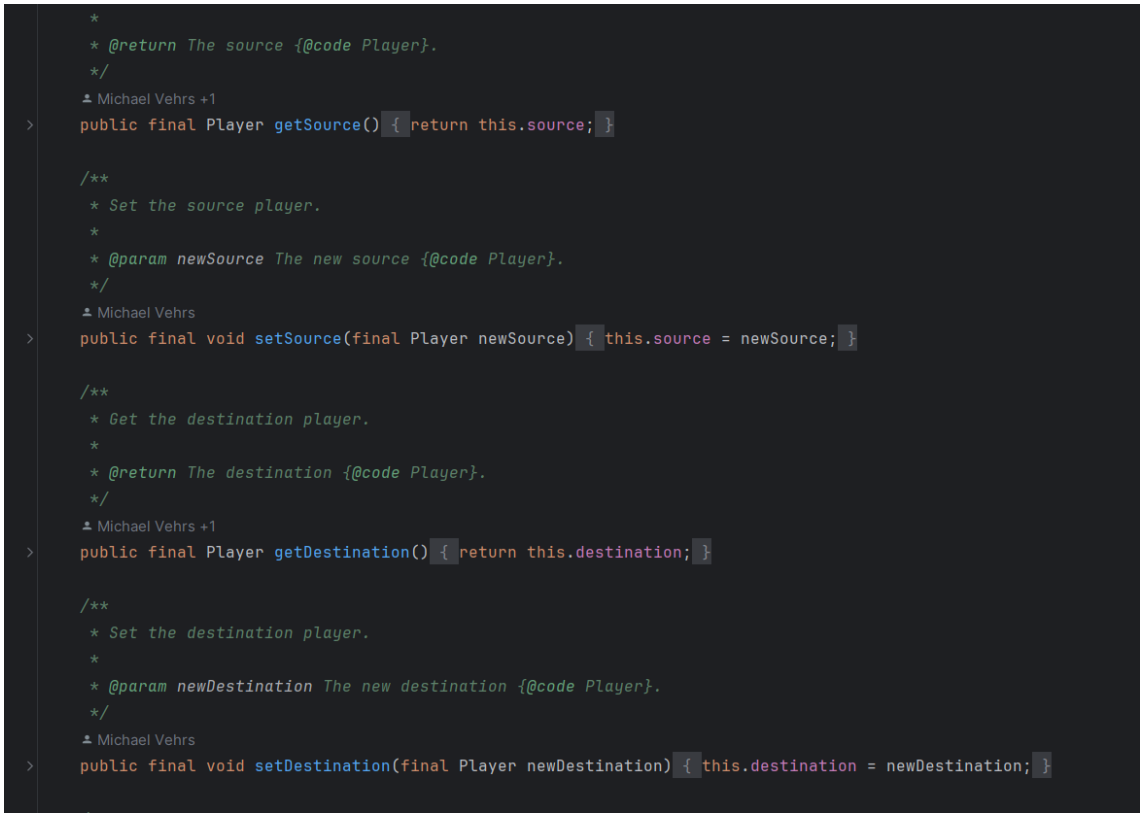
Speculative generality

A classe ModelMessage.java tem presente o método setSourceId que não tem nenhum utilidade, apenas aquele do “pode ser que riva para algo um dia”. Podíamos remover este método o que faria com que o código ficasse mais curto e menos confuso.



Data Class

A classe TradeItem.java é praticamente composta apenas por getters e setters. Claro que se esta for uma classe com um propósito simples podemos aceitar que seja assim composta. No Entanto considero que devesse ser reavaliada de forma a verificar se não poderíamos colocar mais métodos que acrescentasse mais funcionalidades a esta.



Long method

A classe UnitWas.java possui o método fireChanges que para além de ser muito extenso, possui muitas condições de if-else que tornam o código muito difícil de ser analisado. Podíamos reduzir este método fazendo mais métodos privados dentro da classe que fossem chamados pelo método fireChanges.

```
2 usages  Mike Pope +3
public boolean fireChanges() {
    UnitType newType = null;
    Role newRole = null;
    int newRoleCount = 0;
    Location newLoc = null;
    GoodsType newWork = null;
    int newWorkAmount = 0;
    int newMovesLeft = 0;
    boolean ret = false;
    if (!unit.isDisposed()) {
        newLoc = unit.getLocation();
        if (colony != null) {
            newType = unit.getType();
            newRole = unit.getRole();
            newRoleCount = unit.getRoleCount();
            newWork = unit.getWorkType();
            newWorkAmount = (newWork == null) ? 0
                : getAmount(newLoc, newWork);
        }
        newMovesLeft = unit.getMovesLeft();
    }

    FreeColGameObject oldFcgo = (FreeColGameObject)loc;
    FreeColGameObject newFcgo = (FreeColGameObject)newLoc;
    if (loc != newLoc) {
        oldFcgo.firePropertyChange(change(oldFcgo), unit,  newValue: null);
        if (newLoc != null) {
        newFcgo.firePropertyChange(change(newFcgo),  oldValue: null, unit);
    }
    ret = true;
}
if (colony != null) {
    if (type != newType && newType != null) {
        String pc = ColonyChangeEvent.UNIT_TYPE_CHANGE.toString();
        colony.firePropertyChange(pc, type, newType);
        ret = true;
    } else if (role != newRole && newRole != null) {
        String pc = Tile.UNIT_CHANGE;
        colony.firePropertyChange(pc, role.toString(),
                                  newRole.toString());
        ret = true;
    }
    if (work != newWork) {
        if (work != null && oldFcgo != null && workAmount != 0) {
            oldFcgo.firePropertyChange(work.getId(), workAmount,  newValue: 0);
        }
        if (newWork != null && newFcgo != null && newWorkAmount != 0) {
            newFcgo.firePropertyChange(newWork.getId(),
                                       oldValue: 0, newWorkAmount);
        }
        ret = true;
    } else if (workAmount != newWorkAmount) {
        newFcgo.firePropertyChange(newWork.getId(),
                                   workAmount, newWorkAmount);
        ret = true;
    }
}
```

```
            ret = true;
        } else if (workAmount != newWorkAmount) {
            newFcgo.firePropertyChange(newWork.getId(),
                                       workAmount, newWorkAmount);
            ret = true;
        }
    }
    if (role != newRole && newRole != null) {
        unit.firePropertyChange(Unit.ROLE_CHANGE, role, newRole);
        ret = true;
    } else if (roleCount != newRoleCount && newRoleCount >= 0) {
        unit.firePropertyChange(Unit.ROLE_CHANGE, roleCount, newRoleCount);
        ret = true;
    }
    if (unit.getGoodsContainer() != null) {
        ret |= unit.getGoodsContainer().fireChanges();
    }
    if (!units.equals(unit.getUnitList())) {
        unit.firePropertyChange(Unit.CARGO_CHANGE,  oldValue: null, unit);
        ret = true;
    }
    if (movesLeft != newMovesLeft) {
        unit.firePropertyChange(Unit.MOVE_CHANGE, movesLeft, newMovesLeft);
        ret = true;
    }
    return ret;
}
```

Data Clumps

Podemos ainda observar que muitas classes possuem métodos hashCode,equals e to String. Estes métodos poderiam facilmente ser substituídos por classes que tornariam muitas classes bem mais simples, visto estes serem métodos que são repetidos ao longo de varias classes. Para tal poderíamos criar classes como “equals.java” ,“hashCode.java” e “toString.java”

```
public boolean equals(Object o) {
    if (this == o) return true;
    if (o instanceof Ability) {
        Ability other = (Ability)o;
        return this.value == other.value
            && super.equals(other);
    }
    return false;
}

/**
 * {@inheritDoc}
 */
 Mike Pope
@Override
public int hashCode() {
    int hash = super.hashCode();
    hash += (value) ? 1 : 0;
    return hash;
}

/**
 * {@inheritDoc}
 */
 Mike Pope +1
@Override
public String toString() {
    StringBuilder sb = new StringBuilder( capacity: 32);
    sb.append("[ ").append(getId());
    if (getSource() != null) {
        sb.append(" (").append(getSource().getId()).append(')');
    }
```

Guilherme Carvalhão - 62675

Long method

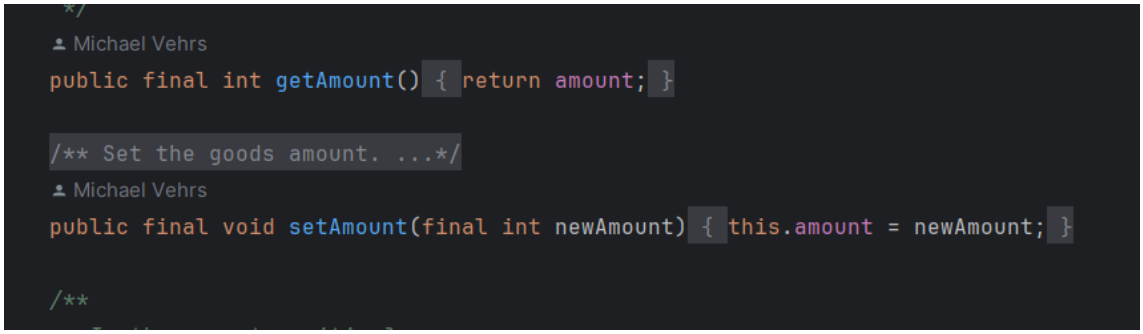
A classe DiplomaticTrade.java tem alguns métodos muito extensos, como é o caso do método toString ou readChild que para além de serem métodos muito extensos e complexos, são de difícil leitura. O método readChild por exemplo possui demasiadas condições de if-else que poderiam ser substituídas por casos switch.

```
@Override
protected void readChild(FreeColXMLReader xr) throws XMLStreamException {
    final String tag = xr.getLocalName();

    if (ColonyTradeItem.TAG.equals(tag)) {
        add(new ColonyTradeItem(getGame(), xr));

    } else if (GoldTradeItem.TAG.equals(tag)) {
        add(new GoldTradeItem(getGame(), xr));

    } else if (GoodsTradeItem.TAG.equals(tag)) {
        add(new GoodsTradeItem(getGame(), xr));

    } else if (InciteTradeItem.TAG.equals(tag)) {
        add(new InciteTradeItem(getGame(), xr));

    } else if (StanceTradeItem.TAG.equals(tag)) {
        add(new StanceTradeItem(getGame(), xr));

    } else if (UnitTradeItem.TAG.equals(tag)) {
        add(new UnitTradeItem(getGame(), xr));

    } else {
        super.readChild(xr);
    }
}
```

Data Class

A classe AbstractGoods.java, etc. é composta basicamente só por métodos getters e setters. Poderíamos adicionar mais métodos de forma a tornar a classe mais funcional e prática.

```
 */
Mike Pope
public final GoodsType getType() { return type; }

/**
 * Set the goods type.
 *
 * @param newType The new {@code GoodsType}.
 */
Michael Vehrs
public final void setType(final GoodsType newType) { this.type = newType; }
```

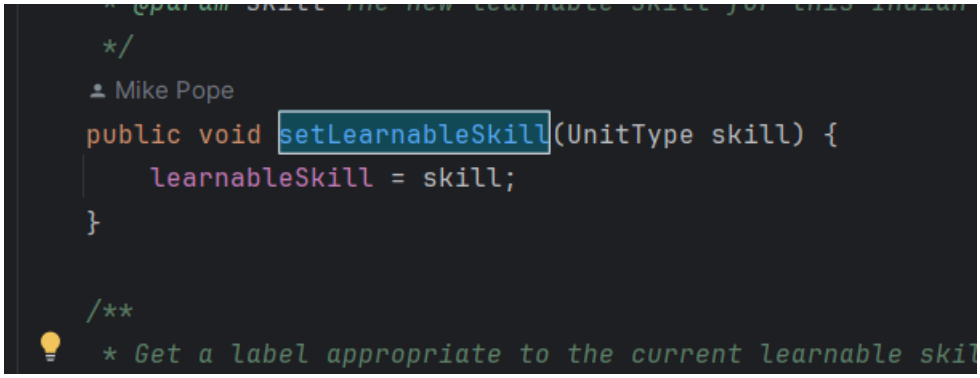


```
 */
Michael Vehrs
public final int getAmount() { return amount; }

/** Set the goods amount. ...*/
Michael Vehrs
public final void setAmount(final int newAmount) { this.amount = newAmount; }

/**
```

```
public StringTemplate getLabel() {
    return StringTemplate.template( value: "model.abstractGoods.label")
        .addNamed( key: "%goods%", getType())
        .addAmount( key: "%amount%", getAmount());
}

/**
 * Get a label for these goods.
 *
 * @param sellable Whether these goods can be sold.
 * @return A label for these goods.
 */
Mike Pope +1
public StringTemplate getLabel(boolean sellable) {
    return (sellable) ? getLabel()
        : StringTemplate.template( value: "model.abstractGoods.boycotted")
            .addNamed( key: "%goods%", getType())
            .addAmount( key: "%amount%", getAmount());
}
```

Speculative generality

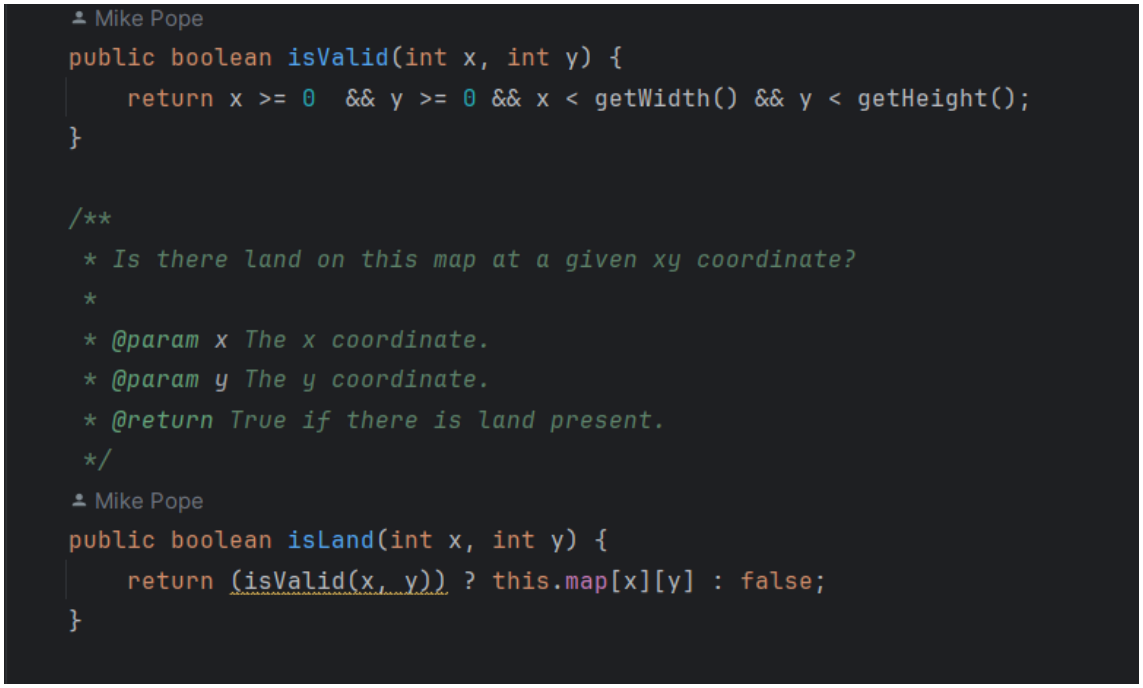
A classe IndianSettlement.java tem presente alguns métodos que não são utilizados em lado nenhum, tal como o setLearnableSkill. Poderiamos remover este método de forma a que a classe fica-se mais simples e menos confusa. Ou então dar alguma utilidade para este método.



Wagna Banar - 64157

Speculative generality

Na classe LandMap.java tem alguns métodos que não têm nenhuma utilidade como é o caso do método isValid ou isLand que não são utilizadas em lado nenhum.
Esses métodos possuem a ideia de que serão utilizados mais tarde, nesse caso podíamos remover esse código de forma a que o código da classe ficava mais legível.



Long method

Na classe LandMap.java temos presente métodos muito longos e difíceis de entender como o addLandMass. Estes métodos tornam o código muito extenso e difícil de perceber. Para corrigir isto devíamos reduzir código removendo por exemplo redundância.

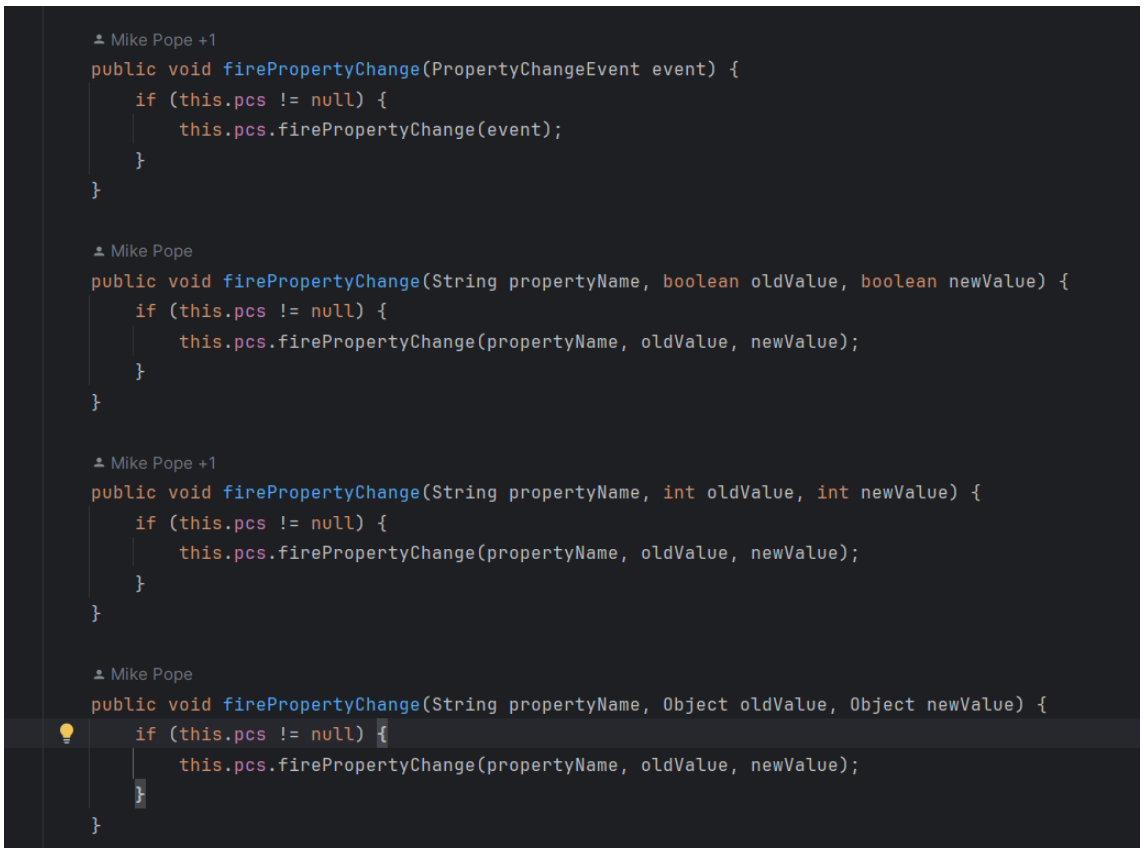
```
        List<Position> l = newPositions(p, distanceToEdge);

        // Get a random position from the list,
        // set it to land,
        // add its valid neighbours to the list
        int enough = minSize + this.cache.nextInt( tighterRange: maxSize - minSize + 1);
        while (size < enough && !l.isEmpty()) {
            int i = this.cache.nextInt(l.size());
            p = l.remove(i);

            if (!newLand[p.getX()][p.getY()]) {
                newLand[p.getX()][p.getY()] = true;
                size++;
                l.addAll(newPositions(p, distanceToEdge));
            }
        }

        // Add generated land to map if sufficiently large
        if (size >= minSize) {
            for (x = 0; x < this.width; x++) {
                for (y = 0; y < this.height; y++) {
                    if (newLand[x][y]) setLand(x, y);
                }
            }
        }
        return (size >= minSize) ? size : 0;
    }
}
```

```
private int addLandMass(int minSize, int maxSize, int x, int y,
                        int distanceToEdge) {
    int size = 0;
    boolean[][] newLand = new boolean[getWidth()][getHeight()];

    // Pick a starting position that is sea without neighbouring land.
    if (x < 0 || y < 0) {
        final int wid = getWidth() - distanceToEdge * 2;
        final int hgt = getHeight() - distanceToEdge * 2;
        do {
            x = distanceToEdge + this.cache.nextInt(wid);
            y = distanceToEdge + this.cache.nextInt(hgt);
        } while (isLand(x, y) || hasAdjacentLand(x, y));
    }

    newLand[x][y] = true;
    size++;

    // Add all valid neighbour positions to list
    Position p = new Position(x, y);
    List<Position> l = newPositions(p, distanceToEdge);

    // Get a random position from the list,
    // set it to land,
    // add its valid neighbours to the list
    int enough = minSize + this.cache.nextInt( tighterRange: maxSize - minSize + 1);
```

Duplicated code

Na classe FreeColObject.java tem muito código repetido, métodos que são praticamente idênticos e que apenas alteram as variáveis dos parâmetros.

Para corrigir isso poderíamos remover esses códigos, e o métodos nelas serem usados diretamente , ou então refatora-los redefini-los com todos os parâmetros que ela recebe.



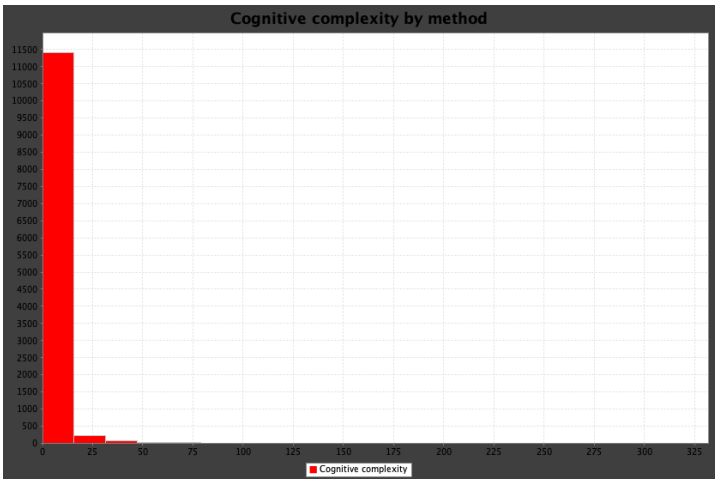
Metrics

José Fernandes - 52970

Complexity Metrics

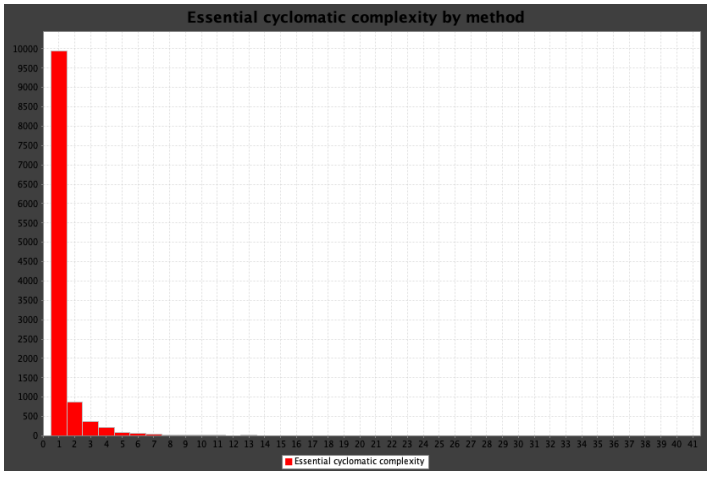
- Cognitive Complexity (CogC)

Consideração de vários fatores, como declarações, loops, entre outros, avaliando o quão difícil é para um ser humano entender um trecho de código.
Valores elevados sugerem que um método pode ser difícil de entender (gráfico demonstra que seria fácil de entender).



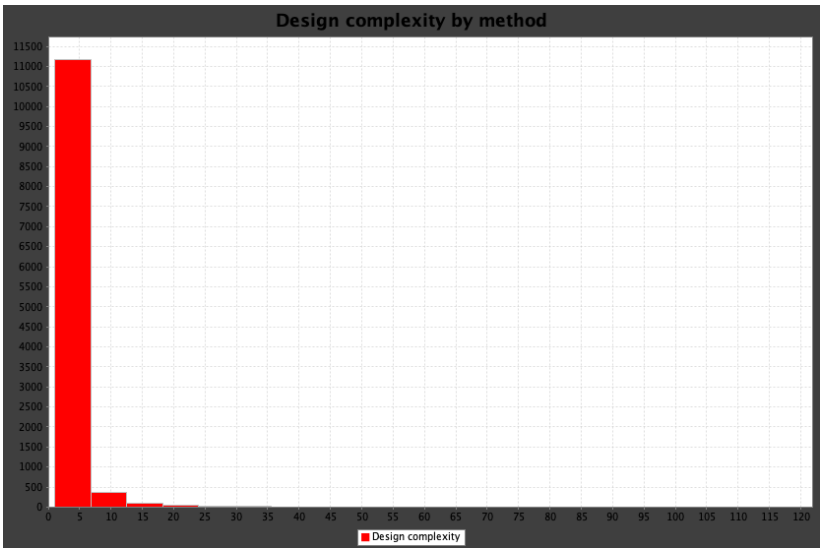
- Essential Cyclomatic Complexity (ev(G))

Baseada em estruturas de fluxo de controle (declarações, loops e saltos), medindo o nº de caminhos linearmente independentes por método.
Valores elevados indicam que pode ser mais difícil de testar (gráfico demonstra que seria fácil de testar).



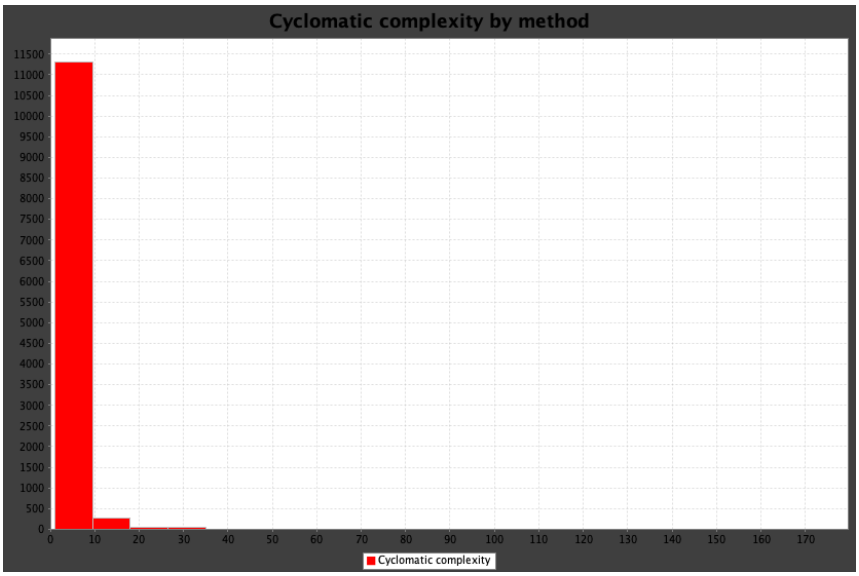
- Design Complexity (iv(G))

Considera os dados que são lidos, modificados e usados ao longo dos métodos, medindo a complexidade dos fluxos de um método (chamadas de outros métodos).
Valores elevados podem indicar que o método está realizando muitas manipulações de dados, podendo tornando-o difícil de entender (gráfico demonstra que seria fácil de entender).



- Cyclomatic Complexity (v(G))

Calcula a complexidade cíclica de non-abstract methods, medindo o nº de caminhos executados por cada método. Também considera o nº mínimo de testes necessários para exercitar completamente o fluxo de controle de um método, isto é:
1 + (nº de: if's, while's, for's, do's, switch cases, catches, expressões condicionais, &&'s e ||'s no método)
(gráfico demonstra que os métodos teriam uma baixa complexidade ciclica).



Em resumo, o código possui algumas métricas de complexidade muito elevadas, o que indica que pode ser difícil de testar e entender. CogC , ev(G), iv(G)) e v(G) geralmente são indicadores de código que pode necessitar de refatoração e simplificação para melhorar a legibilidade, a manutenibilidade e a qualidade geral do código. Reduzir a complexidade e dividir métodos grandes em métodos menores e mais focados pode ajudar a tornar a base de código mais gerenciável e menos sujeita a erros.

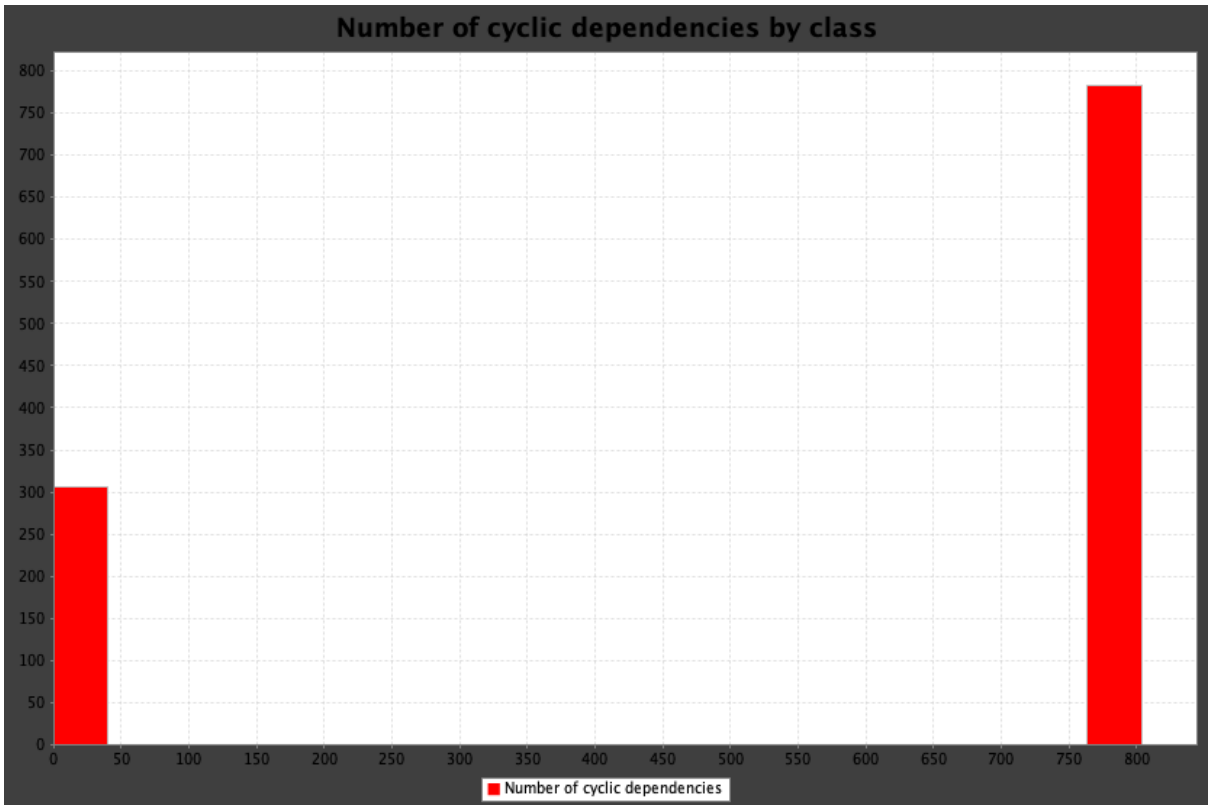
A aplicação de design patterns podem ajuda a melhorar a organização e a estrutura do código.

Brás Ramos – 57800

Dependency-Metrics

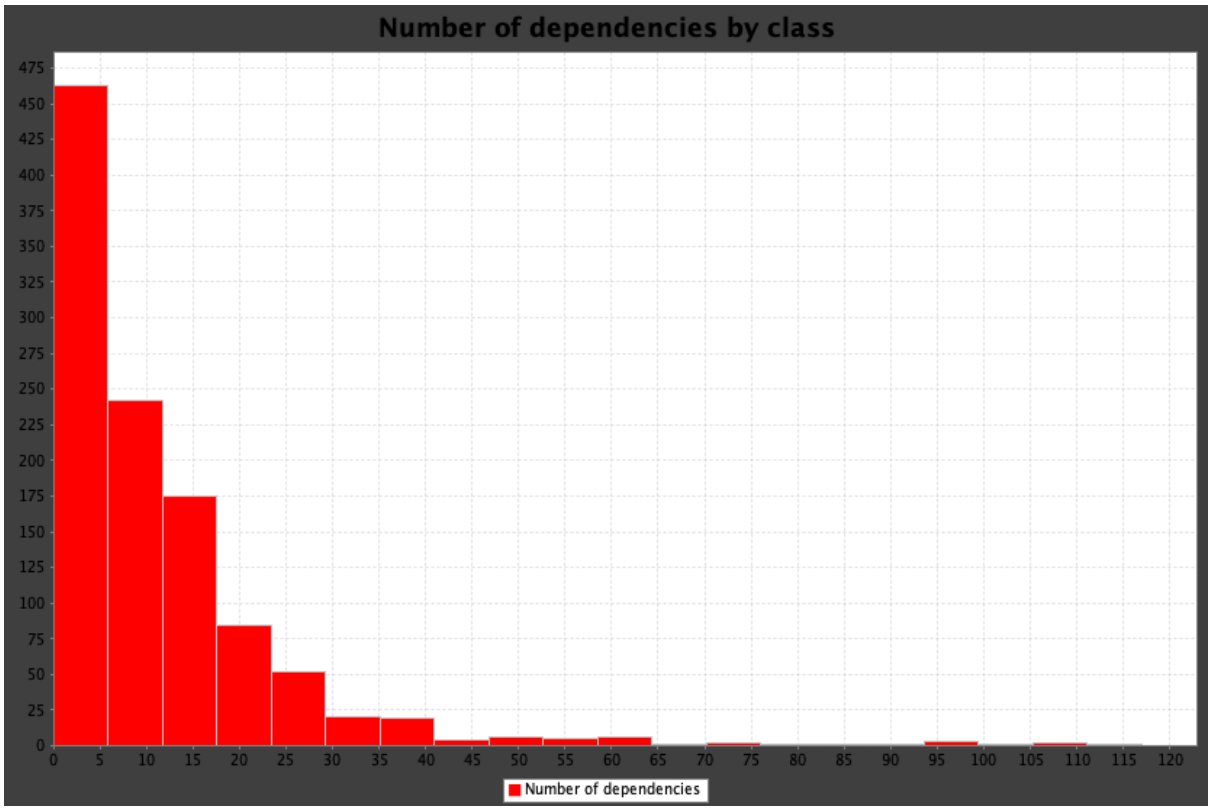
- Cyclic

O gráfico mostra o número de dependências cíclicas por classe no projeto. Dependências cíclicas são situações em que duas ou mais classes se referenciam mutuamente, o que pode causar problemas de design e manutenção.
O gráfico indica que há uma classe com um número muito alto de dependências cíclicas (cerca de 750), o que sugere que essa classe é muito complexa.
Uma boa prática de engenharia de software é minimizar o número de dependências cíclicas por classe.



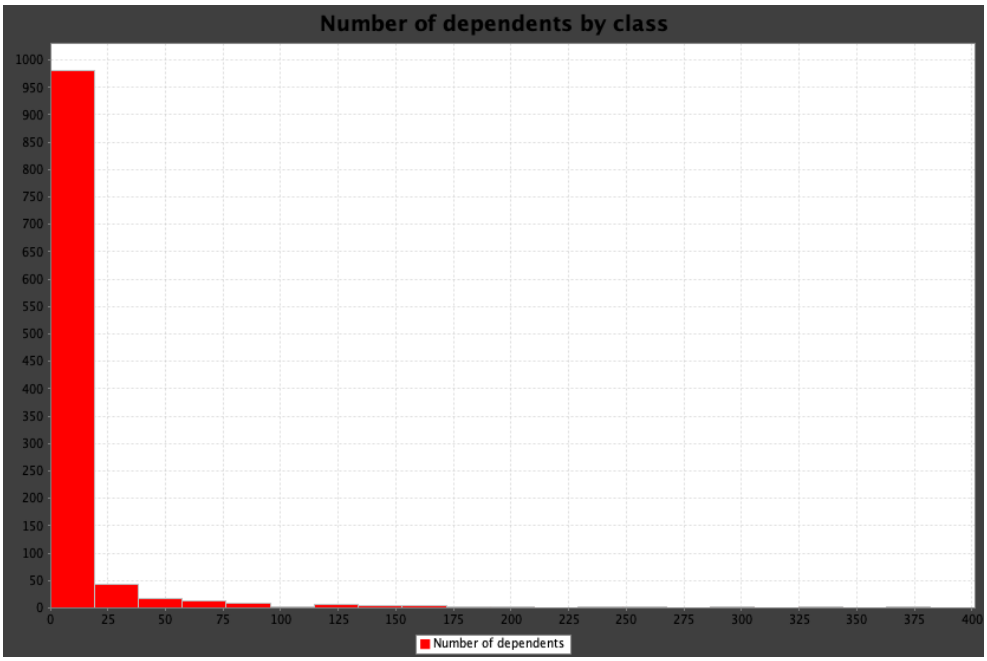
- Dcy

O gráfico mostra o número de dependências por classe no projeto.
O gráfico indica que há uma grande variação no número de dependências por classe, desde 0 até 475.
O gráfico sugere que algumas classes são muito dependentes de outras classes, o que pode indicar um alto acoplamento e uma baixa coesão no código.
Um alto acoplamento significa que as classes são muito interligadas e difíceis de modificar ou reutilizar. Uma baixa coesão significa que as classes têm muitas responsabilidades e não estão bem organizadas.



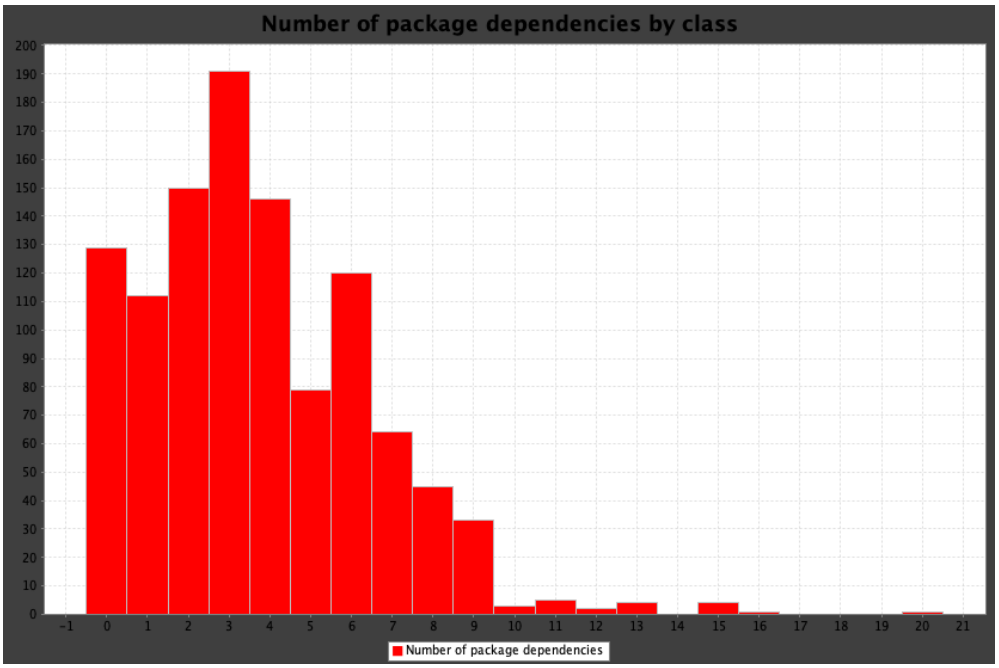
- Dpt

O gráfico mostra o número de dependentes por classe em um projeto de software.
Dependentes são classes que usam ou dependem de outra classe para funcionar.
O gráfico indica que há uma classe com um número muito baixo de dependentes (entre 0 e 25), o que sugere que essa classe é muito isolada e pouco reutilizada.
O gráfico também indica que há uma tendência de diminuição no número de dependentes por classe à medida que o número de dependentes aumenta, o que pode indicar uma distribuição desigual da complexidade e da funcionalidade entre as classes.



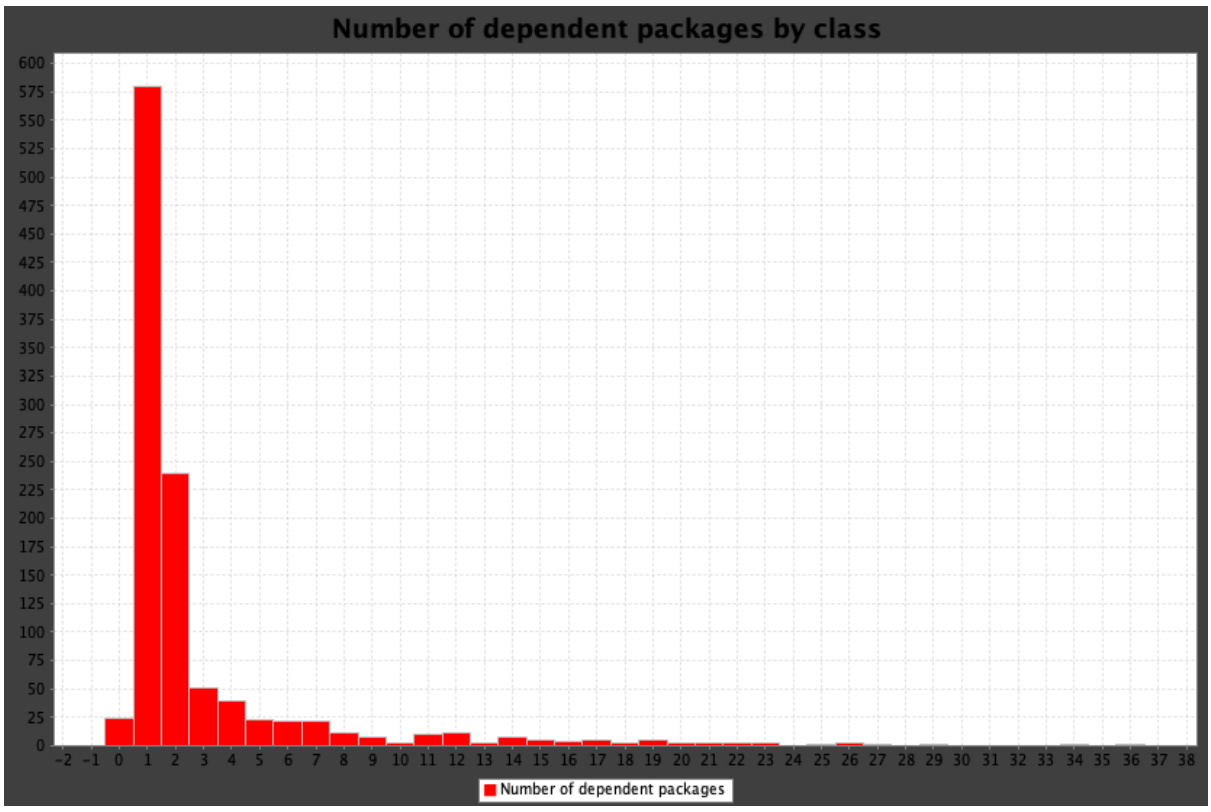
- PDcy

O gráfico mostra o número de dependências de pacote por classe no projeto. Dependências de pacote são relações entre classes que pertencem a diferentes pacotes ou módulos.
O gráfico indica que as classes 3 e 2 têm o maior número de dependências de pacote (cerca de 190 e 150 respectivamente), o que sugere que essas classes são muito acopladas a outros pacotes e podem ter problemas de modularidade e reusabilidade.
O gráfico também indica que algumas classes não têm nenhuma dependência de pacote, o que pode indicar que essas classes são muito isoladas e pouco integradas com outros pacotes.



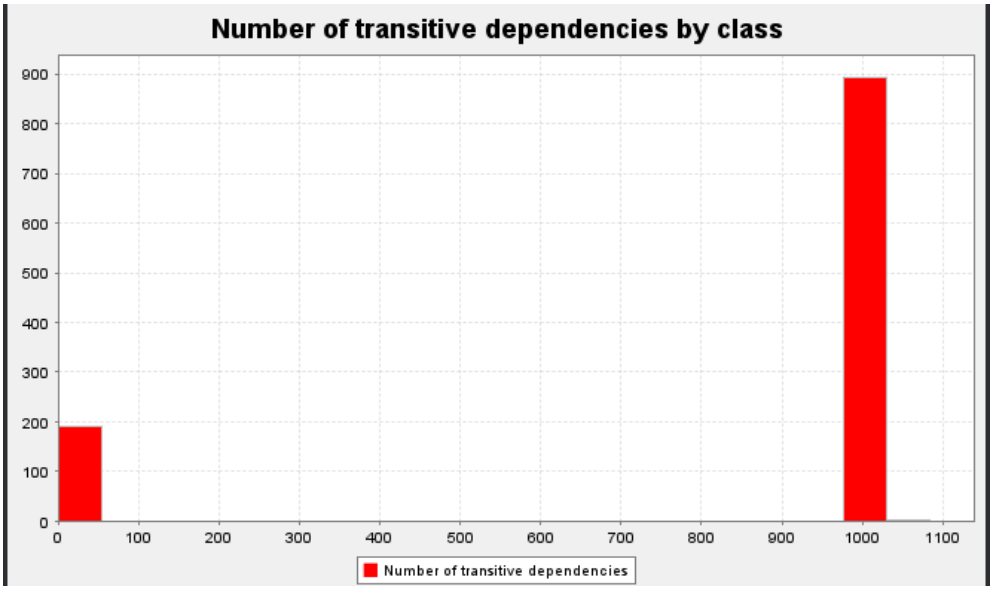
- PDpt

O gráfico mostra o número de pacotes dependentes por classe no projeto.
Pacotes dependentes são pacotes ou módulos de software que usam ou dependem de outro pacote para funcionar.
O gráfico indica que as classes 1 e 2 têm o maior número de pacotes dependentes (mais de 200 cada), o que sugere que essas classes são muito reutilizadas e integradas com outros pacotes.
O gráfico diz que o número de pacotes dependentes por classe diminui à medida que o número da classe aumenta, o que pode ser resultado de uma distribuição desigual da complexidade e da funcionalidade entre os pacotes.



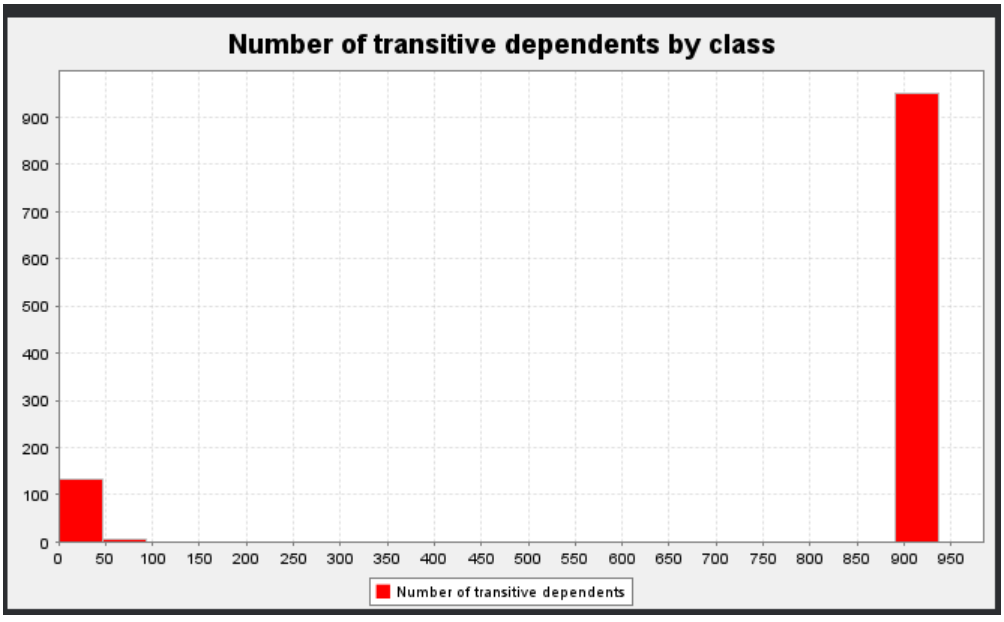
- Dcy*

O gráfico mostra o número de dependências transitivas por classe em um projeto de software.
Dependências transitivas são relações indiretas entre classes que ocorrem quando uma classe depende de outra classe que depende de uma terceira classe, e assim por diante.
O gráfico indica que há duas classes com um número muito alto de dependências transitivas (cerca de 190 e 900 respectivamente), o que sugere que essas classes são muito acopladas a outras classes e podem ter problemas de modularidade e reusabilidade.



-Dpt*

O gráfico mostra o número de dependentes transitivos por classe em um projeto de software.
Dependentes transitivos são classes que usam ou dependem indiretamente de outra classe, através de uma cadeia de dependências diretas.
O gráfico indica que há uma classe com um número muito alto de dependentes transitivos (cerca de 950), o que sugere que essa classe é muito reutilizada e integrada com outras classes.
O gráfico também indica que o número de dependentes transitivos por classe diminui à medida que o número da classe aumenta, o que pode indicar uma distribuição desigual da complexidade e da funcionalidade entre as classes.



Eduardo Dias – 62466

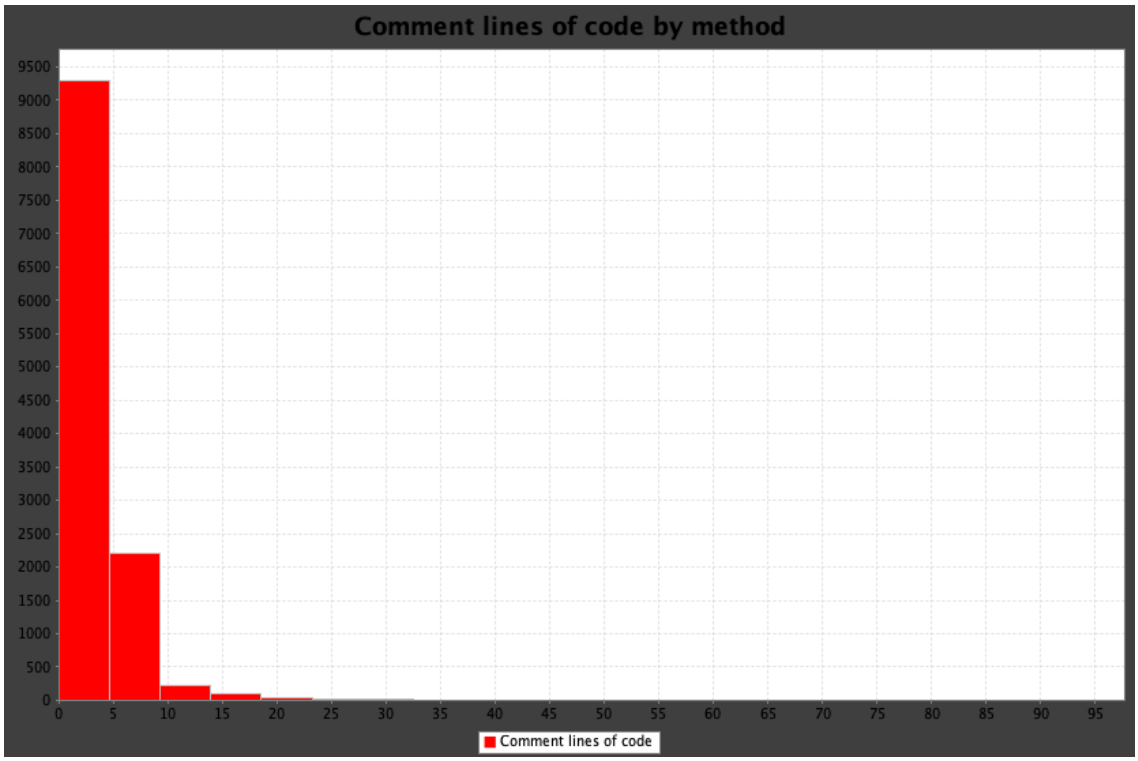
LinesOfCode Metrics

-CLOC

Calcula o número de linhas de comentários em cada método.

Não são contabilizados espaços em branco. Comentários separados por um 'Enter' são contabilizados como uma única linha de comentário.
Um alinha que contenha código e comentário é contabilizado como uma única linha de código. Comentários em classes anonimas e locais não são contados separadamente mas estão incluídos na contagem do método que os contém.

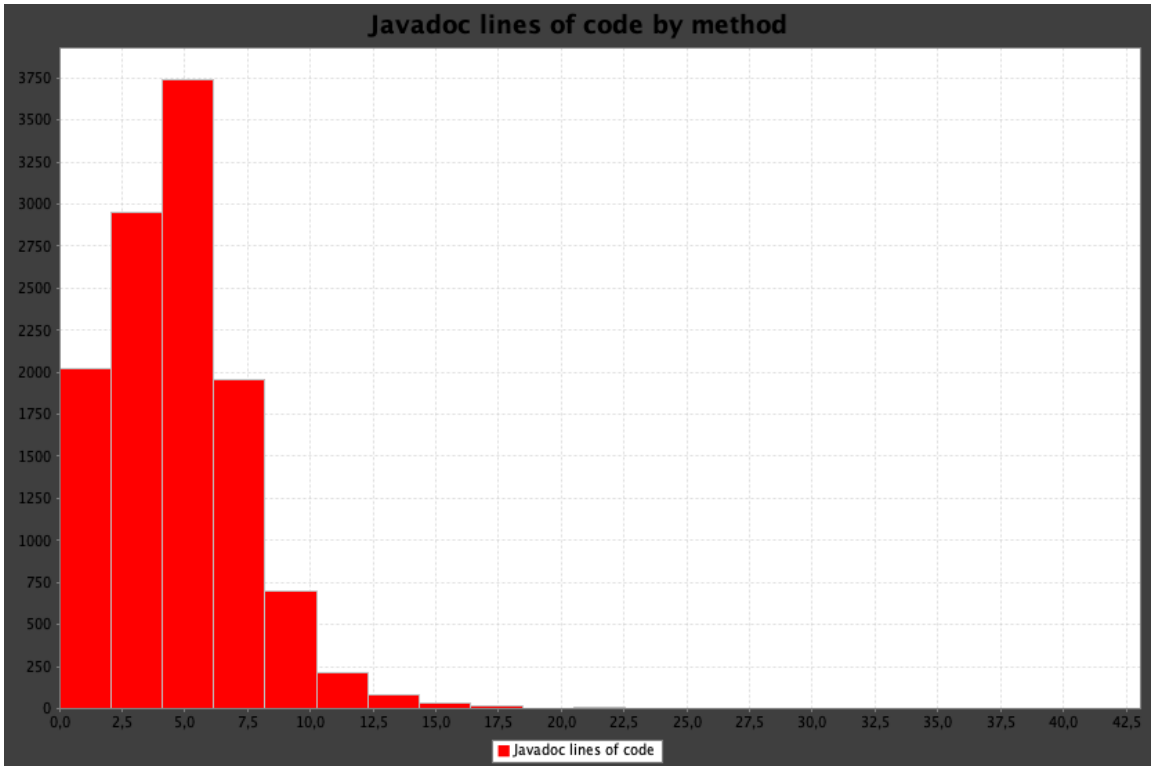
A métrica pode ajudar a entender o quão bem documentado está o código. É essencial encontrar um equilíbrio entre código e comentários para melhorar a manutenção do código. Olhando para o gráfico podemos denotar que a maioria dos métodos tem entre 0 a 15 linhas de comentários. Há uma grande queda de frequência de métodos com mais de 15 linhas comentadas. O número de métodos com 0 linhas de comentários é muito elevado. O que não deveria acontecer, sendo aproximadamente 9200 o número de métodos com 0 linhas comentadas.



-JLOC

Calcula o número de linhas de comentários Javadoc em cada método. Espaços em branco não são contados.

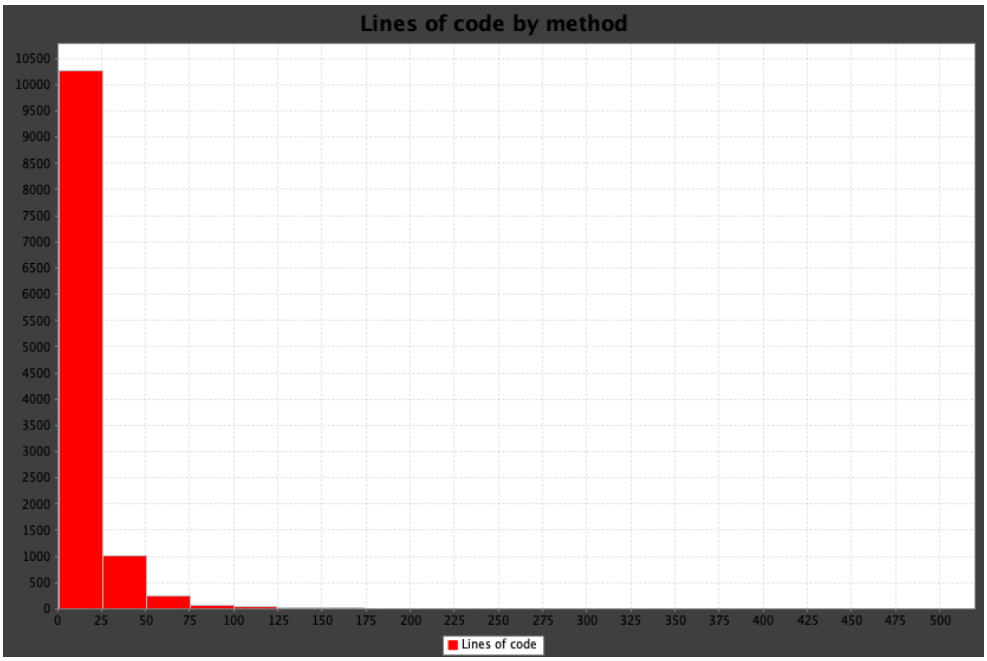
A maioria dos métodos tem um número de linhas de código javadoc que se concentra em torno de 1750. Existem muitos métodos (cerca de 3750) com cerca de 5 linhas de comentários Javadoc. Além disso vemos uma queda enorme de métodos com linhas de comentários Javadoc a partir das 7,5 linhas de comentários Javadoc.



-LOC

Calcula as linhas de código em cada método. As linhas de código em classes anónimas ou locais e nos seus métodos estão incluídas na contagem do método que as contém. Linhas com apenas comentários são contadas, mas linhas com apenas espaços em branco não são.

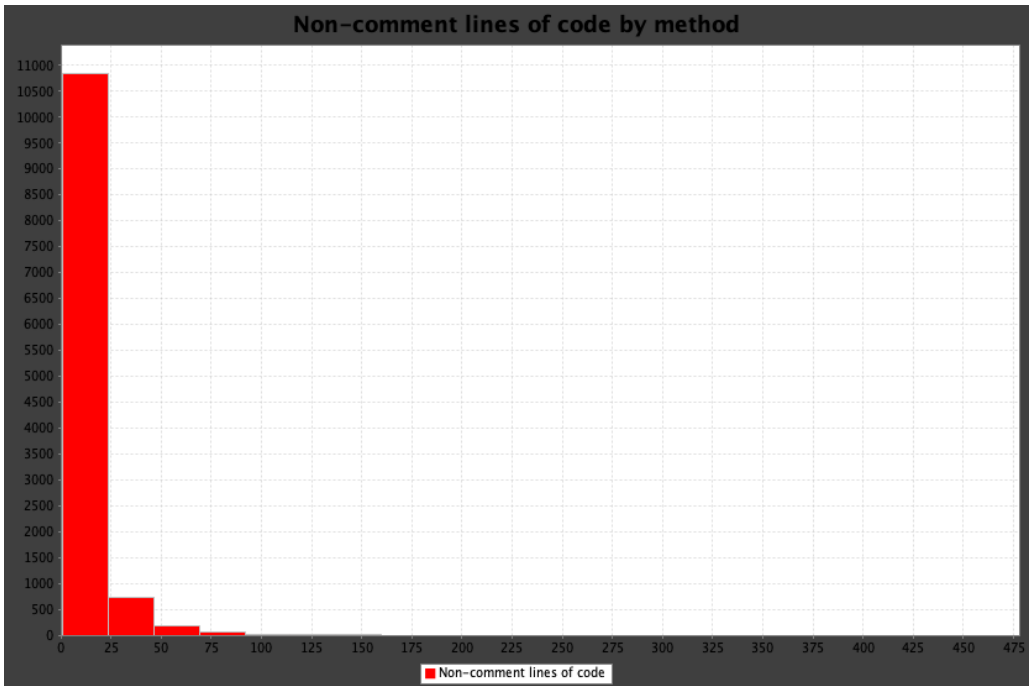
A partir deste gráfico, podemos denotar que cerca de 10000 métodos possui entre 0 a 25 linhas de código, sendo que há um pique enorme de linhas de código em cada método a partir das 25 linhas de código. Isto quer dizer que a grande maioria dos métodos não possui muito mais de 25 linhas de código, o que é bom pois torna o código mais fácil de ler, analisar e de perceber.



- NCLOC

Calcula o número de linhas de código não comentadas em cada método. Linhas de comentário e linhas vazias não são contadas.

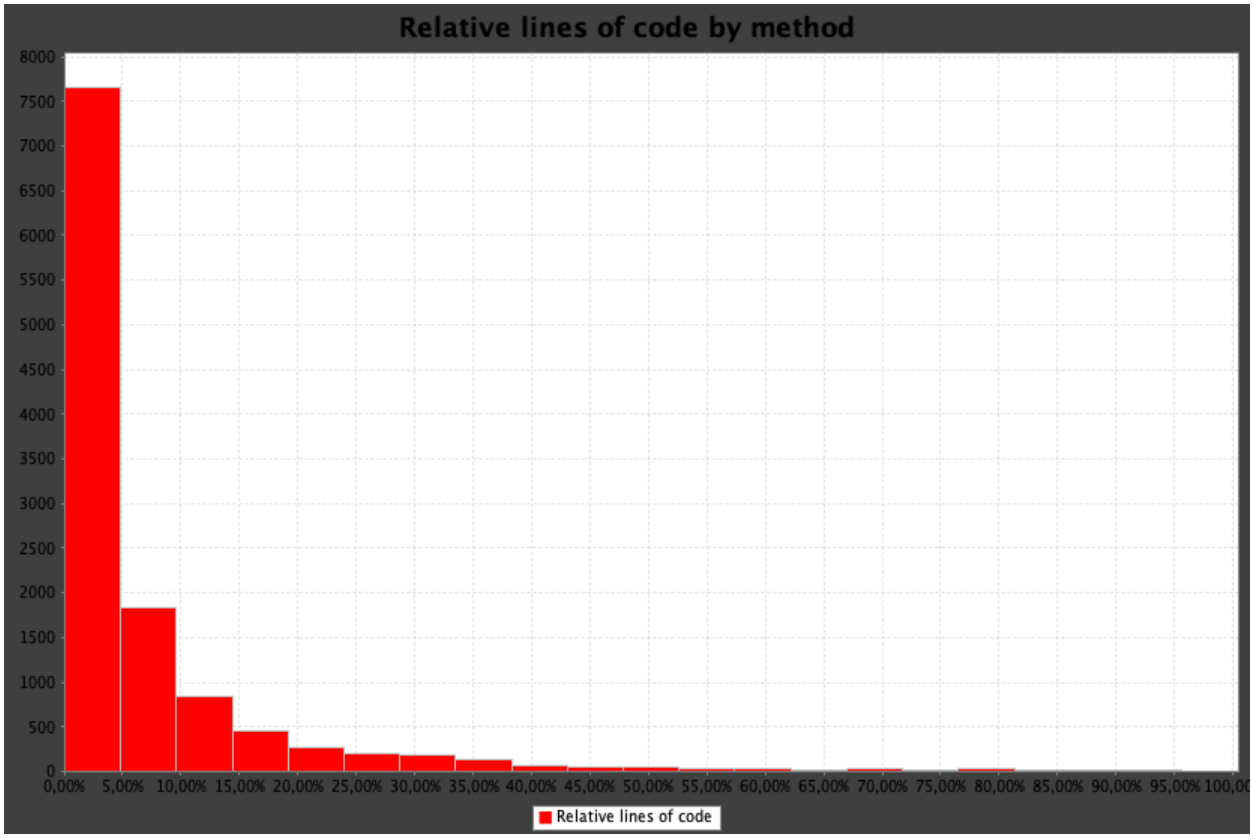
Podemos notar por este gráfico que cerca de 10500 métodos tem entre 0 a 25 linhas de código não comentadas. Podemos ainda observar que a partir das 25 linhas de código não comentadas há uma grande decadência de métodos. Ora isto quer dizer que apesar de haver vários métodos com varias linhas de código comentadas, ainda há cerca de 10500 métodos que têm uma ou 25 linhas de código que não têm nenhum tipo de comentário.



- RLOC

Calcula a proporção de linhas de código de um método em relação às linhas de código de sua classe contida. Métodos com valores relativamente altos de linhas de código podem indicar má abstração.

A partir deste gráfico podemos inferir que numa classe com 7500 linhas de código entre 0 a 5 % são linhas de código pertencentes a um determinado método. Ora isto quer dizer que por cada classe as linhas de código por método não estão bem distribuídas e que portanto há muitas classes onde determinados métodos possuem grandes quantidades de código. Isto não é uma boa prática pois cada método deveria ser o mais curto possível a fim de evitar uma grande complexidade a nível geral de código.



Guilherme Carvalhão - 62675

MOOD Metrics

As métricas MOOD são frequentemente utilizadas para avaliar a qualidade do código e verificar se a manutenção do código é simples ou não.

- AHF

Calcula o grau de encapsulamento de atributos num projeto. Essencialmente, fornece a proporção de em quantas classes, em média, um atributo é visível, além da classe que o define.
Podemos observar que o grau de encapsulamento de atributos neste projeto é de cerca de 73.09%. Isto significa que temos presente uma boa pratica de design pois 73% dos atributos estão encapsulados.

- AIF

Calcula o grau de herança de atributos num projeto. Essencialmente, fornece a proporção que representa que percentagem dos campos disponíveis numa classe média são devido à herança, em vez de serem definidos directamente na classe.
Isto quer dizer que 81,14% dos atributos do projecto são herdados em uma hierarquia de classes. Isto pode ser um bom sinal, tendo em conta que estamos perante um jogo que possui diversas personagens e muitas dessas personagens têm os mesmos atributos.

- CF

Calcula o grau de acoplamento num projeto como um todo. Essencialmente, relata que proporção das classes de um projeto é usada por uma classe média no projeto.
Revendo o projeto, conseguimos observar que estamos perante um bom caso, ou seja, temos presente que apenas 3,06% do código tem uma dependência para muito outro código. Isto é um bom sinal, pois quer dizer que o código é alterável sem que sejam necessárias muitas alterações no restante código.

- MHF

Calcula o grau de encapsulamento de métodos num projeto. Essencialmente, fornece a proporção de quantas classes, em média, um método é visível, além da classe que o define.
Podemos concluir que 25.33% das classes estão encapsuladas. Isto não é uma boa prática, tendo em conta que uma percentagem maior de encapsulamento representa uma melhor prática de design.

- MIF

Calcula o grau de herança de métodos num projeto. Essencialmente, ele fornece a proporção de que percentagem dos métodos disponíveis numa classe média se deve à herança, em vez de serem definidos directamente na classe. Métodos herdados de classes de biblioteca não são contabilizados.
Calculamos que 72.83% dos métodos das classes são herdadas. Assim sendo, concluímos que isto é uma boa prática, tendo em conta que além de reutilizar código, temos presente um jogo que possui muitas personagens que têm características parecidas e que executam funções parecidas dentro do jogo.

- PF

Calcula o grau de polimorfismo num projeto como um todo. Essencialmente, relata a probabilidade de que um método dado seja sobrescrito numa subclasse.
O polimorfimos é um conceito importante à programação orientada por objectos. Um PF de 9.01% indica que é baixo o uso de polimorfismo no código, o que não é uma boa prática para o tipo de projeto presente, porque não fica em conformidade com a ideia de reutilização de código, o que em si não é uma boa pratica de design.

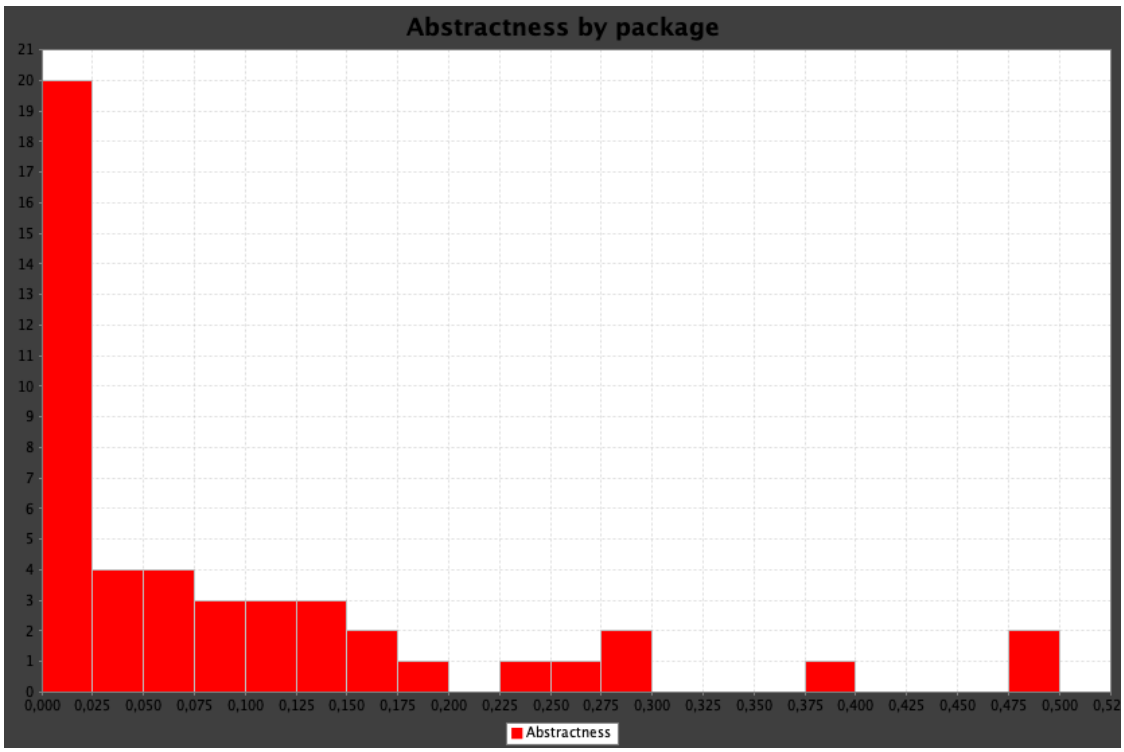
project	AHF	AIF	CF	MHF	MIF	PF
project	73,09%	81,14%	3,06%	25,33%	72,83%	9,01%

Wagna Banar - 64157

MartinPachaging Metrics

- A

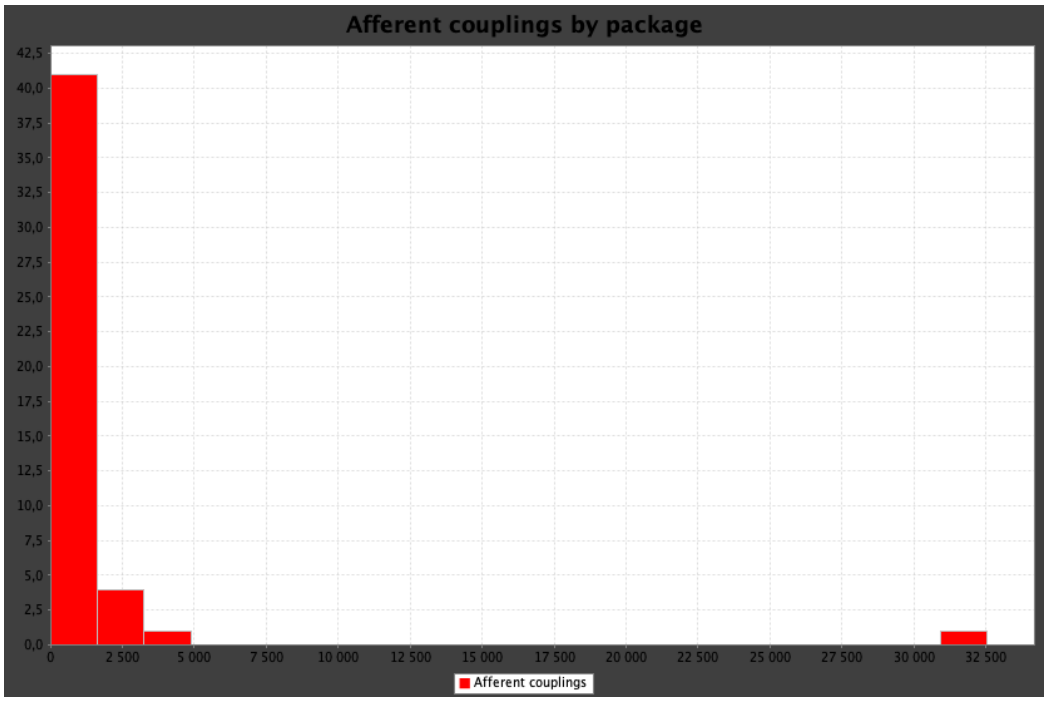
Calcula a Abstracção para cada pacote. A Abstracção é definida como o número de classes abstractas e interfaces dividido pelo número total de classes do pacote. Essa métrica possui uma faixa de valores de [0,1]. A=0 indica um pacote completamente concreto, A=1 indica um pacote completamente abstracto
A (Abstracção):
Olhando para o gráfico podemos concluir que 20 pacotes têm uma abstracção entre 0 e 0.025Ora isto quer dizer que 20 pacotes deste mesmo projecto, não possuem uma abstracção abstracta. Se se der o caso na necessidade de termos mais classes abstractas como é o caso deste trabalho, então seria uma má prática. No entanto isto é normal, visto um jogo ser composto por várias classes que não são necessariamente sobre o conteúdo do jogo, mas sim pelo que o envolve.



- Ca

Calcula o número de Acoplamentos Aferentes para cada pacote. Um Acoplamento Aferente é uma referência de uma classe ou interface externa ao pacote para uma classe ou interface interna ao pacote. Ou seja, Acoplamentos Aferentes são referências que chegam, enquanto Acoplamentos Eferentes são referências que saem ou escapam. Referências de classes de teste e classes de biblioteca não estão incluídas.

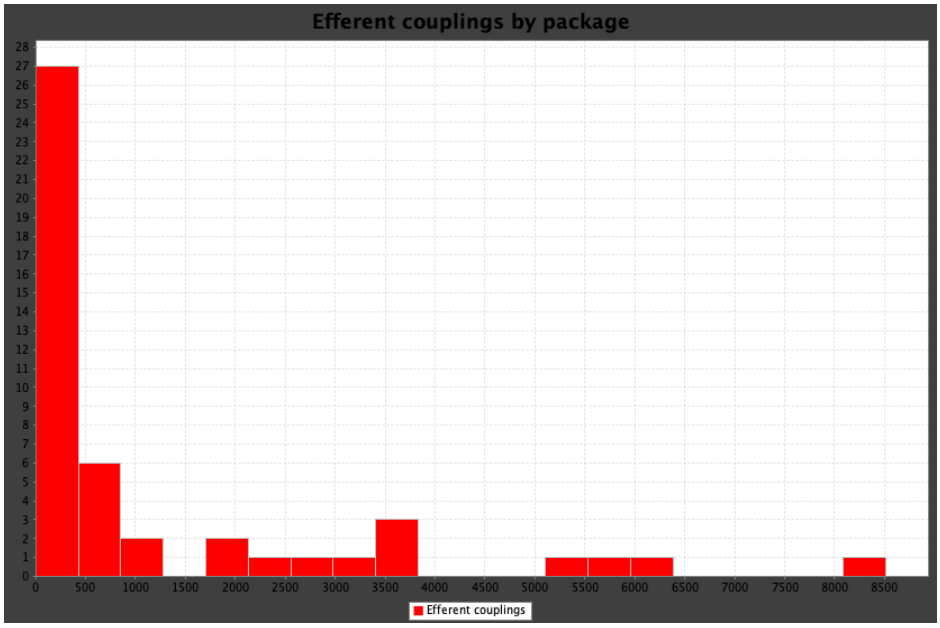
Ora olhando para este gráfico, podemos notar que nos temos entre 0 a 2 pacotes que têm um acoplamento aferente de cerca de 41. Isto quer dizer cerca de 0 a 2 pacotes têm a necessidade de comunicar com 41 métodos ou classes de muitos outros pacotes. Isto não é uma boa prática, pois os pacotes não devem depender assim tanto de outras classes ou métodos. No entanto podemos reparar que isto nem é uma pratica assim tao grande neste projecto, tendo em conta que muitos dos pacotes presentes do projecto não têm muito este comportamento.



- Ce

Calcula o número de Acoplamentos Eferentes para cada pacote. Um Acoplamento Eferente é uma referência de uma classe ou interface interna ao pacote para uma classe ou interface externa ao pacote. Ou seja, Acoplamentos Aferentes são referências que chegam, enquanto Acoplamentos Eferentes são referências que saem ou escapam. Referências a classes de teste e classes de biblioteca não estão incluídas.

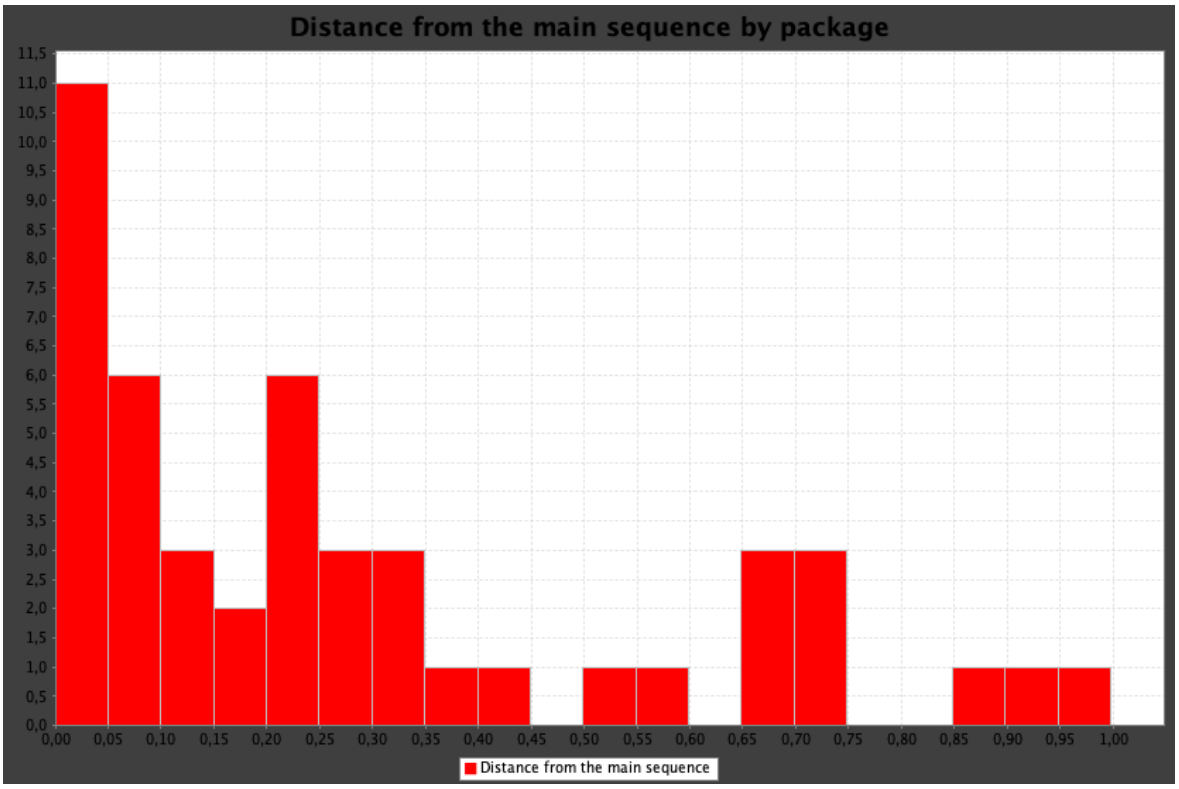
Ora olhando para o gráfico, podemos observar que o pacote com o maior número de acoplamentos eferentes tem um valor de 28.Isto indica que ele possui dependências em 28 classes fora do seu componente. Conforme avançamos para a direita, o número de acoplamentos eferentes diminui, com o valor mais baixo sendo 1. Isso sugere que os pacotes à direita do gráfico são mais autónomos e têm menos dependências em classes externas.



- D

Calcula a Distância da Sequência Principal para cada pacote. A métrica Distância da Sequência Principal é definida como o valor absoluto de 1 - Abstração - Instabilidade (|1 - A - I|). Essa métrica possui uma faixa de valores de [0,1], onde quanto mais próxima D estiver de 0, melhor.

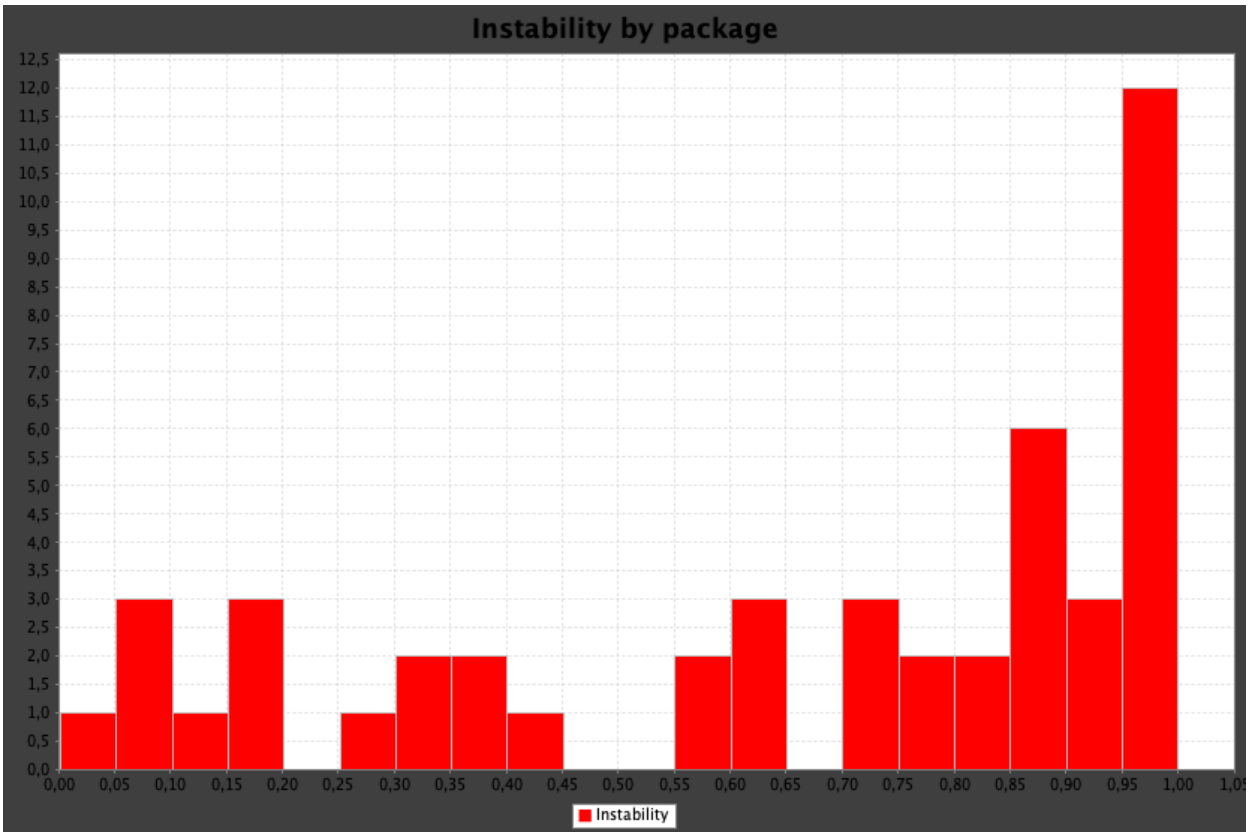
Com base no gráfico, podemos observar que a grande maioria dos pacotes tem uma distância entre 0 e 0.05 ora isto é um bom sinal, pois quando menor a distancia, melhor pratica temos. No entanto podemos observar que temos vários pacotes que estão próximos do 1. Isto não é uma boa prática, pois indica que temos os pacotes não tem um equilíbrio entre abstracção e estabilidade, o que representa um design menos ideal. Desta forma podemos concluir que alguns pacotes podem precisa de serem refatorados a fim de melhor o seu design.



- I

calcula a Instabilidade de um pacote. A Instabilidade de um pacote é definida como os Acoplamentos Eferentes do pacote divididos pela soma dos Acoplamentos Aferentes e Eferentes do pacote (Ce÷(Ca+Ce)). Essa métrica possui uma faixa de valores de [0,1]. I=0 indica um pacote maximamente estável, I=1 indica um pacote maximamente instável.

Olhando para o gráfico, podemos concluir que há uma variação significativa na instabilidade entre os diferentes pacotes deste projeto. Alguns pacotes têm alta instabilidade, o que indica que são mais propensos a ter efeitos inesperados quando modificados. Isto pode ser um risco se esses pacotes forem frequentemente alterados. Já pacotes com baixa instabilidade são menos propensos a ter efeitos inesperados quando modificados, o que pode torná-los mais seguros para trabalhar. Desta forma podemos denotar que grande parte dos pacotes neste projecto, têm uma grande dependência noutros pacotes e desta forma estão sujeitos a várias alterações quando um dos pacotes for também alterado.


Instability by package
12,5
12,0
11,5
11,0
10,5
10,0
9,5
9,0
8,5
8,0
7,5
7,0
6,5
6,0
5,5
5,0
4,5
4,0
3,5
3,0
2,5
2,0
1,5
1,0
0,5
0,0
0,00 0,05 0,10 0,15 0,20 0,25 0,30 0,35 0,40 0,45 0,50 0,55 0,60 0,65 0,70 0,75 0,80 0,85 0,90 0,95 1,00
Instability

Use Case Diagram

José Fernandes - 52970

Name: Criar Nova Colónia

Id: UC-1

DescripAon: O jogador cria uma nova colônia em uma localização específica, isso envolve: a escolha do local, a alocação de colonos e a preparação inicial da colônia.

Actors:
Main: Jogador Secondary: none

Pre-CondiAons: none.

Main flow:
1 - O jogador escolhe uma localização no mapa para a nova colónia e confirma a seleção.
1.1 - O sistema aloca um grupo inicial de colonos e fornece recursos básicos para estabelecer a colóna.

2 - A nova colónia é criada na localização escolhida, e a visão do jogador é transferida

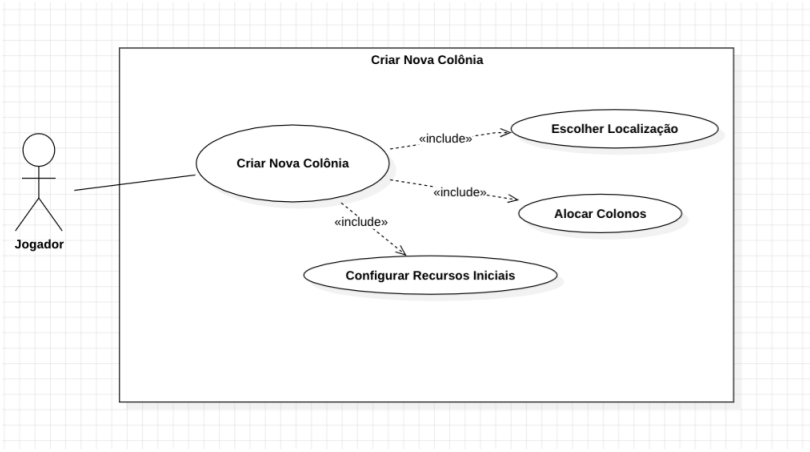
para a nova colónia.

Post flow: none

AlternaAve flow:
1- Se o jogador não Aver colonos suficientes para estabelecer uma nova colónia, é

exibido uma mensagem informaAva "mais colonos são necessários".
2- Se a área escolhida para a colónia esAver ocupada ou for inadequada para

colonização, o sistema exibe a mensagem de informaAva.
3- O jogador pode mudar de ideia e cancelar a criação da nova colónia, daí retornar à visão do mundo para escolher outra localização.



Brás Ramos - 57800

Name: Alcançar Independência

Id: UC-2

Description: O jogador precisa alcançar um número suficiente de pontos para declarar a independência da colônia, e para tal trabalha para acumular pontos de independência através de desenvolvimento econômico, político, cultural e militar.

Actors:

Main: Jogador Secondary: none

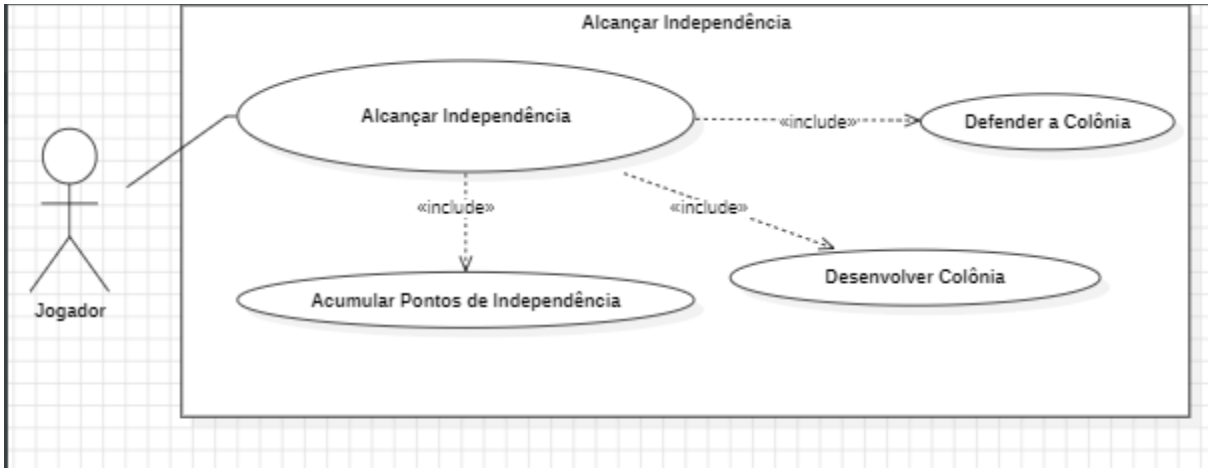
Pre-Conditions: none.

Main flow:
1 - O jogador monitora os pontos acumulando de independência. - O jogador desenvolve a colônia.

2 - O jogador defende a colônia contra ameaças.

Post flow: none

Alternative flow:
- Se a colônia for atacada e não for eficientemente defendida, os pontos de independência podem diminuir.
- Se a colônia não for desenvolvida ou for mal administrada, os pontos podem não aumentam significativamente para alcançar a independência.



Eduardo Dias - 62466

Name: Explorar Mapa Inicio

Id: UC-3

Description: O jogador explora um pouco do mapa no inicio do jogo movendo as personagens do jogo(exploradores) e o barco.

Actors:
Main: Jogador
Secondary: none

Pre-Conditions: o jogador deve ter algum dinheiro na sua posse.

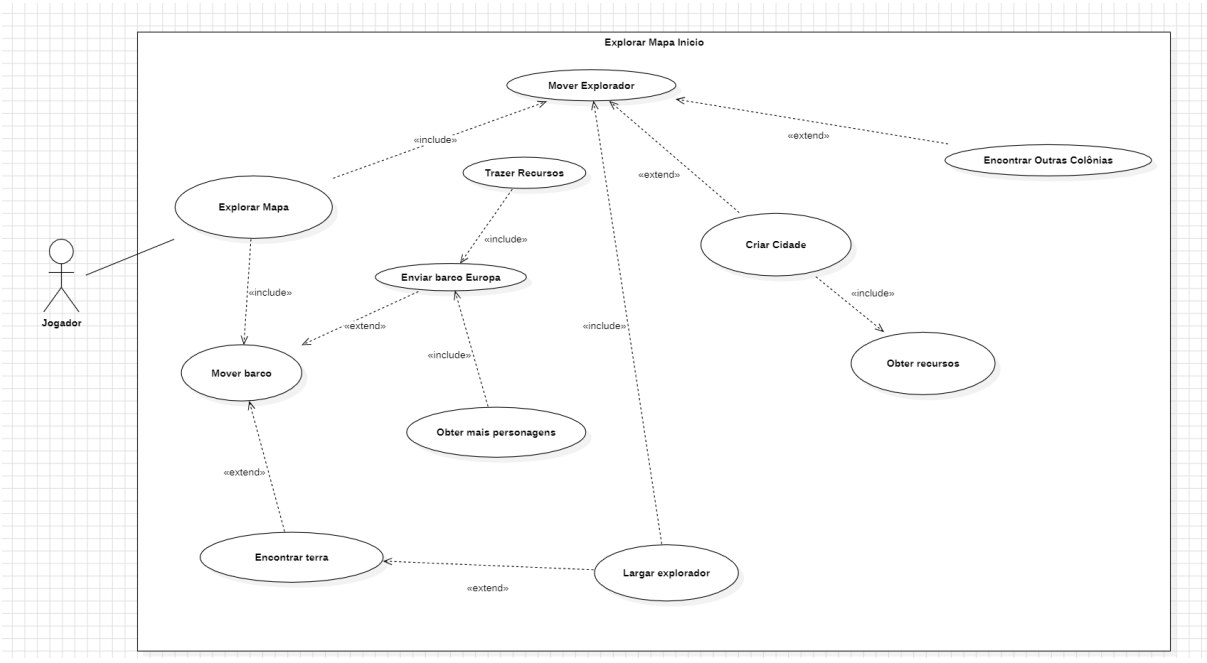
Main flow:
1 - o jogador deve seleccionar o barco e percorrer as casas que for permitido.
1.2 - o barco deve ir para a Europa e regressar com mais personagens e com mais recursos.
1.3 - o barco deve procurar terra e largar os seus colonos.

2 - o jogador deve seleccionar o explorador e percorrer as casas que for permitido.
2.1 - os exploradores devem procurar novo terreno e instalar uma cidade

3 - o jogador deve procurar expandir o seu mapa com a utilização do barco e dos seus colonos.

Post conditions: o mapa está mais visível

Alternative flow: None.



Guilherme Soares Carvalhão - 62675

Name: Estabelecer Relações Diplomáticas

Id: UC-4

Description: O jogador interage com outras colônias para fazer negociações comerciais, tratados de paz ou fazer alianças.

Actors:
Main: Jogador
Secondary: none

Pre-Conditions: nenhuma.

Main flow:

1 - O jogador inicia uma negociação com outra colônia para estabelecer um tratado específico, como um tratado comercial.

2 - O jogador pode propor a formação de uma aliança com outra colônia.

3 - Em caso de conflitos diplomáticos, o jogador pode iniciar um processo de negociação.

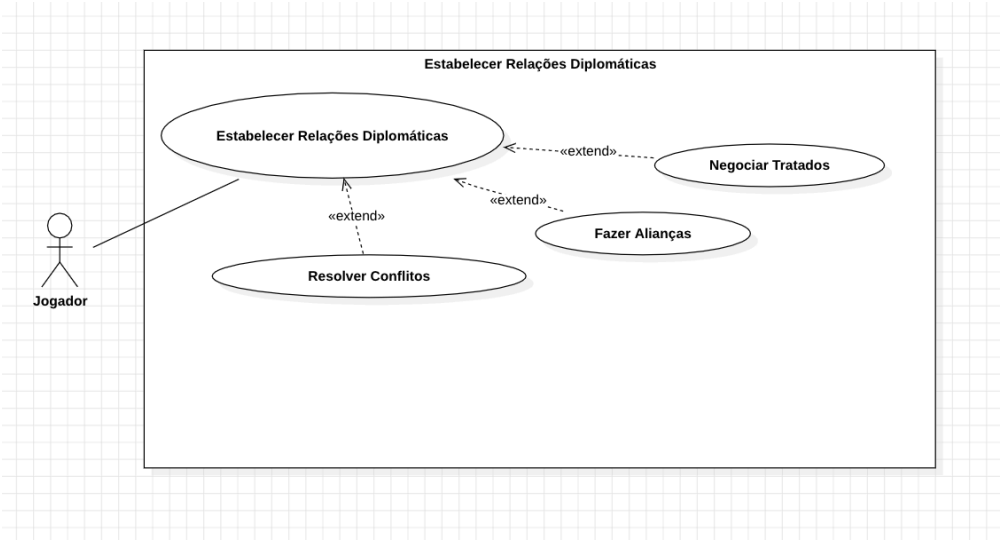
Post flow: none

Alternative flow:

1 - Se as negociações falharem, o tratado proposto é cancelado e as relações diplomáticas podem ficar tensas entre as colônias envolvidas.

2 - Se uma aliança for violada por uma das partes, as colônias aliadas podem decidir suspender ou encerrar a aliança.

3 - Se um conflito não puder ser resolvido pacificamente, as colônias envolvidas podem entrar em guerra.



Wagna Banar – 64157

Name: Gerenciar Recursos

I**d:** UC-5

Description: O jogador gerencia os recursos (comida, madeira, minério e ouro) da colônia incluindo, isso envolve: a produção, coleta e distribuição eficiente desses recursos para manter a colônia em funcionamento.

Actors:
Main: Jogador
Secondary: none

Pre-Conditions: none.

Main flow:

1 - O jogador avalia as áreas ao redor da colônia em busca de recursos naturais. Envia colonos ou trabalhadores para coletar os recursos encontrados.

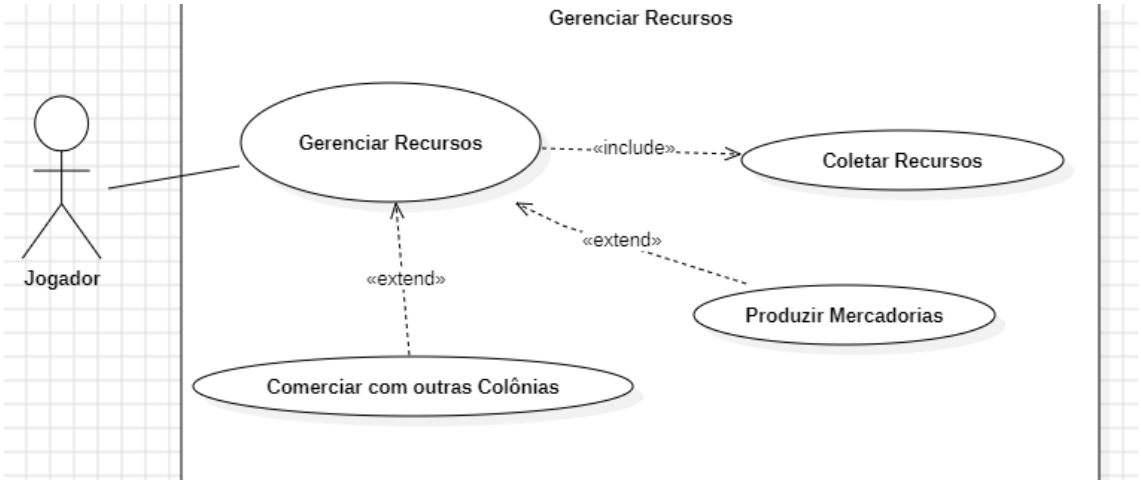
2 - O jogador decide quais mercadorias produzir com base nos recursos disponíveis e nas necessidades da colônia.

3 - O jogador pode optar por utilizar as mercadorias internamente ou negociá-las com outras colônias.

Post flow: none

Alternative flow:
1 - Se não houver recursos suficientes para produzir as mercadorias desejadas, o jogador deve decidir se coleta mais recursos ou prioriza a produção de outras mercadorias.

2- Se as negociações com outras colônias falharem, o jogador pode tentar negociar com diferentes colônias ou considerar outras estratégias para obter os recursos necessários.



Fase 2 Reviews

José Fernandes 52970

Design Patterns ; Code Smells ; Metrics ; Use Case :

Wagna Banar 64157

Brás Ramos 57800

Design Patterns ; Code Smells ; Metrics ; Use Case :

Guilherme Soares Carvalhão 62675

Eduardo Dias 62466

Design Patterns ; Code Smells ; Metrics ; Use Case :

José Fernandes 52970

Guilherme Soares Carvalhão 62675

Design Patterns ; Code Smells ; Metrics ; Use Case :

Eduardo Dias 62466

Wagna Banar 64157

Design Patterns ; Code Smells ; Metrics ; Use Case :

Brás Ramos 57800

Postmortem

Decidimos alterar a User Storie 1, devido a sua complexidade e também por aconselho do próprio professor que desde logo nos disse que seria algo muito difícil de implementar.

Decidimos também alterar a User Storie 3, isto porque apesar de termos conseguido implementar, derivava alguns “bugs”. Estes “bugs” eram devido à própria *engine* do jogo que apenas nos permitia retroceder a jogada logo após termos realizado mais uma jogada, o que anulava a nossa acção de retroceder.