Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo – Campus Serra

Bacharel em Sistemas de Informação

Igor Soares dos Santos

Italo Lourenço Trindade

TRABALHO 1 POO2

Serra / 2015

Igor Soares dos Santos

Italo Lourenço Trindade

TRABALHO 1 POO2

Trabalho apresentado ao Instituto Federal do Espírito Santo – Campus Serra como parte a das exigências da disciplina Programação Orientada a Objetos 2 do Curso de Bacharelado em Sistemas de Informação sob a orientação do professor Paulo como avaliação parcial para aprovação.

Serra / 2015

Sumário

Diagrama de padrões de projeto, explicação dos projetos e discussão sobre a refatoração do código pelos padrões e seu impacto

A intenção de adotar os padrões Abstract Factory e Factory Method no projeto desenvolvido, veio através da necessidade de fornecer uma interface para criação de famílias de objetos e também de definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. O padrão factory method permite adiar a instanciação para as subclasses, atendendo a uma das nossas necessidades. Com a análise do código desenvolvido na primeira entrega do trabalho e com os estudos baseados em design patterns, a nossa motivação para adotar o padrão Abstract Factory e o Factory Method veio através da criação dos objetos guerreiros.

A classe guerreiro é uma classe pai de ofensor e defensor, essas possuem como filhas os tipos de guerreiros com a sua especialização e comportamento de atacar e defender conforme a abaixo:

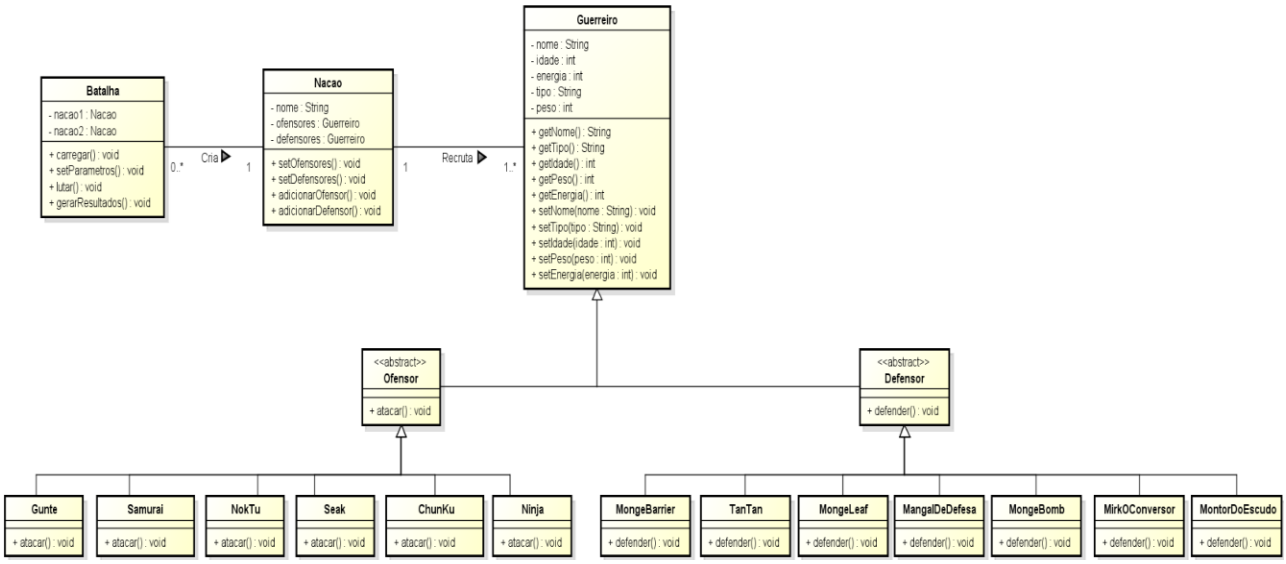


Figura 01 – Diagrama de Classe da lógica de negócio

Cada guerreiro possui um comportamento, estilo e uma interação distinta que definem diferentes apresentações para as classes clientes, então instanciando as classes específicas com seus comportamentos dos métodos diferentes na aplicação como um todo ia tornar difícil a manutenção e atualização futuras do código.

Para tentarmos resolver esse problema definimos uma classe abstrata Superfabrica que declara uma “interface” para criação de cada guerreiro, e as suas subclasses concretas implementam os criarGuerreiro, então as classes clientes que chamam as operações para instanciação de guerreiros não possuem o conhecimento das classes concretas que estão usando.

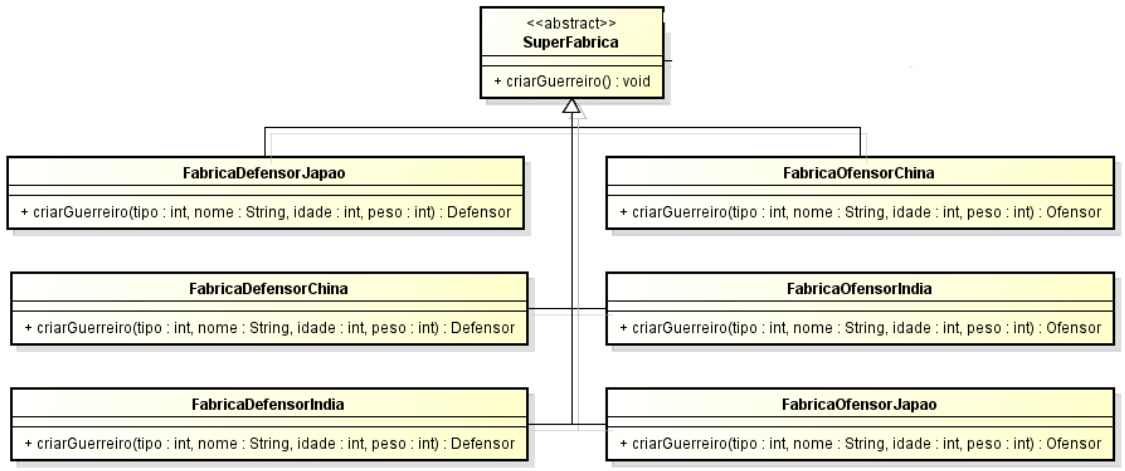


Figura 02 – Diagrama de Classe da utilização do padrão Factory Method

O padrão Factory Method oferece uma solução, ele encapsula o conhecimento da subclasse que deve ser criada, essa subclasse implementa o método criarGuerreiro() da classe Superfabrica e retorna a subclasse apropriada no cliente, então a Factory Method foi aplicado. A existência de subclasses concretas de cada tipo de guerreiro para cada nação, onde cada subclasse implementa a operação de criar o guerreiro apropriado. Então os clientes criam apenas guerreiros e não possuem o conhecimento das classes que implementam o comportamento de cada guerreiro, em outras palavras as classes cliente têm somente que se comprometer com uma interface definida por a classe abstrata Superfabrica e não de uma determinada classe concreta. Dessa forma, os clientes ficam independentes do padrão de interação usado na classe AbstractNacaoBuilder que será explicado em breve.

Entendemos que a aplicabilidade do padrão Factory Method acontece quando uma classe não pode antecipar a criação de um objeto e ela precisa que suas subclasses fiquem encarregadas de realizar a criação do objeto, assim a classe delega responsabilidade para uma dentre várias subclasses auxiliares. E

também a aplicabilidade do uso do padrão Abstract Factory quando um sistema deve ser independe de como seus produtos são criados, compostos ou representados. Também quando o sistema deve ser configurado como um produto de uma família de múltiplos produtos e uma família de objetos-produto for projetada para ser usada em conjunto e garantia dessa restrição e fornecer uma biblioteca de classes de produtos e revela somente suas interfaces, não suas implementações, dentre esses motivos e outros adotamos o padrão Abstract Factory.

Adotamos também o padrão Builder com a intenção de separar a construção de um objeto complexo da sua representação de modo que o mesmo processo de construção possa criar diferentes representações.

Nossa motivação para adotar e tentar aplicar esse padrão foi a seguinte, a classe Nação possui um processo de criação complexo, sendo composta por diversos outros objetos, logo ela deve ser capaz de possuir todos os guerreiros pertencentes aquela nação que são os seus defensores e seus ofensores, então para construímos um AbstractNacaoBuilder que tem como comportamento a criação dos defensores e ofensores através do padrão Abstract Factory e as suas subclasses se especializaram para construir as listas de defensores e ofensores de cada nação.

Na imagem abaixo podemos observar como funciona a hierarquia de classes nesse padrão, a “interface” NacaoBuilder define os métodos para a construção de uma nação, AbstractNacaoBuilder é responsável por implementar os metodos definidos em NacaoBuilder.

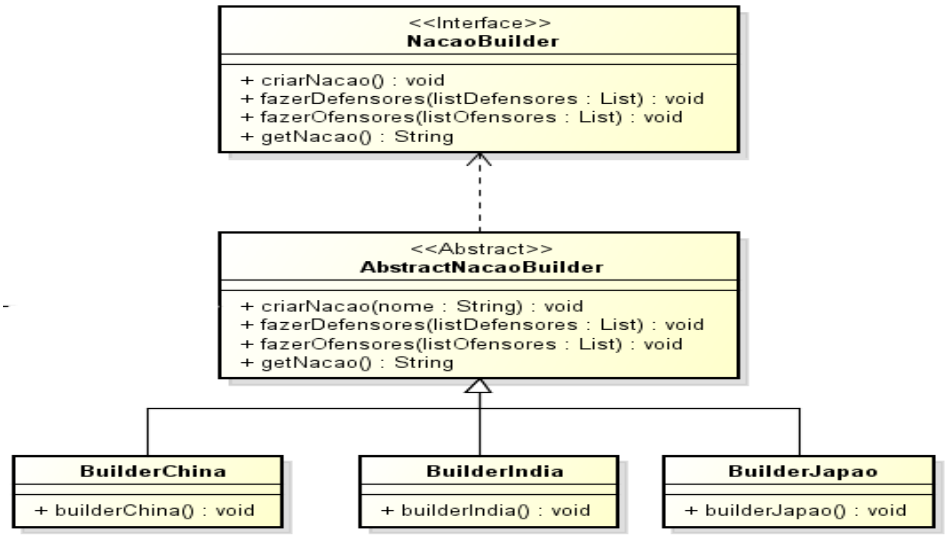
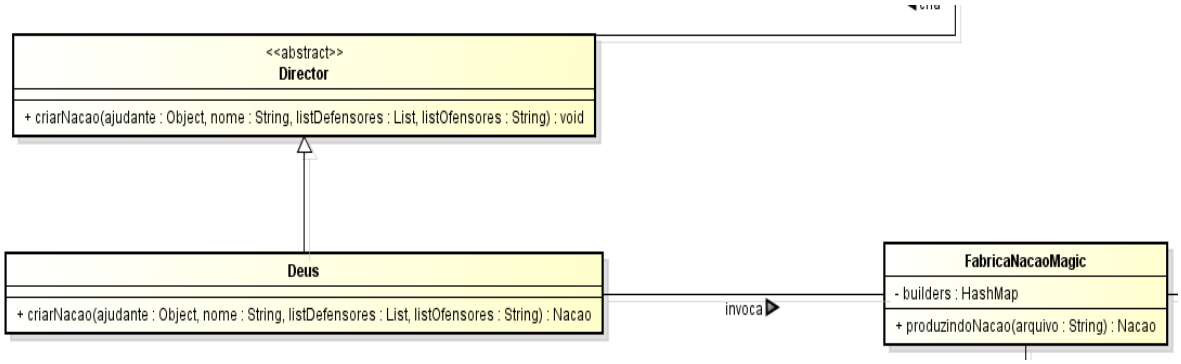


Figura 03 – Diagrama de Classe da utilização do padrão Builder

Cada tipo de classe builder da nação implementa o mecanismo para criação dos guerreiros e montagem daquela nação. O padrão Builder captura todos estes relacionamentos, e essas classes construtoras de nação são chamadas de Builder por esse relacionamento com os guerreiros e nação, e o leitor é chamado de director conforme a figura abaixo:



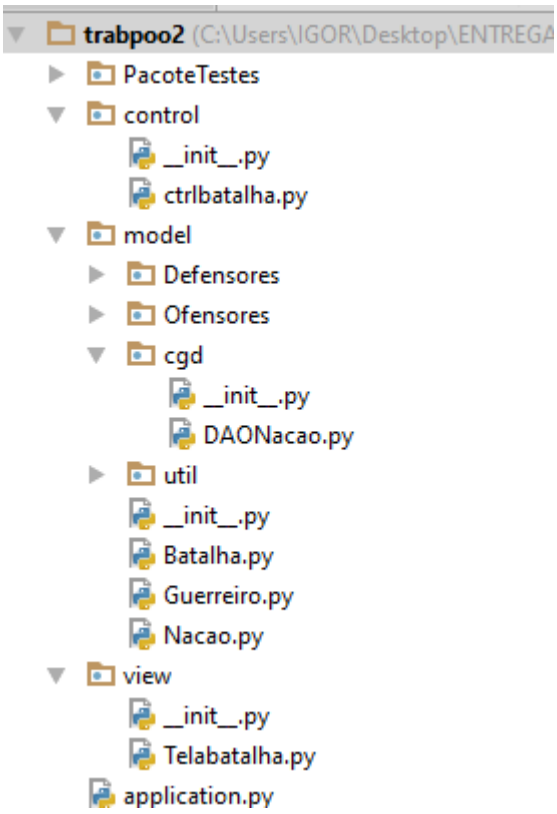
Entendemos que iremos aplicar o padrão Builder quando o algoritmo de criação de um objeto complexo deve ser independente das partes que compõem o objeto e de como elas serão montadas e esse processo de construção deve permitir diferentes representações para o objeto que é construído, por esse motivo ligamos os padrões Builder, Factory Method e Abstract Factory para criação de guerreiros e da nação.

Aplicamos o singleton na classe FabricaNacaoMagic, para garantir essa classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma. Nessa classe que os builders são chamados para a construção dos objetos guerreiros e nações então a motivação seria essa primeiramente que torna essa classe singleton para poder a própria ser responsável por manter o controle da sua única instância.

Entendemos que iremos aplicar o padrão singleton quando for preciso haver apenas uma instância de uma classe, e essa instância tiver que dar acesso aos clientes através de um ponto bem conhecido, a única instância tiver de ser extensível através de subclasses, possibilitando aos clientes usar instâncias estendias sem alterar o seu código.

Já o padrão prototype não aplicamos pois ele especifica os tipos de objetos a serem criados usando uma instância protótipo e criar novos objetos pela cópia desse protótipo e também diz que um sistema tiver que ser independente de como seus produtos são criados, compostos e representados para o seu uso. E na criação de guerreiros cada guerreiro aplica um comportamento distinto onde não seria útil cloná-los.

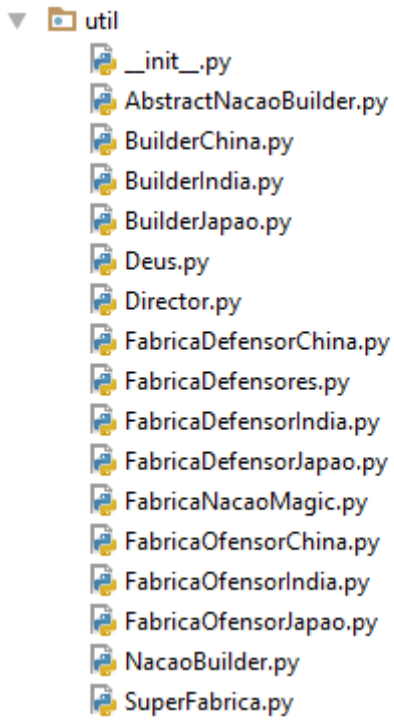
MVC



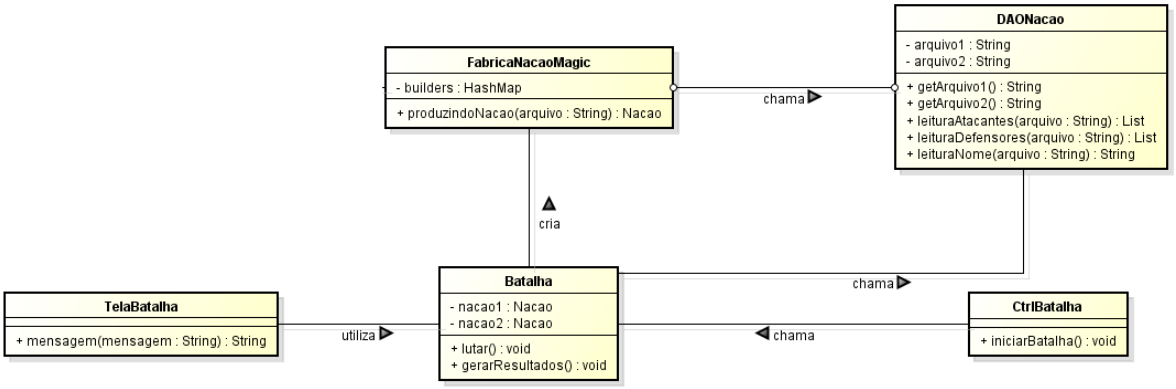
O padrão Model-View-Controller foi uma forma de organizar a aplicação com intuito de dividir em camadas, onde o os objetos escondem as suas informações e como elas são manipuladas e apresentam uma simples interface para o seu uso.

Mantivemos no modelo os pacotes Defensores e Ofensores onde encontra-se as classes de acordo com o diagrama de classe e o seu comportamento de como atacar e defender de cada guerreiro seja ele defensor ou ofensor, porém alteramos a classe Batalha para modelo devido ao entendimento do grupo, no cgd ficou a classe DAONacao com a responsabilidade de saber comunicar com o arquivo e realizar a leitura dos guerreiros e da nome da nação, assim retornando listas de ofensores e defensores e o nome da nação.

Foi adicionado o novo pacote util no modelo contendo as regras e a lógica do négocio para a criação dos objetos.



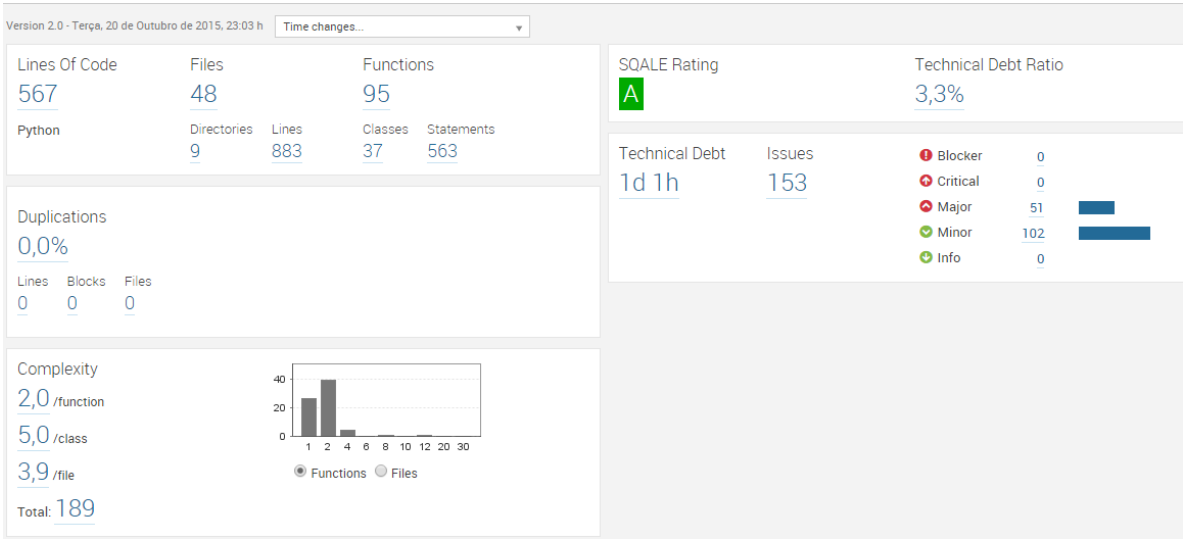
Já na camada do controle na classe CrtlBatalha ficou apenas com a responsabilidade de instanciar um objeto batalha onde a classe batalha possui o comportamento que vai se interagir com as classes DAONacao e TelaBatalha, sendo assim comunicando-se com o modelo e visão, portanto o encapsulamento do código acontece pois o modelo fica dividido em camadas e também consegue a separação dos conceitos e do código.



Análise de dados

Resultados obtidos utilizando o SonarQube:

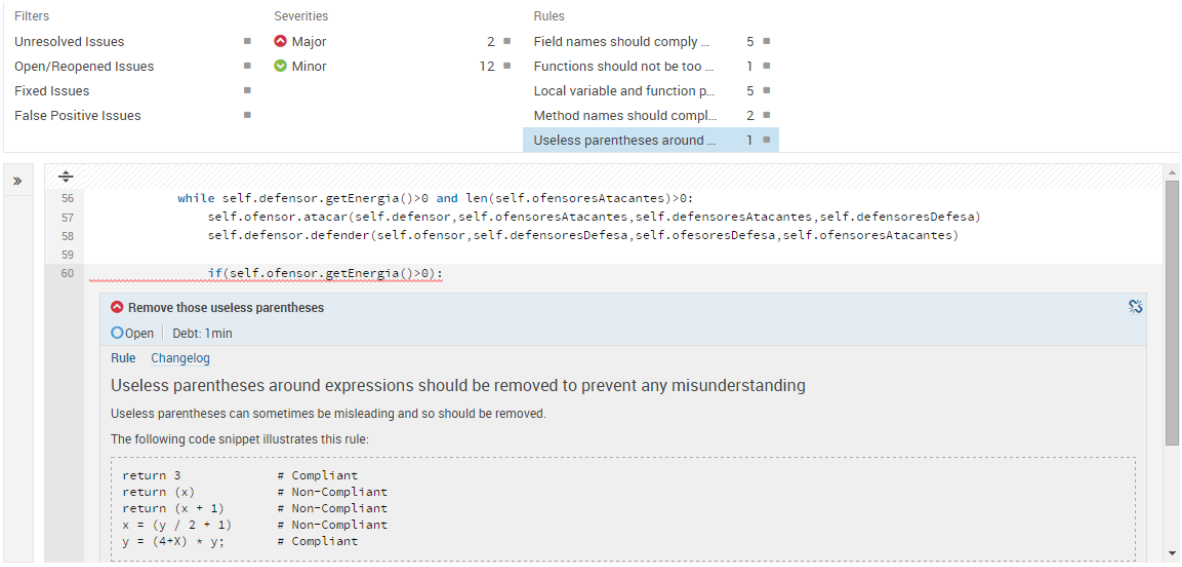
A imagem abaixo fornece várias métricas como número de linhas de código, número de classes, duplicações de código e complexidade ciclomática.



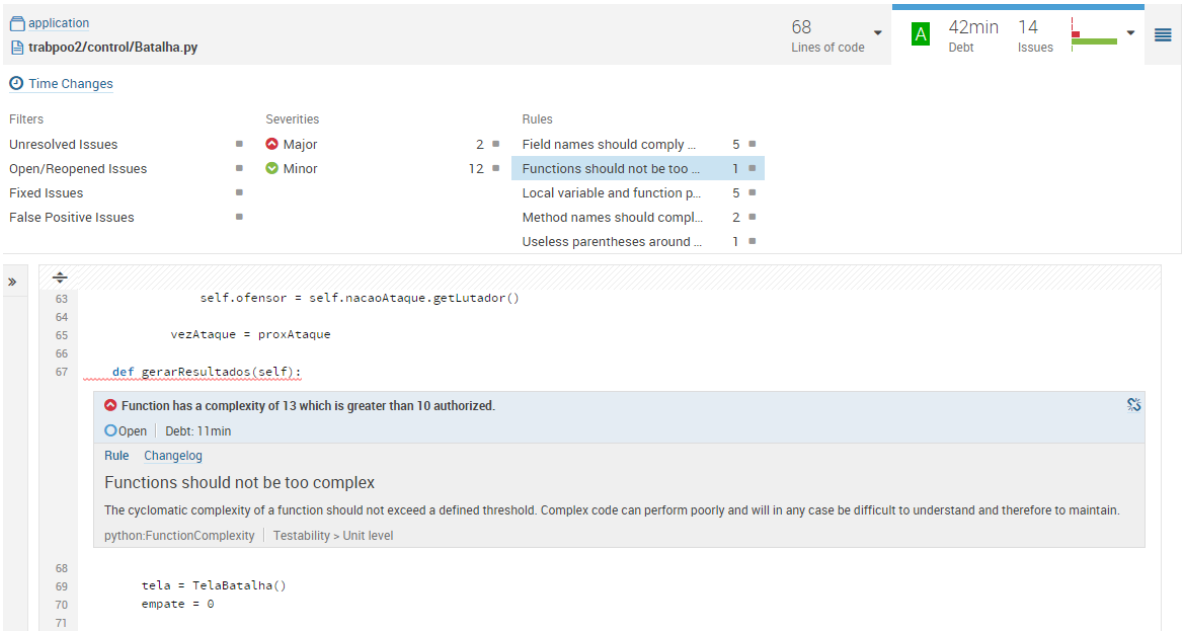
Fazendo uma breve comparação sobre o trabalho 0, em termos de quantidade de linhas de código, funções, arquivos tiveram um aumento assim como no Techinical Debt e Issues que iremos explicar em breve, porém conseguimos diminuir a complexidade do código comparado em números o trabalho 0 tinha 2.1/function, 7.0/class e 5.3/file.

O número de regras que foram quebradas foram 153 Issues, onde teve um aumento do trabalho 0 que tinha 121 Issues. Algumas dessas Issues são simples de serem consertadas e outras mais complexas, assim deixamos essas simples para mostrar que o SonarQube falar que quebrou regras mesmo assim.

As imagens abaixo mostram algumas regras quebradas e também as suas possíveis “soluções” segundo o SonarQube:



Essa imagem acima mostra que segundo o SonarQube foi quebrada uma regra dos parênteses, onde diz que usar parênteses de forma enganosa deve ser removido.



A imagem acima mostra que ele reclama também dessa função, pois segundo o SonarQube essa função passou do limite máximo autorizado de complexidade. A regra é que as funções não podem ser muito complexas assim podendo ser executadas mal e ser difícil de serem compreendidas.