UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

DEPARTAMENTO DE ESTATÍSTICA E INFORMÁTICA

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Discentes: Arthur Queiroz, Caio Farias e Yasmmin Claudino

Análise estática do código-fonte do projeto HeyFood

Recife, 2019

Sumário

1. Visão Geral

Através da Análise Estática do código, foi feito uma busca em erros que poderiam acarretar no mau funcionamento do software, podendo assim diminuir seu ciclo de vida.

2.1 **Resultados da Análise Estática e Evolução do Código**

	Versão 1	**Versão 2**	**Versão 3**
Bug	**9**	**1**	**1**
Vulnerabilidades	**0**	**0**	**0**
Code Smells	**158**	**162**	**189**
Complexidade Ciclomática	**440**	**529**	**616**
Linhas Duplicadas	**13.7%**	**15.3%**	**13.2%**

2.2 **Code Smells**

A maior parte dos Code Smells encontrados têm uma maior influência nas boas práticas para se manter um código. Dessa forma, podendo influenciar diretamente na evolução e na manutenibilidade do software.

Versão 1 - 16/06

Foram encontrados 158 Code Smells na primeira versão de verificação do projeto, variando entre Major Code Smell e Minor Code Smell.

- ➔ Variáveis declaradas em uma mesma linha;
- ➔ Variáveis sem uso;
- ➔ Comentários em várias partes do código;
- ➔ Uso de exceções genéricas;
- ➔ 8 imports que não foram usados.

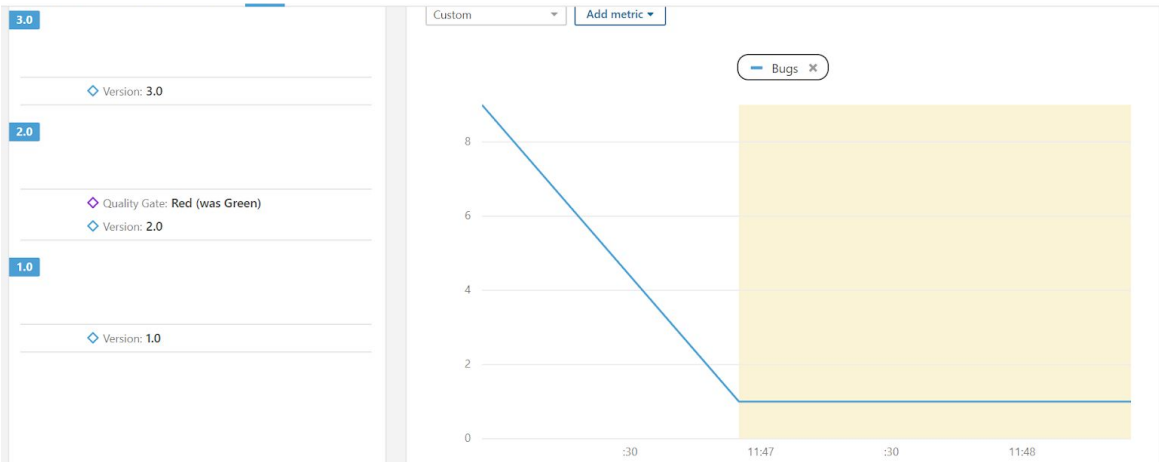
Versão 2 - 23/06

Aumentou o número de code smell, mas diminui o nível de bug consideravelmente a versão 1. E conseguiram alcançar um Quality Gate, que é a melhora de uma política de organização.

Versão 3 - 30/06

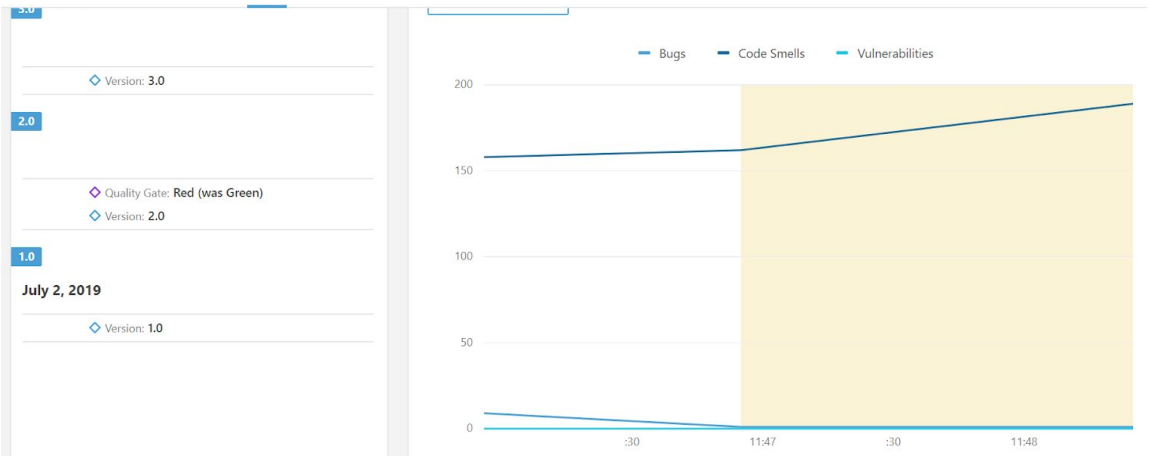
Aumentaram mais ainda o número de code smell e se perdeu a Quality Gate.

Gráfico de Bugs



Evoluíram na parte da diminuição de bugs do sistema

Gráfico de Code Smell, vulnerabilidade e bugs



Aumentaram o número de code smell, enquanto bug diminuiu e vulnerabilidade se manteve constante

2.3 **Complexidade Ciclomática**

Métodos com alto nível de controle têm uma dificuldade para manter o seu fluxo. Dessa forma, torna-se complicado de se manter o código com a evolução do software.

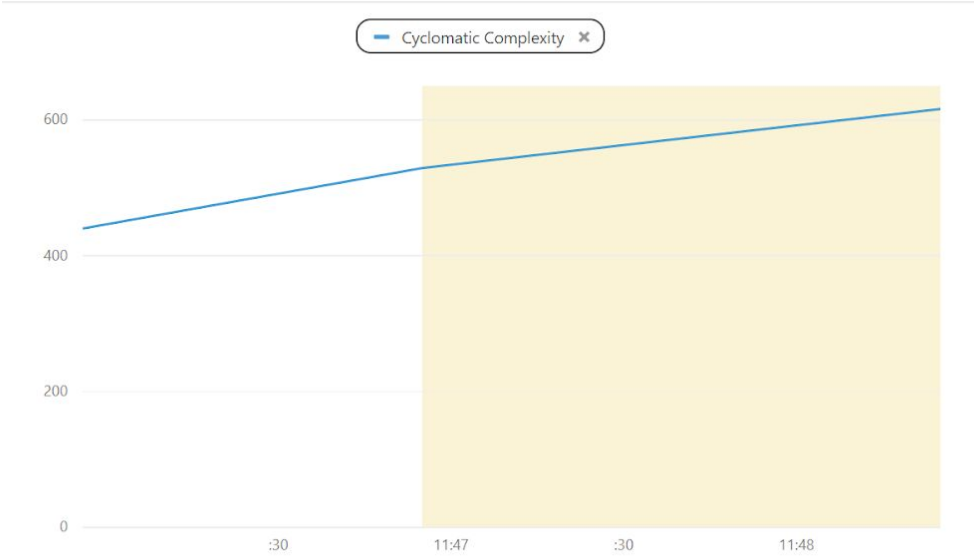
- Versão 1:
    - Método para validar CNPJ tem um alto índice de Complexidade, o qual pode ser difícil de manter.

```
    private boolean validarCnpj() {
        String CNPJ = this.cnpj.getText().toString().replace( target: ".",  replacement: "");
        CNPJ = CNPJ.replace( target: "-", replacement: "");
        CNPJ = CNPJ.replace( target: "/", replacement: "");
// considera-se erro CNPJ's formados por uma sequencia de numeros iguais
        if (CNPJ.equals("00000000000000") || CNPJ.equals("11111111111111") ||
                CNPJ.equals("22222222222222") || CNPJ.equals("33333333333333") ||
                CNPJ.equals("44444444444444") || CNPJ.equals("55555555555555") ||
                CNPJ.equals("66666666666666") || CNPJ.equals("77777777777777") ||
                CNPJ.equals("88888888888888") || CNPJ.equals("99999999999999") ||
                (CNPJ.length() != 14))
            return(false);

        char dig13, dig14;
        int sm, i, r, num, peso;

// "try" - protege o código para eventuais erros de conversao de tipo (int)
        try {
// Calculo do 1o. Digito Verificador
            sm = 0;
            peso = 2;
            for (i=11; i>=0; i--) {
// converte o i-ésimo caractere do CNPJ em um número:
// por exemplo, transforma o caractere '0' no inteiro 0
// (48 eh a posição de '0' na tabela ASCII)
                num = (int)(CNPJ.charAt(i) - 48);
                sm = sm + (num * peso);
                peso = peso + 1;
                if (peso == 10)
                    peso = 2;
            }

            r = sm % 11;
            if ((r == 0) || (r == 1))
                dig13 = '0';
            else dig13 = (char)((11-r) + 48);

// Calculo do 2o. Digito Verificador
            sm = 0;
            peso = 2;
            for (i=12; i>=0; i--) {
                num = (int)(CNPJ.charAt(i)- 48);
                sm = sm + (num * peso);
                peso = peso + 1;
                if (peso == 10)
                    peso = 2;
            }

            r = sm % 11;
            if ((r == 0) || (r == 1))
                dig14 = '0';
            else dig14 = (char)((11-r) + 48);

// Verifica se os dígitos calculados conferem com os dígitos informados.
            if ((dig13 == CNPJ.charAt(12)) && (dig14 == CNPJ.charAt(13)))
                return(true);
            else return(false);
        } catch (InputMismatchException erro) {
            return(false);
        }
    }
```

2.4 **Evolução do Código**

A equipe demonstrou evolução do código da versão 1 para a versão 2, no entanto, teve uma recaída na de evolução na versão 3.

Gráfico Evolução Complexidade Ciclomática



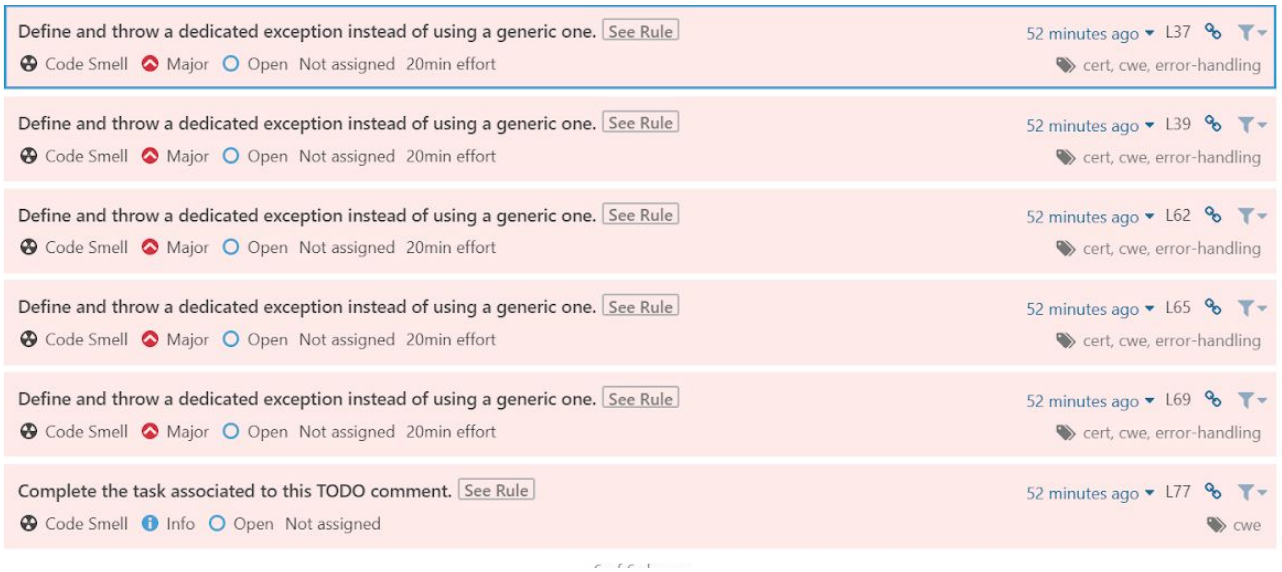
Houve um aumento da complexidade ciclomática da versão 1 a versão 3. Dessa forma colocando em risco a continuidade de certas partes do código.

Tendo em vista que as quantidades de bugs diminuíram da versão 1 para versão 2, e se mantendo a mesma quantidade na versão 3.

Além do mais, o projeto mostrou-se forte na questão da vulnerabilidade que se manteve em 0 da primeira versão até a terceira versão.

Versão 1

Versão 3



Erros que persistiram nas três versões foi manter exceções e métodos genéricos, ao invés de especificar. Dessa forma, podendo saber exatamente onde foi o bug ou problema no código quando acontecesse. Além do mais, o grande número de linhas e blocos de códigos repetidos também se mantiveram constantes.