# Análise do TornGest com SonarQube

FC47806 - Paulo Santos
FC47878 - Rafael Melo
FC47888 - Diogo Pereira

25 de Novembro de 2018

# Conteúdo

# Capítulo 1

# Introdução

Feita a análise arquitetural e de desenho no *Software Architecture Document*, iremos agora submeter o sistema *TornGest* a uma análise ao nível do código com a ferramenta *SonarQube* que nos permitirá ver se os problemas no código do sistema influenciam os requisitos de qualidade que foram descritos no SAD.

Dadas as métricas que o sonarqube nos fornece na sua forma mais básica, identificámos os conceitos de qualidade fiabilidade, segurança e modificabilidade associadas às métricas do *SonarQube*: bugs, vulnerabilidades e *Technical Debt*.

# Capítulo 2

# Impacto das métricas nos conceitos de qualidade

## 2.1 Segurança

Em relação ao aspeto do conceito de qualidade de segurança, este projeto TornGest contém 5 vulnerabilidades obtendo nota B. 4 destas vulnerabilidades residem no *FrontController* e uma no *RegistarResultadoHandler*.

**RegistarResultadoHandler**

A vulnerabilidade do RegistarResultadoHandler trata-se de uma falta de uso do *Logger* para o caso da operação de adicionar o resultado dar uma exceção. Apenas é feito a impressão do stack trace o que implica que mesmo sendo raro obter um erro na adição do resultado, se ocorrer o utilizador vai ficar a par de alguns detalhes de implementação do sistema ou outra informação sensível o que pode revelar uma vulnerabilidade por onde um agente malicioso possa explorar o sistema.

A solução desta vulnerabilidade passa por utilizar um logger para expor a exceção como por exemplo: LOGGER.log("Context", e); informando o utilizador apenas da exceção ocorrida.

**FrontController**

A primeira vulnerabilidade encontra-se no método doGet, no facto de que a chamada do método process do comando não está num *try/catch*. O método doGet lança a ServletException e IOException mas essas exceções podem mandar para o output ao utilizador informações sensíveis de *debug* do servlet que pode levar também a ataques DDOS. Portanto a solução passa por colocar um *try/catch* para tentar expor a exceção de um modo que não comprometa a segurança deste sistema como por exemplo anunciando um erro mais genérico.

A segunda vulnerabilidade encontra-se no método *doPost* que chama o método *doGet* que sofre do mesmo problema da vulnerabilidade anterior, lançando as exceções ServletException e IOException expondo elementos da servlet ao utilizador que podem resultar em informações sensíveis que são apanhadas por agentes maliciosos que penetrar o sistema TornGest e assim violar a qualidade de segurança.

## 2.2 Modificabilidade

Para o conceito de modificabilidade o projecto TornGest contém 53 *code smells* obtendo assim um *rating* de A e um *technical debt* de 1 dia e 5 horas com um debt rate de 0.9%. Isto é bastante bom mas ainda pode ser melhorado. Para isso fomos ver os vários *bad smells* mencionados pela ferramenta *sonarqube*. Reparamos que a maior parte estava contida no *business* da aplicação tendo 33 bad smells, com mais 16 no *web client* e 4 no cliente *gui*.

**Modificabilidade no GUI Client**

No gui os únicos *smells* que existem são três de uso de lambdas nos *models* e um no MenuPrincipalController em que é criado um atributo que já existe na classe que extende, o BaseController. No total isto dá um technical debt de 11 minutos, que é bastante reduzido e simples de resolver.

**Modificabilidade no Web Client**

No cliente web já existe um *technical debt* de 2 horas e 37 minutos que, embora seja maior que o do GUI, continua a ser bastante reduzido e fácil de corrigir. Os maiores culpados deste technical debt são a classe FrontController, que tem um technical debt de 1 hora e 5 minutos, devido à falta de @Override nos métodos relevantes e às strings estarem *hardcoded*. Nas Actions os únicos smells que existem é no nome das packages, por não estarem de acordo com uma expressão regular.

**Modificabilidade no Business**

A grande parte do technical debt vem do business, tendo um technical debt de 1 dia e 2 horas, que comparado com os outros já é um tamanho considerável. Os maiores causadores deste valores elevados são no RegistarParticipanteHandler que tém dois métodos com complexidade acima do máximo, o método registarEmTorneioIndividual com uma complexidade de 31 e o método registarEmTorneioEquipa com uma complexidade de 26. O resto do technical debt está nas classes que representam as entidades persistidas na base de dados, em que o maior culpado deste é a implementação do *clone* em cada uma das classes.

**Problemas a resolver e efeitos das mudanças**

Visto o *technical debt* estar relativamente baixo não consideramos que seja crítico tratar o mais rápido possível dos problemas. No entanto verificámos que existem alguns smells que têm uma classificação de critical e portanto decidimos alterar alguns desses e ver o resultado que obtemos no sonarqube.

## 2.3 Reliability

Relativamente ao conceito de *Reliability*, o projeto do TornGest apresentou no total 5 bugs. Quando temos um sistema onde este requisito não funcional é crítico, a resolução dos bugs apresentados por ferramentas de avaliação destes requisitos como o *Sonarqube* torna-se essencial. Os casos apresentados no *Sonarqube* representam bugs que poderão não são críticos e que não causariam grande impacto durante o funcionamente da aplicação. Os bugs apresentados pelo *Sonarqube* foram no *FrontController* e no *RegistarParticipanteHandler*.

**FrontController**
Esta classe apresenta 3 bugs relativos aos seus atributos. Visto que o *Front Controller* é um objeto que uma vez criado os seus atributos nunca mais irão mudar, faz sentido aplicar a resolução indicada pela *Sonarqube* de colocar como **final** os atributos indicados. Deste modo não é possível que o valor em tempo de execução destes atributos mude, colocando em risco o acesso dos clientes *web* aos serviços prestados pelo sistema.

**RegistarParticipanteHandler**
Nesta classe são apresentados dois bugs relativos à utilização do *Java Functional* para realizar a ordenação de dois arrays. A sua resolução passaria por realizar a chamada aos *streams* visto que estes são *lazy* e só realizam o método neste caso proposto quando é realizada a chamada.