Projeto 1 – Managing security risks in the use of third-party components
Projeto 2 – Ferramentas e técnicas de Coding standards

Universidade do Minho
Escola de Engenharia

Command Line Program para teste das Operações do Serviço de Assinatura com Chave Móvel Digital

Reverse Engineer da aplicação CMD-SOAP com aplicação dos conceitos de Programação Segura

Submetido por:

Diogo Araújo A78485

Diogo Nogueira A78957

Conteúdo

Contextualização

O processo de *Reverse Engineer* encontra-se muito ligado a toda a área da Engenharia de Segurança, dado que é um processo que permite descobrir o funcionamento do *software* de empresas concorrentes e dessa forma criar/desenvolver novos programas a partir de *softwares* com tecnologia possivelmente ultrapassada.

Por isso que este projeto se torna tão importante, tendo em conta que permite desenvolver um *software* "clone" numa outra linguagem, mas totalmente supervisionado desde o início do seu ciclo de vida e com a aplicação de várias técnicas e testes para garantir a criação de um novo *software* o mais seguro possível.

Assim, este processo de *Reverse Engineer* pode e dever fazer-se acompanhar de todos os conceitos de Técnicas de Desenvolvimento Seguro de *Sofware* (Projeto 1) e Ferramentas e Indicadores de Qualidade de Software e/ou Testes de Software (Projeto 2), de modo a criar um novo *software* que:

- Possua critérios específicos definidos desde o início do seu processo de desenvolvimento.
- Faça jus ao *software* inicial, adotando medidas mais vantajosas em termos de utilização para o Utilizador e dessa forma tornando o programa mais intuitivo, mas nunca deixando de prezar pela segurança oferecida no que toca aos dados a processar/validar.
- Possua um processo de gestão de TPC's bem definido e modelável para as várias versões.
- Vá ao encontro das várias Características e Métricas de Qualidade de um *Software*, de modo a criar um programa totalmente íntegro, compatível, modular, tolerante a possíveis falhas e sempre focado na segurança dada ao Utilizador.

Técnicas de Desenvolvimento Seguro de *Software*

Durante o desenvolvimento de um *software*, os *developers* procuram garantir a entrega de um produto final com a mínima quantidade de *bugs* pensando sempre no menor intervalo de tempo possível para a realização da sua tarefa.

Para que o *software* corresponda assim a todas as expectativas, torna-se indispensável insistir em boas práticas de segurança, através da criação de rotinas que visam desenvolver um *software* perfeitamente seguro e que faça jus aos objetivos inicialmente planificados.

Por isso que é tão importante pensar no "miolo" do *software*, no sentido de compreender aquilo que o compõe e se essa composição não viola/compromete a segurança do produto final.

Uso de *Third-party Components*

Conforme abordado no primeiro projeto, "um *Third-Party Component* consiste (...) num componente de terceiros produzido com o objetivo de ser reutilizável para que possa ser distribuído de forma livre ou até mesmo vendido posteriormente por uma entidade que não o fornecedor original do mesmo". Estes componentes revelam-se assim muito convenientes quando se está a desenvolver um novo *software*, dado que são uma forma eficiente de reduzir os custos do produto final ao mesmo tempo que diminuem o tempo do ciclo de vida do desenvolvimento do *software* em si.

Rapidamente se entende que esta é uma prática muito comum na maioria dos *softwares*, trazendo consigo grandes vantagens e desvantagens, tornando-se por isso impreterível adotar cuidados aquando do seu uso.

Consoante ficou explícito na Contextualização inicial, o objetivo primordial deste projeto passa então por produzir uma monitorização imediata assim que se começa a transpor/escrever o novo *source code*, de forma a criar um novo *software* o mais seguro e adaptável possível.

Isto implica o uso de TPC's de forma consciente nunca deixando de pensar nas melhores alternativas e na sua real utilidade para o projeto.

Com esta preocupação em mente, e atendendo às necessidades do projeto em causa, foi fundamental o uso de *External Libraries* do Java, com o objetivo de diminuir de forma drástica o tempo de desenvolvimento do novo *software*:

Nome do TPC/Java *Library*	Utilidade do TPC no *Software*
JAX WS RI Runtime	Usadas para manipular tudo aquilo que foi criado e compilado pelo *First-party Component* wsimport. No projeto, este FPC foi usado para criar todas as classes que permitem estabelecer uma ligação com o SOAP *Server* para a execução das várias operações do serviço de *Signature* CMD.
Javax JWS API	
JAX WS API	
Apache Commons Lang	Usada para manipular *Strings*. No projeto foi usado para manipular/extrair conteúdo do Certificado e ainda para extrair apenas os 9 dígitos pertencentes ao *User Id* (Número de Telemóvel) do utilizador da aplicação.

Note-se que se fez uso do *First-party Component* wsimport, que possibilitou criar todo o serviço SOAP que é depois responsável por estabelecer a ligação entre as operações de Assinatura de CMD disponíveis.

Apesar de não fazer parte de um TPC, acaba por criar dependências em termos de bibliotecas Java, que são necessárias importar para o correto funcionamento do *Test Command Line Program*.

Gestão dos *Third-party Components*

Uma vez importadas todas as *External Libraries* do Java necessárias para a otimização do *software* e consequente diminuição do ciclo de vida do seu desenvolvimento, entra-se na parte da gestão destes TPC's, onde se estabelece um conjunto de aplicações que vão auxiliar a definir a segurança de todo o projeto e dessa forma descomplicar também a sua utilização por futuros utilizadores/ *developers*.

Aplicação dos Ingredientes-chave para a Gestão dos TPC's

No primeiro projeto prático, abordou-se a ideia de estabelecer ingredientes-chave que constituem uma boa gestão dos *Third-party Components* de acordo com as várias fases do *Life Cycle* de um TPC.

A ideia passa agora por referir os vários métodos usados para o *software* Java desenvolvido, fazendo-se numa secção próxima uma ligação de todas as fases do *Life Cycle* dos vários TPC's com o *Software Development Life Cycle* e com isso compreendendo-se assim o peso de ambos e de que forma ajudam a manter o *software* o mais preparado possível para o consumidor final desde o início da sua criação.

O *software* Java foi então desenvolvido para ir ao encontro de todas estas fases, criando-se assim boas garantias em termos de produto final:

- Fase de *Maintain* que garantiu a supervisão dos vários TPC's usados no projeto, através da listagem dos mesmos e das suas respetivas informações.
- Fase de *Assess* que garantiu uma avaliação posterior de toda a lista de TPC's usados.
- Fase de *Mitigate* que garantiu a verificação de possíveis mitigações de um determinado TPC.
- Fase de *Monitor* que garantiu um acompanhamento contínuo dos vários TPC's, validando sempre a sua utilidade em termos de *software* e possíveis riscos que os mesmos possam estar a causar.

Fase *Maintain* - Manter uma lista de TPC's

Método	Descrição do Método e sua Utilidade
Criação de uma *Bill of Materials* (BOM) através de Ferramentas Automáticas	Dado que se programou na linguagem Java, a decisão passou por criar todo o projeto em modo ***Maven***, criando-se desde logo uma BOM automática através do ficheiro **pom.xml**. Este ficheiro enumera todas as dependências do *software* em causa e as suas respetivas versões, permitindo ainda ter todas estas bibliotecas na sua versão mais recente.
Uso de Identificadores Únicos para cada TPC	Esta prática encontra-se automaticamente em execução ao fazer-se uso do ***Maven Project Object Model,*** uma vez que cada dependência apresenta a sua própria identificação.

- A linguagem Java facilita muito todo o trabalho de manter a lista dos vários TPC's ao longo do projeto, dado que permite a integração com o ***Maven Project***.
- Esta BOM automaticamente criada e representada pelo ficheiro **pom.xml** facilita não só o uso de programa para futuros utilizadores/ *developers*, mas também permite criar hábitos de verificação das versões declaradas nesta ***Bill of Materials***, tendo-se assim os componentes sempre na sua versão mais atual.
- Com esta **BOM** garante-se ainda o uso de Identificadores Únicos para os vários componentes, evitando-se a existência de qualquer tipo de colisões entre eles.

Fase *Assess* – Avaliar o risco de segurança

Nome do TPC/Java *Library*	Data da Última Atualização	N.º de *Artifacts* que usam o TPC	Contactos *Developers*	Vulnerabilidades Recentes
JAX WS RI Runtime	Maio de 2020	243	Sim	Não
Javax JWS API	Junho de 2018	32	Sim	Não
JAX WS API	Outubro de 2018	764	Sim	Não
Apache Commons Lang	Março de 2020	16.809	Sim	Não

- Todas as librarias são relativamente recentes e apresentam um bom suporte por parte da comunidade no geral.
- O número de *artifacts* que usam estas librarias são relativamente altos para a maioria dos componentes, o que salienta ainda mais este suporte e confere ao componente uma ideia de possível manutenção estável e minimamente atualizada.
- Todos os componentes exibem o contacto dos vários *developers*, o que por si só pode ser útil para futuras necessidades em termos de licenças e manipulação do próprio TPC.
- Apesar de ter sido complicado tentar detetar possíveis vulnerabilidades, não se verificou a existência de vulnerabilidades recentes diretamente relacionadas com os componentes em si.

Note-se que uma vez que todo o projeto foi pensado para funcionar com o *Maven Project*, o mais importante acaba por ser a confirmação da versão em uso e das suas possíveis vulnerabilidades. Isso pode ser encontrado de forma detalhada em http://maven.apache.org/security.html.

Fase *Mitigate* – Mitigar ou aceitar o risco

Relativamente à fase de mitigar os possíveis riscos existentes para os vários TPC's, não existiu qualquer método específico, a não ser a garantia de que todas as bibliotecas usadas se encontram nas suas versões mais recentes, já que não se encontraram quaisquer vulnerabilidades potencialmente perigosas ao projeto em causa.

Este processo pode ser feito manualmente através da edição da BOM no ficheiro **pom.xml**. O *website* https://mvnrepository.com permite iniciar a procura das várias bibliotecas em uso e verificar a versão mais recente para cada uma delas. Feito isso, basta alterar o campo **version** para a libraria em si, se necessário.

Fase *Monitor* – Monitorizar mudanças aos TPC's

De modo a assegurar uma monitorização de possíveis mudanças nestes *Third-party Components* existe um conjunto de medidas que podem e devem ser tomadas após a conclusão do *software.*

- Verificar a possível *End of Life* (EOL) dos TPC's, analisando se o componente em si foi descontinuado ou apenas a sua versão atual. O próprio IntelliJ IDEA consegue fazer esta verificação através dos vários *warnings* que são mostrados ao longo do código.
- Atentar a possíveis vulnerabilidades que possam surgir para os componentes usados no projeto.
- Manter um perfil de risco no geral de forma a se poder comunicar futuras vulnerabilidades.

Third-party Components Life Cycle vs *Software Development Life Cycle*

Relacionar estes dois ciclos é meio caminho andado para compreender a importância dos mesmos na construção de todo o projeto e na ajuda que fornecem à segurança final do *software*.

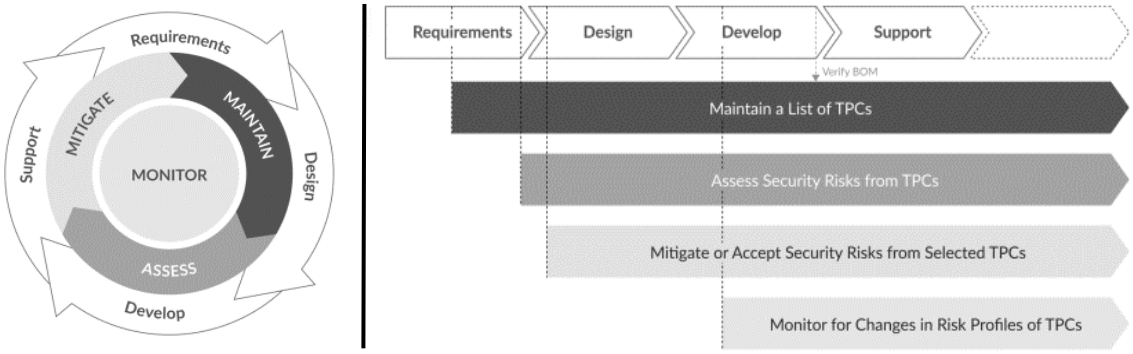


Figura 1 Relação das fases do Ciclo de Vida de um TPC e o SDLC

- A fase de ***Mantain*** do *Third-party Components Life Cycle* deve começar aquando da fase de ***Requirements*** do *Software Development Life Cycle*, dado que dessa forma consegue-se que os requisitos funcionais possam ditar o uso de TPC's específicos.
  - No projeto em causa a análise dos requisitos representados sob a forma de código Python foi decisiva para criar uma lista de TPC's inicial/prematura.

- A fase de ***Assess*** em que é feita a avaliação dos riscos de segurança inicia-se assim que um TPC candidato é identificado através da listagem anterior.
  - No projeto em causa consistiu em enumerar os TPC's escolhidos, a data da sua última atualização, o número de *artifacts* que fazem uso do TPC em si, etc. Com esta avaliação cria-se desde logo um perfil de possível risco relativamente aos demais componentes.

- A etapa de *Mitigate* acompanha o projeto desde a fase do *Design* do SDLC até ao restante das fases, dado que pode ser necessário salvaguardar certos níveis do código e do próprio projeto em si para que consiga mitigar os riscos de um TPC que deve ser usado.
    - No projeto em causa consistiu em analisar os TPC's escolhidos, garantindo que os mesmos se encontravam nas suas versões mais recentes.

- Finaliza-se com a fase de *Monitor* que visa acompanhar a lista de TPC's, em que pode ser necessário acionar a avaliação de riscos no caso de TPC's novos ou atualizados serem adicionados à BOM.
    - No projeto em causa este acompanhamento foi validado antes de se dar por encerrado o produto final e de se entrar na fase de *Support* do SDLC.
    - Por isso que existe a sub-etapa "*Verify BOM"* antes de se entrar na fase de *Support*. Esta verificação é feita e dessa forma entrega-se o projeto final com as versões mais recentes dos vários TPC's.

Ferramentas e Indicadores de Qualidade de *Software* e/ou Testes de *Software*

Características de Qualidade de *Software*

No segundo projeto foi analisado uma ISO 25000 que surgiu no sentido para definir um modelo de qualidade num produto de *software*. Esse padrão pode e deve ser aplicado neste projeto, estipulando e aplicando os oito fatores principais de qualidade, de forma a obter uma ideia mais substancial e tornar um produto/projeto o melhor possível.

Característica de Qualidade de *Software*	Definição da Qualidade de *Software*	Utilização no Projeto
Functional Suitability	O produto apresenta integridade, correção e adequa-se ao propósito pretendido.	O projeto tem como base a assinatura de documentos e facilmente se adequa a esse propósito.
Performance Efficiency	O produto faz uma boa gestão dos recursos ao longo do tempo, consoante a capacidade disponível.	O projeto tem uma *footprint* mínima de recursos computacionais, efetuando ligação ao servidor apenas nos momentos fulcrais.
Compatibility	O produto tem uma boa compatibilidade em termos de troca de informações e/ou execução de funções necessárias perante os vários sistemas a compartilhar o mesmo ambiente de hardware ou software.	O projeto foi desenvolvido em Java, fornecendo uma compatibilidade de funcionar em milhares de milhões de dispositivos.

Usability	O produto demonstra facilidade em reconhecer o propósito/algoritmo do programa em si, tendo em conta o visual que é apresentado ao utilizador consoante as necessidades pensadas para o mesmo e ainda na acessibilidade caso o utilizador possua alguma incapacidade.	O menu feito de maneira acessível, com escolhas simples de entender e no máximo necessitando *input* fulcral, funcionando autonomamente nos outros casos.
Reliability	O produto apresenta maturidade, disponibilidade, tolerância à falha e capacidade de recuperar em situações necessárias.	O programa oferece uma panóplia de *handlers* e exceções de forma a ter uma tolerância à falha correspondente à complexidade do mesmo.
Security	O produto deve garantir a confidencialidade, integridade, não repúdio, autenticação e ainda um bom ambiente de testes.	O projeto utiliza as *KeyStores* da API Java que fornecem uma segurança complexa de forma a proteger as informações dos certificados e das assinaturas correspondentes.
Maintainability	O produto deve ser modular de modo a poder ser reutilizável e dessa maneira ser mais fácil de testar, analisar e modificar.	O projeto contém vários módulos e classes para interligar com o *web service* da AMA e também as várias etapas do projeto.
Portabilty	O produto deve estar pronto para ter partes substituíveis, adaptando-se a novas e futuras instalações/atualizações.	O projeto pode e deve ser atualizado com *updates* às dependências e atualizações aos módulos de *web servicing*.

Métricas de Qualidade de *Software*

Apesar da secção passada falar duma norma utilizada, tem alguma ambiguidade e inconvenientes que podem ser complicados para um projeto com uma seriedade e complexidade maior. Assim surgem oito métricas de qualidade de código para serem medidas duma maneira mais automatizada.

1. *Code Coverage*: Todo o código deve ter testes de forma a automatizar e verificar o código. Esse *code coverage* é um indicador de *Functional Suitability* e *Reliability*. Dessa forma, o código tem as várias funções separadas em unidades de forma a verificar cada parte do código de forma independente e efetuar testes a cada parte do *software* antes de correr todos os passos/funções.

2. *Abstraction Interpretation*: Uma tecnologia mais recente é a possibilidade de encontrar problemas com uma análise computadorizada que nos avisa de todos os tipos de erros de programação no fluxo dum programa. Assim, com a ajuda do *software* IntelliJ IDEA utilizado para a produção e desenvolvimento deste projeto, podemos sempre analisar problemas de segmentação de memória, apontadores nulos, *overflows* de *buffer* ou até mesmo conexões e exceções nunca utilizadas. Estes erros que poderiam ter uma severidade considerável ficam assim calculados e analisados por um programa (IDE) de maneira eficiente. Esta métrica é mapeada no atributo *Reliability* falado anteriormente.

3. *Cyclomatic Complexity*: Uma das métricas mais antigas na classificação de *software* é a complexidade de caminhos independentes possíveis num programa. O nosso programa em causa tem vários ciclos e caminhos dada a sua natureza de existirem menus, momentos cíclicos quando existem erros ou exceções e dessa forma considera-se um programa complexo e teve de existir um cuidado especial recorrendo a ferramentas para ter a certeza de que os testes englobavam todos os caminhos possíveis.

4. *Compiler Warnings*: Avisos do compilador são sempre importantes e o compilador JAVA não é diferente nesse ramo. Dessa forma, todo o programa foi pensado com a geração de erros e avisos e os mitigar logo no processo de desenvolvimento do mesmo.

5. *Coding Standards*: A manutenção de *software* deve ser das coisas mais difíceis e que consome mais tempo, porque é realmente difícil de entender a intenção do código muito depois de ser escrito, logo é necessário reduzir essa curva de aprendizagem com *standards* e regras que todos os engenheiros devem seguir.

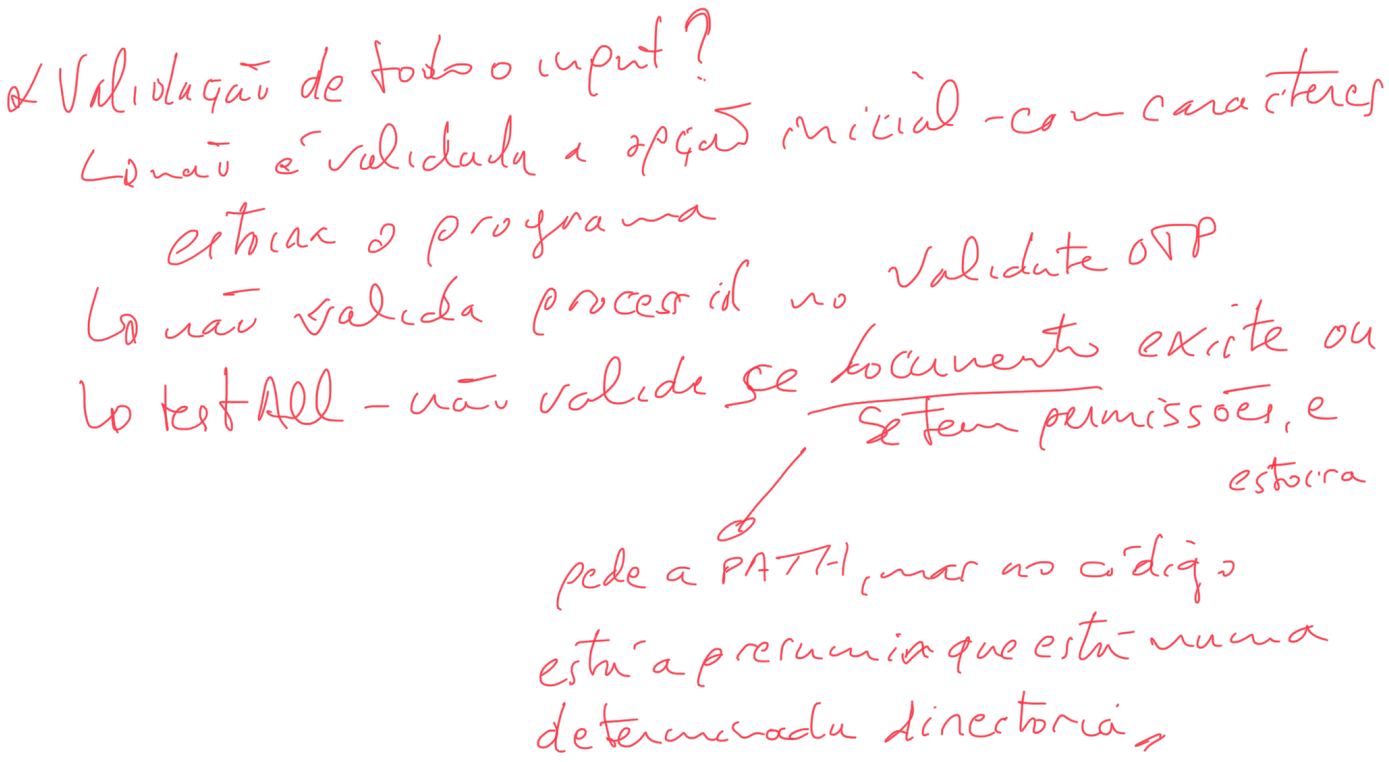
   O nosso projeto não diverge de tal, sendo que obedece a todos esses *standards*, como a geração de documentação seguindo os parâmetros internacionais do *JavaDoc* e assim criou-se as páginas HTML da documentação. As classes em *Java* seguiram as regras de espaçamento corretas, desde linhas em branco separando métodos, espaços pelos vários parenteses. O uso de *camel case* foi prática geral nos nomes dos métodos, classes, variáveis, todos os objetos desta linguagem de programação. Estas decisões todas tornam o programa não só mais legível, como mais fácil de entender a algoritmia por detrás do *source code*.

6. *Code Duplication*: É comum existir a tentação dum engenheiro de *software* copiar pedaços de código e fazer umas modificações pequenas sobre o mesmo em vez de generalizar a funcionalidade num outro método. Desta forma, a manutenção pode demorar mais tempo quando se adiciona uma funcionalidade ou corrige-se um *bug*, que tem de ser agora feito de maneira análoga nos outros lugares copiados. Desta maneira, todo o nosso código-fonte do projeto foi analisado por uma ferramenta de *code analysis* que nos informou que não existe essa duplicação.

7. *Fan Out*: Um programa de *software* pode ter módulos e componentes externos que pode tornar uma interdependência maior e ser mais difícil a modificação do mesmo para corrigir *bugs* dada a cadeia de dependência difícil de corrigir. O nosso programa dividiu-se em dois *packages*, um com dependências e classes feitas automaticamente para funcionar com o *web service* fornecido

pelo WSDL da Agência para a Modernização Administrativa e apenas tem três dependências, mas como é um código que é compilado automaticamente sem poder alterar, torna-se um *low fan out*. No outro *package* apenas são utilizados poucos *imports* da própria API Java, melhorando todo o programa para não ter uma dependência grande em vários componentes externos.

8. ***Security***. O nosso programa tinha de ter esta métrica de maneira importante dado que envolve códigos de segurança, assinatura de documentos, processos únicos e certificados X.509 do utilizador a usar. Dessa forma, a utilização das próprias ferramentas Java, desde as classes *Certificate, KeyStore, PublicKey, Signature*, que nos ajudaram a não só manter o projeto seguro, mas simples de entender numa leitura futura. São essas classes responsáveis para toda a criptografia e matemática envolvente na segurança do projeto e dos seus dados sensíveis passíveis quando o programa corre. Assim, o programa segue métricas internacionais e bem conhecidas que protegem qualquer falha que poderia ter acontecido caso a implementação fosse feita de raiz.



Funcionamento *Command Line Program*

O programa *Test Command Line* foi criado pensando-se sempre em simplificar a experiência com o utilizador, através de uma abordagem diferente do programa original em Python que trabalha em modo de comandos. Dessa forma, o programa em Java surge sob a forma de menus, no qual são pedidos ao utilizador os dados necessários para executar as demais operações do serviço em si, devolvendo-se depois o resultado correspondente à operação pedida.

Para simplificar e tornar público o funcionamento do *Command Line Program*, criou-se um ficheiro README totalmente detalhado com as seguintes secções:

1. Implementação/Estrutura do Programa que detalha como foi pensada a implementação e estrutura com todas as diretorias e os ficheiros que fazem parte do *software*.

2. Inicialização/Utilização do Programa que exemplifica como inicializar o programa pela primeira vez e como espoletar a sua posterior utilização.

    a. Exemplo da utilização da opção *"Run All Commands"* que demonstra o uso da opção em causa e os dados pedidos pelo programa em si, bem como o *output* gerando aquando da sua utilização.

3. Notas que enumera um conjunto de informações extra, mas que são úteis para o funcionamento do programa.

Este repositório encontra-se então público em https://github.com/uminho-miei-engseg-19-20/Grupo5/tree/master/Projetos/Projeto%203/CMD-SOAP

Referências

AGREEMENT AND PLAN OF MERGER. (25 de julho de 2019). Obtido de Law Insider: https://www.lawinsider.com/contracts/8NVe5f8UNc0#third-party-components

Boff, R. J. (s.d.). *Padrões de Codificação no Desenvolvimento de Sistemas*. Obtido de Micreiros: https://micreiros.com/padroes-de-codificacao-no-desenvolvimento-de-sistemas/

ckan. (s.d.). Obtido de Python coding standards: https://docs.ckan.org/en/2.8/contributing/python.html

Guido van Rossum, B. W. (5 de julho de 2001). *PEP 8 – Style Guide for Python Code*. Obtido de https://www.python.org/dev/peps/pep-0008/

List of tools for static code analysis. (8 de maio de 2020). Obtido de Wikipedia: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Mohino, J. d., Higuera, J. B., Higuera, J. R., & Montalvo, J. A. (2019). The Application of a New Secure Software Development Life Cycle (S-SDLC) with Agile Methodologies. *Electronics 8.11*, 2.

Rezvi, M. (7 de janeiro de 2020). *gitconnected*. Obtido de Write Better Code with Coding Standards: https://levelup.gitconnected.com/write-better-code-with-coding-standards-546faf3fd4d1

Rezvi, M. (07 de janeiro de 2020). *Write Better Code with Coding Standards*. Obtido de gitconnected: https://levelup.gitconnected.com/write-better-code-with-coding-standards-546faf3fd4d1

SDLC (Software Development Life Cycle) Phases, Methodologies, Process, And Models. (10 de novembro de 2019). Obtido de Software Testing Help: https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/

SDLC (Software Development Life Cycle) Tutorial: What is, Phases, Model. (s.d.). Obtido de Guru99: https://www.guru99.com/software-development-life-cycle-tutorial.html

Sengayire, P. (25 de novembro de 2019). *CODING STANDARDS AND CONVENTIONS IN SOFTWARE DEVELOPMENT TEAM*. Obtido de Medium: https://medium.com/@psengayire/the-importance-of-coding-standards-and-conventions-in-the-software-development-team-how-they-can-5d252556a05

Svilicic, B., Rudan, I., Frančić, V., & Doričić, M. (2019). Shipboard ECDIS Cyber Security: Third-Party Component Threats. *Scientific Journal of Maritime Research*, 176-180.

The Hidden Risk in All IoT Devices: Third-Party Components. (25 de junho de 2019). Obtido de Sternum: https://www.sternumiot.com/blog/2019/6/25/third-party-components-the-hidden-risk-in-all-iot-devices

TIOBE Coding Standard Methodology. (s.d.). Obtido de TIOBE: https://www.tiobe.com/coding-standard-methodology/

Xie, W., Hu, J., Kudjo, P. K., Yu, L., & Zeng, Z. (2018). A New Detection Method for Stack Overflow Vulnerability Based on Component Binary Code for Third-Party Component. *Institute of Electrical and Electronics Engineers*, 1-5.

AGREEMENT AND PLAN OF MERGER. (2019, julho 25). Retrieved from Law Insider: https://www.lawinsider.com/contracts/8NVe5f8UNc0#third-party-components

Boff, R. J. (n.d.). *Padrões de Codificação no Desenvolvimento de Sistemas*. Retrieved from Micreiros: https://micreiros.com/padroes-de-codificacao-no-desenvolvimento-de-sistemas/

ckan. (n.d.). Retrieved from Python coding standards: https://docs.ckan.org/en/2.8/contributing/python.html

Guido van Rossum, B. W. (2001, julho 5). *PEP 8 – Style Guide for Python Code*. Retrieved from https://www.python.org/dev/peps/pep-0008/

List of tools for static code analysis. (2020, maio 8). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Mohino, J. d., Higuera, J. B., Higuera, J. R., & Montalvo, J. A. (2019). The Application of a New Secure Software Development Life Cycle (S-SDLC) with Agile Methodologies. *Electronics 8.11*, 2.

Rezvi, M. (2020, janeiro 7). *gitconnected*. Retrieved from Write Better Code with Coding Standards: https://levelup.gitconnected.com/write-better-code-with-coding-standards-546faf3fd4d1

Rezvi, M. (2020, janeiro 07). *Write Better Code with Coding Standards*. Retrieved from gitconnected: https://levelup.gitconnected.com/write-better-code-with-coding-standards-546faf3fd4d1

SDLC (Software Development Life Cycle) Phases, Methodologies, Process, And Models. (2019, novembro 10). Retrieved from Software Testing Help: https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/

SDLC (Software Development Life Cycle) Tutorial: What is, Phases, Model. (n.d.). Retrieved from Guru99: https://www.guru99.com/software-development-life-cycle-tutorial.html

Sengayire, P. (2019, novembro 25). *CODING STANDARDS AND CONVENTIONS IN SOFTWARE DEVELOPMENT TEAM*. Retrieved from Medium: https://medium.com/@psengayire/the-importance-of-coding-standards-and-conventions-in-the-software-development-team-how-they-can-5d252556a05

Svilicic, B., Rudan, I., Frančić, V., & Doričić, M. (2019). Shipboard ECDIS Cyber Security: Third-Party Component Threats. *Scientific Journal of Maritime Research*, 176-180.

The Hidden Risk in All IoT Devices: Third-Party Components. (2019, junho 25). Retrieved from Sternum: https://www.sternumiot.com/blog/2019/6/25/third-party-components-the-hidden-risk-in-all-iot-devices

TIOBE Coding Standard Methodology. (n.d.). Retrieved from TIOBE: https://www.tiobe.com/coding-standard-methodology/

Xie, W., Hu, J., Kudjo, P. K., Yu, L., & Zeng, Z. (2018). A New Detection Method for Stack Overflow Vulnerability Based on Component Binary Code for Third-Party Component. *Institute of Electrical and Electronics Engineers*, 1-5.