Universidade Federal da Bahia
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

GIFFLAR: UM FRAMEWORK PARA GERAÇÃO DE CONTRATOS INTELIGENTES EM TEMPO DE EXECUÇÃO

Levy Marlon Souza Santiago

DISSERTAÇÃO DE MESTRADO

Salvador
18 de abril de 2023

LEVY MARLON SOUZA SANTIAGO

GIFFLAR: UM FRAMEWORK PARA GERAÇÃO DE CONTRATOS INTELIGENTES EM TEMPO DE EXECUÇÃO

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Profa. Dra. Fabíola Gonçalves Pereira Greve

Salvador
18 de abril de 2023

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências e Tecnologias Prof. Omar Catunda, SIBI - UFBA.

C871 Santiago, Levy

Gifflar: Um framework para geração de contratos inteligentes em tempo de execução – Salvador, 2023.

182 f.

Orientadora: Profª. Drª Fabíola Gonçalves Pereira Greve

Dissertação (Mestrado) – Universidade Federal da Bahia. Instituto de Computação, 2023.

1. Blockchain. 2. Smart Contract. 3. Framework. 4. Geração de Código. I. Greve, Fabíola Gonçalves Pereira. II. Universidade Federal da Bahia. III. Título.

CDU:616-083:173.4

TERMO DE APROVAÇÃO

LEVY MARLON SOUZA SANTIAGO

GIFFLAR: UM FRAMEWORK PARA GERAÇÃO DE CONTRATOS INTELIGENTES EM TEMPO DE EXECUÇÃO

Esta Dissertação de Mestrado foi julgada adequada à obtenção do título de Mestre em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia.

Salvador, 18 de abril de 2023

Profa. Dra. Fabíola Gonçalves Pereira Greve
Instituto de Computação
Universidade Federal da Bahia

Prof. Alex Borges Vieira
Instituto de Ciências Exatas
Universidade Federal de Juiz de Fora

Prof. Dr. Rodrigo Rocha Gomes e Souza
Instituto de Computação
Universidade Federal da Bahia

Dedico este trabalho aos meus pais.

AGRADECIMENTOS

Agradeço primeiramente a Deus por me conceder saúde, força, condição e capacidade para realizar todas as atividades enquanto mestrando.

Agradeço aos meus pais, minha mãe Rosilene e meu pai Milton, por terem me permitido vir a este mundo e me educado da melhor forma possível e a todos os meus familiares, em especial àqueles que convivem comigo, minha irmã Mayana e meu irmão Wagner que sempre me apoiaram, e àqueles que já nos deixaram fisicamente mas estão sempre enviando vibrações positivas.

Ao Ensinamento da Seicho-No-Ie, por me ajudar a reconhecer minhas capacidades, descobrir o verdadeiro motivo de estudar, ter concentração nas atividades, superar obstáculos, vencer dificuldades e ser uma pessoa melhor.

À Universidade Federal da Bahia (UFBA) por conceder a infraestrutura necessária para o desenvolvimento de pesquisas durante o curso e a todos os funcionários que a mantêm sempre limpa.

À Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) pela de bolsa de projeto recebida durante um período do mestrado.

À Professora Doutora Fabíola Gonçalves Pereira Greve, pela orientação, pela grande amizade e apoio no desenvolvimento do projeto de mestrado. Levarei para sempre comigo toda a experiência vivida.

À banca avaliadora de qualificação e dissertação, Prof. Rodrigo Rocha Gomes e Souza, Prof. Cláudio Nogueira Sant'Anna e Prof. Alex Borges Vieira pelas importantes recomendações de melhorias para o projeto.

A todos(as) os(as) professores(as) do curso que fazem um trabalho grandioso no ensino e orientação de alunos.

A todos aqueles que fazem parte do Grupo de Algoritmos e Computação Distribuída (GAUDI) que estão sempre ajudando um ao outro e sempre dispostos a publicar mais um artigo científico.

A todos os desenvolvedores e desenvolvedoras que participaram do teste de usabilidade do Gifflar, minha eterna gratidão.

"Aquele que mantém a sua mente desperta e vigilante e procura sempre melhorar a sua vida e o seu trabalho progride infalivelmente, mas aquele que se acomoda só terá que regredir. Na vida há somente duas alternativas: progredir ou regredir. Não há estacionamento ou meio-termo."

—MASAHARU TANIGUCHI (Fonte: O Livro dos Jovens)

RESUMO

Blockchain é uma tecnologia disruptiva que permite transações diretas entre entidades distribuídas sem a necessidade de uma terceira parte confiável, oferecendo uma rede descentralizada segura. Contratos inteligentes (SmCs) são códigos executáveis hospedados na blockchain e permitem a implementação de aplicações descentralizadas, em diferentes domínios. Devido à complexidade de construção dos SmCs, é necessário buscar formas de facilitar e tornar robusto esse processo de desenvolvimento inovador. Muitos projetos da literatura propuseram soluções baseadas em geração automática de código a partir de uma modelagem de alto nível, como diagramas. Porém, esta abordagem traz consigo uma constante necessidade de uma pessoa para modelar os contratos e realizar o processo de compilação e implantação. Esta dissertação de mestrado apresenta o Gifflar, um *framework* para geração automática de código de SmC em tempo de execução, que permite a um sistema escrever, compilar e implantar SmCs numa rede blockchain com uma aplicação ainda em execução, o que reduz a frequente precisão do desenvolvedor de SmC e possibilita automatizar ainda mais o processo de desenvolvimento de SmC. Uma das suas principais ferramentas é a biblioteca Solgen, que é estruturada em componentes que implementam padrões de projeto para melhor definir a responsabilidade de cada uma dessas partes. A Solgen oferece uma API que permite ao desenvolvedor modelar, gerar códigos e gerenciar os SmCs através de métodos que abstraem até certo nível o código do SmC utilizando o JSON como modelo para a geração de código. Em complemento, foram realizadas duas avaliações para validar o projeto: (i) uma avaliação de usabilidade da Solgen e (ii) uma avaliação conceitual de aplicação do Gifflar em outros projetos. Até onde se sabe, este *framework* é um dos primeiros a permitir gerar SmCs dinamicamente em tempo de execução, assim contribuindo para o estado da arte ao abordar um novo paradigma, onde os sistemas podem atuar como desenvolvedores de contratos inteligentes.

Palavras-chave: Blockchain. Contrato Inteligente. Framework. Geração Automática de Código. Aplicações Descentralizadas.

ABSTRACT

Blockchain is a disruptive technology that offers a secure decentralized network, allowing direct transactions between distributed entities without needing a trusted third party. Smart contracts (SmC) are executable codes hosted on the blockchain and enable the implementation of several decentralized applications in different domains. However, due to the complexity of SmC construction, seeking ways to facilitate and make this innovative development process robust is necessary. Many projects in the literature proposed solutions based on automatic code generation from high-level modelings, such as diagrams. However, this approach brings with it a constant need for a person to model the contracts. This paper presents Gifflar, a framework for generating SmC code on the fly, in such a way that it allows a system to write, compile and implement blockchain SmCs with an application still running, which reduces the frequent precision of the SmC developer and allows to automate the SmC development process further. One of the main tools is a component-structured library called Solgen that implements design patterns better to define the responsibility of each of these parts. The Solgen offers an API that allows the developer to model, generate code and manage SmCs through methods that abstract to a certain level the SmC code using the JSON as a model to code generation. In addition, project evaluations were carried out to validate the project: (i) a usability evaluation of the Solgen and (ii) a conceptual evaluation of Gifflar's application in other projects. As far as we know, this framework is one of the first to allow dynamically generating SmCs on the fly, thus contributing to state of the art by approaching a new paradigm where systems can act as smart contract developers.

Keywords: Blockchain. Smart Contract. Framework. Automatic Code Generation. Decentralized Applications.

SUMÁRIO

Capítulo 1—Introdução 1

1.1 Objetivos . . . 5
1.1.1 Objetivo Geral . . . 5
1.1.2 Objetivos Específicos . . . 5
1.1.3 Publicações Obtidas . . . 6
1.2 Organização . . . 6

Capítulo 2—Fundamentação Teórica e Trabalhos Relacionados 9

2.1 Blockchain . . . 9
2.1.1 Chaves Pública e Privada . . . 10
2.1.2 Cadeia de Blocos . . . 11
2.1.3 Consenso e Mineração . . . 11
2.1.4 Contrato Inteligente . . . 12
2.1.5 Vulnerabilidade de Contratos Inteligentes . . . 13
2.2 Programação orientada a objetos . . . 16
2.3 Bibliotecas . . . 16
2.4 *Frameworks* . . . 16
2.5 Princípios SOLID . . . 17
2.6 JavaScript, JSON e TypeScript . . . 19
2.7 Geração automática de código . . . 22
2.8 Trabalhos Relacionados . . . 23
2.9 Resumo do Capítulo . . . 26

Capítulo 3—Gifflar: Geração de Contratos Inteligentes em Tempo de Execução 29

3.1 Estrutura do Gifflar . . . 29
3.1.1 Núcleo (*Core*) . . . 30
3.1.2 Solgen (Solidity Generator) . . . 31
3.1.2.1 Linguagem de Contrato Inteligente . . . 31
3.1.2.2 Modelo para Geração de Contratos . . . 32
3.1.2.3 Aplicação dos Princípios SOLID . . . 33
3.1.2.4 Componentes da Solgen . . . 34
3.1.2.5 Fluxo de Etapas . . . 35
3.1.2.6 Fluxo Interno de Componentes na Etapa de Escrita . . . 36
3.1.3 Uso do *Framework* . . . 37
3.2 Desafios da geração automática de SmC . . . 39

3.3 Código, versão e propriedades suportadas 42
3.4 Quando utilizar o Gifflar 42
3.5 Resumo do capítulo 44

Capítulo 4—Estratégia das Avaliações 47

4.1 Estratégia de Avaliação de Usabilidade 47
4.1.1 Questões de pesquisa 48
4.1.2 Elaboração da Estratégia 49
4.1.2.1 Questões da Entrevista 49
4.1.2.2 Tokens de Usabilidade 51
4.1.3 Execução da Estratégia 53
4.1.3.1 Tasks 53
4.1.3.2 Ambiente de Desenvolvimento 54
4.1.3.3 Participantes 55
4.1.3.4 Protocolo 56
4.2 Estratégia de Avaliação de Aplicabilidade 57
4.2.1 Condução da Pesquisa 57
4.2.2 Triagem dos Artigos 58
4.2.3 Etapas da Pesquisa 58

Capítulo 5—Resultados das Avaliações 61

5.1 Avaliação de Usabilidade 61
5.1.1 QP1: Compreensibilidade 62
5.1.2 QP2: Abstração 64
5.1.3 QP3: Reusabilidade 66
5.1.4 QP4: Aprendizagem 67
5.2 Discussão 68
5.3 Ameaças à Validade 70
5.3.1 Validade de Construção 71
5.3.2 Validade Interna 71
5.3.3 Validade Externa 73
5.4 Avaliação de Aplicabilidade 73
5.4.1 Busca e Seleção dos Artigos 73
5.4.2 Projetos 74
5.4.2.1 IoTCocoa 74
5.4.2.2 Cadeia de Suprimentos na Indústria Farmacêutica 76

Capítulo 6—Conclusões e Trabalhos Futuros 79

6.1 Conclusões 79
6.2 Trabalhos Futuros 81
6.2.1 Grupo 1 82
6.2.2 Grupo 2 83

6.3 Dificuldades e Obstáculos 84

Apêndice A—Tasks da avaliação de usabilidade 95

A.1 Task 1: Criando Gifflar Model 95
A.2 Task 2: Desenvolvendo Contrato Inteligente com Gifflar Contract Model . 95
A.3 Task 3: Criando Gifflar Script para implantação do contrato na blockchain 96
A.4 Task 4: Configurando escrita e compilação do contrato 97
A.5 Task 5: Construindo script de implantação na rede blockchain testnet . . 97

Apêndice B—Exemplo base de código Solidity 99

Apêndice C—Exemplo de código final após tasks 101

Apêndice D—Tabelas completas de resultados da avaliação de usabilidade 105

LISTA DE FIGURAS

2.1 Demonstração da estrutura da cadeia de blocos 11

3.1 Estrutura do Gifflar *core* 30
3.2 Exemplo do modelo JSON padrão do Gifflar 33
3.3 Relação de componentes da Solgen 35
3.4 Organização de etapas da Solgen 36
3.5 Diagrama de sequência da etapa de escrita do Gifflar 37
3.6 Exemplo de trecho do *gifflarconfig.json* 38
3.7 Exemplo de uso da Solgen como pacote no TypeScript 39
3.8 Código gerado pelo modelo da Figura 3.2 39
3.9 Fluxograma de decisão para uso do Gifflar 43

LISTA DE TABELAS

2.1 Comparação de trabalhos com base no tipo de ferramenta, etapas de criação do SmC e dinamicidade 24
2.2 Comparação de trabalhos com base na estrutura de escrita, linguagens de SmC, genericidade e geração em tempo de execução 25

3.1 Propriedades do *Solidity* atualmente suportadas ou não pelo *Gifflar*. . . . 42

4.1 Estatísticas de conhecimento dos participantes. 56

5.1 Respostas das perguntas da entrevista. 62
5.2 Tokens de usabilidade mais frequentes 63
5.3 Artigos selecionados na pesquisa de avaliação de aplicabilidade 74

D.1 Respostas das perguntas da entrevista (versão detalhada). 105
D.2 Tokens de usabilidade mais frequentes (versão detalhada). 106

LISTA DE SIGLAS

ABI	Interface Binária de Aplicação
API	Interface de Programação de Aplicação
DApp	Aplicação Descentralizada
EVM	Máquina Virtual Ethereum
IoT	Internet das Coisas
JSON	Notação de Objeto do *JavaScript*
OOP	Programação Orientada a Objetos
PoW	Prova de Trabalho
P2P	Ponto a Ponto
REST	Transferência de Estado Representacional
SmC	Contrato Inteligente
SNMP	Protocolo Simples de Gerência de Rede
XML	Linguagem Extensível de Marcação

Capítulo 1

INTRODUÇÃO

A blockchain é uma tecnologia que está se aprimorando cada vez mais e continuará a crescer nos próximos anos, oferecendo novos casos de uso especialmente suportados por ela (GARTNER, 2019). Introduzida por Satoshi Nakamoto com a Bitcoin (NAKAMOTO; BITCOIN, 2008), a blockchain fornece uma maneira de realizar transações para outras entidades sem a necessidade de terceiros para controlar as operações. A blockchain em si cria uma estrutura à prova de violação, integrando diversos conceitos de computação distribuída tolerante a falhas e segurança computacional já consolidados na literatura, tais como protocolos de consenso distribuído e criptografia de chave assimétrica (GREVE et al., 2018).

Um dos conceitos importantes, introduzidos no bojo da nova tecnologia, são os contratos inteligentes (SmCs - do inglês *Smart Contracts*). Estes foram idealizados por Szabo (1997), mas somente implementados por Buterin et al. (2014) com a plataforma de blockchain Ethereum, que oferece as ferramentas necessárias para a execução desses *scripts* complexos em uma rede descentralizada. Na Ethereum, os SmCs são escritos na linguagem Solidity, uma linguagem Turing-completa, e são compilados para uma linguagem de baixo nível específica, para que sejam executados na Máquina Virtual Ethereum (EVM - do inglês *Ethereum Virtual Machine*). Os SmCs abriram oportunidades de levar a mesma segurança oferecida pela blockchain da criptomoeda Bitcoin para diferentes áreas, como a saúde, indústria, cadeia de suprimentos, governo e Internet das Coisas (IoT - do inglês *Internet of Things*) permitindo a criação de Aplicações Descentralizadas (DApps - do inglês Decentralized Applications). Tais aplicações funcionam sem uma autoridade central pois utilizam a blockchain como principal tecnologia.

Como precursora do SmC, a Ethereum recebeu muita atenção, de modo que muitas bibliotecas e *frameworks* foram propostos para melhorar o desenvolvimento de SmCs. As bibliotecas são uma coleção de códigos pré-escritos que, a partir de uma interface de programação de aplicação (API - do inglês *Application Programming Interface*), facilitam a resolução de algumas tarefas e também deixam o código mais limpo (MCFARLAND, 2011). Como um conjunto de componentes e bibliotecas que podem ser reutilizados

(JOHNSON; FOOTE, 1988), os *frameworks* são muito importantes para o crescimento da tecnologia agilizando o processo de desenvolvimento dos SmCs.

Ainda que tenham surgido ferramentas para auxiliar o desenvolvimento de SmCs, são poucos os desenvolvedores que possuem esse conhecimento devido ao alto nível de complexidade de construção de SmCs (MAVRIDOU et al., 2019). Além disso, a DApp é uma abordagem nova, comparada às aplicações centralizadas que são criadas com programação em linguagens já conhecidas e utilizadas há muito tempo. Ou seja, mesmo que a blockchain tenha evoluído em alguns contextos (SmCs, DApps, etc), ainda existe uma demanda com relação a desenvolvedores para esta área.

Por esse motivo, muitos trabalhos vêm propondo diferentes maneiras de facilitar o desenvolvimento de SmC, através do processo de geração automática de código. Nessa abordagem, parte-se da definição de um modelo que é utilizado para representar o SmC abstraindo certos recursos da linguagem e, ao defini-lo, propõe-se uma funcionalidade que aciona um mecanismo para traduzir o modelo em código de SmC. Este modelo funciona como uma representação abstrata do SmC, onde o nível de abstração pode variar. Quanto maior o nível da abstração, mais compreensível fica para o ser humano; quanto menor o nível de abstração de um modelo, menor é o grupo de pessoas que compreendem tal modelo.

Dentre o conjunto de trabalhados relacionados, destacam-se o de Luo et al. (2019) que propõem uma forma de gerar código de SmC em JavaScript a partir de uma estrutura textual que pode ser entendida e utilizada pelo ser humano para modelar SmCs. Outro exemplo é a ferramenta FSolidM criada por Mavridou e Laszka (2018), que permite modelar SmCs através de um diagrama de máquina de estados finita para então gerar o código de SmC na linguagem Solidity. Os autores Ladleif, Friedow e Weske (2020) criam um protótipo de uma ferramenta que permite a modelagem dos SmCs em Notação e Modelo de Processos de Negócios (BPMN) a fim de posteriormente interpretar o modelo e gerar o código Solidity.

Todos estes trabalhos relacionados usam estruturas de alto nível de abstração na modelagem, que serão primeiro projetadas por uma pessoa e então traduzidas em código. No entanto, esta abordagem evidencia a constante necessidade do desenvolvedor para modelagem e criação dos SmCs, pois, toda vez que uma aplicação necessitar de um novo SmC em determinada situação, precisará do desenvolvedor para modelar o novo SmC, gerar o código, compilar, implantar na rede e integrar com a aplicação.

Esta dissertação objetiva automatizar o processo de desenvolvimento de SmC, diminuindo a necessidade de uma pessoa desenvolvedora. Tal abordagem possibilita construir sistemas capazes de modelar os SmCs, gerar seus códigos-fonte, compilá-los, implantá-los e interagir com estes por conta própria em tempo de execução e de forma dinâmica, ou seja, sem o uso de fragmentos de códigos pré-prontos. Desta maneira, a principal contribuição desta dissertação é a introdução de um novo paradigma, onde sistemas podem atuar como desenvolvedores de SmCs, o que é alcançado a partir da proposta do Gifflar, um *framework* que oferece as ferramentas necessárias para permitir sistemas gerarem códigos de SmCs em tempo de execução. O Gifflar é uma forte contribuição para o estado da arte, pois, até onde se sabe, ele é um dos primeiros *frameworks* a introduzir esta abordagem.

O Gifflar oferece um ambiente de desenvolvimento que integra diferentes ferramentas internas e também fornece uma interface de linha de comandos para facilitar o processo de desenvolvimento com o *framework*. A principal ferramenta do Gifflar é a biblioteca Solgen (*Solidity Generator*). Uma biblioteca estruturada em componentes que implementa padrões de projeto para melhor definir a responsabilidade de cada uma dessas partes. A API da Solgen permite ao desenvolvedor modelar os SmCs e configurar todo o processo de automação de criação e a implantação (*deploy*) dos mesmos. Toda a modelagem de contratos pelo Gifflar é realizada através de métodos da própria biblioteca que utiliza a linguagem JSON para gerar um modelo de representação que posteriormente será utilizado para gerar o código de SmC em Solidity. Além de oferecer as ferramentas apresentadas, o Gifflar busca futuramente acoplar outras bibliotecas que geram códigos em outras linguagens de SmC, permitindo assim utilizar o Gifflar para gerar SmCs em outras redes blockchain que não utilizam o Solidity.

O processo de geração de códigos com a biblioteca Solgen é organizado em quatro etapas que definem o fluxo desde o JSON modelado até a implantação do contrato na rede blockchain. A primeira etapa representa a modelagem do contrato utilizando os métodos da biblioteca. Uma vez o contrato modelado em JSON, este JSON é enviado para a etapa de escrita, que é a etapa responsável por interpretar o JSON e gerar o código-fonte do contrato. A partir do código-fonte, a terceira etapa (etapa de compilação) irá compilar o código e gerar duas estruturas importantes que são a ABI (Interface Binária de Aplicação) do contrato e os *bytecodes*. A ABI é a interface utilizada pelas aplicações para se comunicarem com as funções do SmC. Já os *bytecodes* são uma representação em baixo nível do código-fonte que são enviadas para a máquina virtual da rede blockchain. A última etapa é a etapa de implantação, a qual é responsável por implantar o contrato na rede utilizando os *bytecodes* e retornar a instância do contrato a partir da ABI. Este processo oferecido pelo Gifflar contribui de forma inovadora para o estado da arte, permitindo que sistemas possam gerar SmCs de forma dinâmica em tempo de execução do sistema através da modelagem com a API da biblioteca Solgen.

Comparado com os trabalhos relacionados, o nível de abstração do modelo de representação de SmC do Gifflar é menor do que o nível de abstração utilizado nos modelos desses trabalhos. Isso quer dizer que não é qualquer ser humano que consegue entender e modelar os SmCs utilizando o Gifflar, somente um grupo específico: desenvolvedores de software. Ainda assim, o Gifflar permite modelar SmCs de uma forma mais simples e abstrata do que o desenvolvimento direto com Solidity, isso porque a biblioteca Solgen, utilizada no processo de modelagem do código do SmC, foi criada na intenção de promover uma API intuitiva para os desenvolvedores, ou seja, atém de possibilitar a automatização da criação de SmCs, o Gifflar como um todo também auxilia o desenvolvedor na modelagem das instruções do contrato.

Além disso, o fato de a modelagem do Gifflar ser voltada para desenvolvedores permite que os próprios trabalhos relacionados se beneficiem do *framework*. Ou seja, os desenvolvedores destes projetos poderiam utilizar o Gifflar dentro do sistema e, a partir da modelagem oferecida pelo *framework*, criar um modelo com um nível de abstração maior. Isto facilitaria o desenvolvimento do sistema, uma vez que a Solgen abstrai a sintaxe da linguagem Solidity. Um exemplo seria utilizar o Gifflar para criar um sistema de geração

de SmCs a partir de um diagrama visual. Neste exemplo, ao usuário modelar o SmC visualmente, internamente o sistema estaria utilizando as funções da Solgen para gerar o modelo JSON. Posteriormente, ao acionar o mecanismo de gerar o código de SmC, o sistema recorre à biblioteca Solgen para gerar o SmC a partir do JSON modelado.

Pelo fato da Solgen ser a principal ferramenta do *framework*, apresenta-se um teste de usabilidade na API da Solgen com desenvolvedores a fim de validar a usabilidade da biblioteca. O teste seguiu a metodologia de Piccioni, Furia e Meyer (2013), que elaboram um estudo empírico para avaliar a usabilidade especificamente de APIs e que pode ser aplicado para vários domínios diferentes. Neste trabalho, eles também fazem uso de algumas metodologias já consolidadas na literatura; uma delas é a noção de dimensões cognitivas (BLACKWELL et al., 2001), que permite detectar características entre expectativas de usuários e o que a API realmente entrega; este foi utilizado no teste da Solgen para obter um retorno dos usuários ao experimentar a biblioteca.

Outra metodologia utilizada foi o protocolo *Thinking aloud* (ERICSSON; SIMON, 1993) onde, durante a experiência de cada desenvolvedor em utilizar a biblioteca, são expressas em voz audível qualquer pensamento ou expectativa sobre os recursos da Solgen. De acordo com as palavras expressas pelos participantes do experimento ou a intenção inicial em como utilizar algum recurso, são definidos *tokens* de usabilidade – uma metodologia dos autores Piccioni, Furia e Meyer em que pode-se classificar as reações dos participantes em tokens, como por exemplo, "*surprise*" ou "*unexpected*".

De modo geral, o teste de usabilidade alcançou resultados muito positivos. Todos os participantes conseguiram concluir a sequência de atividades criada para guiar o teste e relataram que é um projeto muito promissor, ainda que proponha um paradigma diferente de programação. Alguns dos participantes também relataram que acharam a modelagem de contratos no Gifflar tão simples que preferem utilizar o Gifflar na criação de SmCs do que programar direto na linguagem Solidity. Por outro lado, foram encontradas algumas dificuldades dos participantes ao utilizar a Solgen, além disso, foram detectados alguns recursos que não condizem com suas expectativas. Todos estes fatores são discutidos com detalhes nesta dissertação e, ao fim da discussão, são apresentadas as principais alterações a serem realizadas na biblioteca e, consequentemente, no Gifflar.

Esta nova abordagem oferecida pelo Gifflar permite aprimorar as diversas aplicações existentes que utilizam ou pretendem utilizar a tecnologia blockchain. Por isso, além da avaliação de usabilidade, também é realizada uma avaliação mais conceitual objetivando evidenciar o interesse em utilizar o Gifflar em um projeto. Para isso, foi realizada uma revisão simples da literatura em busca de projetos que utilizam a tecnologia blockchain e que poderiam ser aprimorados caso utilizassem o Gifflar internamente. Como resultado desta pesquisa, dentre 100 artigos, foram selecionados quatro que condizem com os critérios de inclusão e exclusão.

Para melhor guiar um projeto em decidir se utiliza ou não o Gifflar, foi criado um fluxograma de decisão (Figura 3.9 na Seção 3.4 do Capítulo 3), e as condições apresentadas neste fluxograma foram utilizadas na revisão como critérios de inclusão. Já como critérios de exclusão, foram desconsiderados *whitepapers*, livros ou slides, artigos duplicados e artigos em idiomas diferentes do inglês. Dos quatro projetos selecionados, dois foram escolhidos para descrever com detalhes como o Gifflar poderia ser aplicado na intenção

de aprimorar estes projetos, são eles: o IoTCocoa (ABIJAUDE et al., 2019) e um voltado para a indústria farmacêutica (CHIACCHIO et al., 2020).

O projeto IoTCocoa (ABIJAUDE et al., 2019) é voltado para a área de Internet das Coisas (do inglês, *IoT (Internet of Things)*), uma tecnologia que tornou possível objetos do dia a dia serem alcançados de qualquer local utilizando aparelhos como um *smarthphone* para controlar e/ou monitorar tais objetos através de dispositivos que utilizam protocolos como *WiFi* e *Bluetooth* (SETHI; SARANGI, 2017). O IoTCocoa visa melhorar o monitoramento da produção de cacau gourmet por meio de dispositivos IoT, uma aplicação web e um middleware, com o desafio de integrar blockchain e SmCs para herdar todas as características de segurança e rastreabilidade oferecidas pela tecnologia. Já o projeto da indústria farmacêutica (CHIACCHIO et al., 2020) utiliza cadeia de suprimentos, a qual oferece um sistema integrado que sincroniza uma série de processos de negócios inter-relacionados a fim de gerenciar o ciclo de vida de produtos e facilitar a comunicação entre as entidades do negócio (MIN, 2015). Tal projeto propõe um sistema de rastreabilidade do ciclo de vida de produtos farmacêuticos utilizando SmCs. Estes dois projetos são descritos de forma mais técnica nos resultados da avaliação evidenciando os aprimoramentos que o Gifflar pode promover para ambos.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Este trabalho de dissertação tem como objetivo geral automatizar o desenvolvimento de contratos inteligentes de forma a permitir que sistemas sejam capazes de modelar, escrever os códigos-fonte, compilar, e implantar os contratos inteligentes em tempo de execução.

1.1.2 Objetivos Específicos

A fim de alcançar os objetivos gerais, foram definidas as metas abaixo:

1. Criar uma estrutura de *framework* definindo os padrões de projeto e técnicas a serem utilizadas, escolha da linguagem de contrato a ser gerada e do tipo de blockchain onde serão implantados os contratos;

2. Propor o Gifflar, um *framework* que facilita o desenvolvimento de SmCs e que principalmente pode ser utilizado por sistemas para gerar SmCs em tempo de execução;

3. Verificar os riscos de segurança em gerar contratos em tempo de execução e propor uma forma de aplicar esta segurança no Gifflar;

4. Desenvolver o Gifflar com as principais propriedades de linguagem de SmC, como criação de variáveis, atribuição e criação de funções;

5. Realizar uma avaliação de usabilidade da API da biblioteca Solgen a partir de uma estratégia definida;

6. Realizar uma avaliação de aplicabilidade do Gifflar em projetos definidos;

7. Destacar quais são as principais modificações futuras a serem realizadas no Gifflar.

1.1.3 Publicações Obtidas

Até o momento, os resultados parciais obtidos com esta dissertação originaram os seguintes principais artigos científicos:

- *"Gifflar: A Framework to Generate Smart Contracts On the Fly"*: Este artigo introduz o Gifflar ainda no estágio de biblioteca explicando todos os componentes, modelos e padrões utilizados na sua criação. Publicado em *CASCON '21: Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering* (SANTIAGO; ABIJAUDE; GREVE, 2021b).

- "*A Framework to Generate Smart Contracts On the Fly*": Este artigo complementa o anterior com um estudo de caso básico utilizando a versão de protótipo do Gifflar como biblioteca a fim de gerar SmCs para sensores IoT, evidenciando mais em detalhes o funcionamento e diferencial da proposta. Publicado em *SBES '21: Proceedings of the XXXV Brazilian Symposium on Software Engineering* (SANTIAGO; ABIJAUDE; GREVE, 2021a).

Para além desses principais artigos, as seguintes publicações correlacionadas ao trabalho de dissertação e à grande área da blockchain foram obtidas:

- "Blockchain applied to academic environments as a way to ensure educational process quality control" (SANTIAGO et al., 2020).

- "IoTCocoa - an IoT platform to assist gourmet cocoa production" (ABIJAUDE et al., 2019).

1.2 ORGANIZAÇÃO

A dissertação está organizada como segue: O Capítulo 2 faz uma revisão mais aprofundada sobre *blockchain*, programação orientada a objetos, bibliotecas, *frameworks*, princípios SOLID, as linguagens *JavaScript*, *TypeScript* e JSON, geração automática de código, finalizando com os trabalhos relacionados. O Capítulo 3 explica toda a estrutura do *framework* destacando todos os seus componentes, desafios encontrados, o código, versão e propriedades suportadas e o fluxograma que determina quando é possível utilizar o Gifflar em outros projetos. O Capítulo 4 apresenta toda a estratégia de avaliação de usabilidade da biblioteca do Gifflar, evidenciando as questões de pesquisa, as perguntas realizadas aos participantes, os tokens de usabilidade para classificação das reações dos desenvolvedores, as *tasks* criadas para guiar o teste, o ambiente de desenvolvimento utilizado, quem foram os participantes do teste e então é detalhado todo o protocolo seguido com cada participante. Neste mesmo capítulo é apresentada a estratégia de avaliação de aplicabilidade do Gifflar em outros projetos, iniciando com a definição das questões de

pesquisa desta avaliação, seguindo da condução da pesquisa dos projetos na literatura, então a definição de como foi realizada a triagem dos artigos obtidos e por fim, é listada a sequência de etapas do estudo. O Capítulo 5 expõe e discute todos os resultados obtidos tanto da avaliação do teste de usabilidade quanto da avaliação de aplicabilidade do Gifflar em outros projetos. Por fim, o Capítulo 6 apresenta as conclusões finais e os trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS

Este Capítulo apresenta os conhecimentos necessários em relação à tecnologia blockchain; programação orientada a objetos (OOP - do inglês *Object-Oriented Programming*); bibliotecas; *frameworks*; princípios SOLID; linguagens *JavaScript*, *TypeScript* e JSON; geração automática de código; além de mencionar os estudos relacionados, evidenciando o diferencial deste trabalho de dissertação. Sobre a blockchain, é explicado todo o processo de assinatura com chaves pública e privada até o processo de mineração e consenso, discussão sobre a implantação e execução dos SmCs e desafios em aberto. Posteriormente é explicado sobre a OOP e quais as vantagens em utilizar este paradigma de programação, seguindo com os conceitos sobre bibliotecas, *frameworks* e padrões de projeto. Em seguida são esclarecidos cada um dos cinco princípios da Engenharia de Software que formam a sigla SOLID. Sequencialmente são introduzidas as linguagens *JavaScript* e *TypeScript* destacando suas diferenças e também explicando o JSON trazido pela primeira linguagem. Posteriormente, é descrito o contexto onde esta dissertação se encontra, introduzindo sobre a geração automática de códigos. E finalmente, nos trabalhos relacionados são referenciados alguns autores que também propuseram diferentes *frameworks* e ferramentas de geração automática de códigos para SmCs.

2.1 BLOCKCHAIN

A blockchain (cadeia de blocos) é uma tecnologia que oferece uma rede Ponto a Ponto (P2P – do inglês *Peer-to-Peer*) de registros para transacionar informações de forma distribuída e segura, dispensando o terceiro elemento de confiança, como bancos, corretoras, entre outros. Essa tecnologia foi inicialmente proposta por Satoshi Nakamoto com a Bitcoin em 2008 (NAKAMOTO et al., 2008). A ideia principal da Bitcoin, é fazer com que duas ou mais partes realizem negociações sem necessitar de empresas intermediárias e sem precisar informar suas credenciais, proporcionando que as partes não confiáveis mutuamente, façam transações entre elas de forma segura (KARAME; ANDROULAKI,

2016). Tais transações permitem a transferência de valores a partir da moeda virtual chamada *bitcoin*. A Bitcoin implementa uma máquina de estados simplificada, permitindo a criação de *scripts* não muito complexos. Foi decidido desta forma para evitar ataques. Se sua linguagem de *script* fosse Turing-completa, um atacante poderia criar um algoritmo que nunca termina, causando um *loop* infinito, o que poderia derrubar a rede. Embora simplificada, a linguagem é suficientemente poderosa para o desenvolvimento de várias aplicações, as chamadas Aplicações Descentralizadas (DApps – do inglês *Decentralized Applications*) (FRANCO, 2014).

As plataformas baseadas em blockchain possuem algoritmos para coordenar os nós da rede, de forma que estes consigam chegar a um consenso sobre as informações que estão sendo adicionadas na base de dados distribuída, chamada de *ledger*. Estas características quando combinadas, mostram que a blockchain serve como uma ferramenta poderosa para facilitar atividades econômicas e sociais (FILIPPI, 2018). Para que esta tecnologia funcione da forma mais confiável possível, existem vários elementos e conceitos que juntos buscam alcançar este objetivo.

A blockchain, embora proposta inicialmente por Satoshi Nakamoto com a Bitcoin, apresenta variações em alguns aspectos, considerando a grande quantidade de diferentes implementações por diferentes empresas ou pesquisadores, como por exemplo, a plataforma Ethereum, que trouxe o conceito dos SmCs que permitem a execução de *scripts* mais complexos e que serão melhor abordados no final desta seção. A explicação dos componentes desta tecnologia, nesta dissertação, é baseada na Bitcoin, uma vez que foi a plataforma pioneira na implementação da blockchain e as outras são basicamente extensões desta. Já os conceitos de SmC serão apresentados considerando a plataforma Ethereum. As próximas subseções descrevem o mecanismo de chaves públicas e privadas; cadeia de blocos, consenso e mineração e SmCs, todos fundamentais para a compreensão do funcionamento básico desta tecnologia.

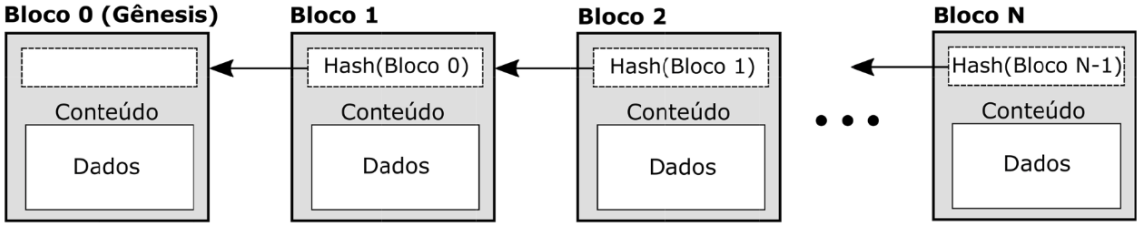
2.1.1 Chaves Pública e Privada

A Bitcoin utiliza criptografia assimétrica, que consiste na utilização de duas chaves (pública e privada) para encriptar mensagens. Nesta técnica, a mensagem quando encriptada com uma chave pública, somente pode ser desencriptada pela chave privada correspondente, o mesmo vale para a operação inversa. Porém, na Bitcoin, esta criptografia não é utilizada para encriptar as mensagens, mas para assinaturas digitais. Cada nó da rede P2P recebe estas duas chaves referentes à uma conta específica. A chave pública é utilizada como endereço único, para recebimento de fundos. A chave privada é utilizada para assinar as transações na blockchain a fim de consumir os fundos e, portanto, não deve ser revelada para ninguém além do seu proprietário (FERNÁNDEZ-CARAMÉS; FRAGA-LAMAS, 2018). Estas assinaturas garantem que o emissor da transação é realmente aquele que possui a chave privada e que cada transação tenha uma assinatura diferente a partir desta.

2.1.2 Cadeia de Blocos

Para garantir a segurança, a blockchain possui uma base de dados chamada de *ledger*, ou livro razão, replicado por vários nós, onde, uma vez adicionada uma informação, esta nunca mais poderá ser alterada (SINGHAL; DHAMEJA; PANDA, 2018). O *ledger* é organizado em uma cadeia de blocos e, sendo uma estrutura imutável, todos os blocos antigos ainda estão na blockchain e nunca serão removidos. O primeiro bloco da cadeia Bitcoin é chamado de bloco *genesis* e foi adicionado por Satoshi Nakamoto. O último bloco adicionado na blockchain é chamado de *blockchain head* (cabeça da cadeia de blocos). A Figura 2.1 ilustra a estrutura da cadeia de blocos. Como cada bloco aponta para o *hash* do bloco anterior, a cadeia de blocos torna-se inviolável, sendo quase impossível adulterar o valor de uma transação em toda a blockchain (FRANCO, 2014).

Figura 2.1 Demonstração da estrutura da cadeia de blocos



Fonte: Rebello et al. (2019)

2.1.3 Consenso e Mineração

Uma vez que não existe terceira parte confiável, para garantir a autenticidade das transações, a blockchain precisou da implementação de um mecanismo para assegurar que a transação a ser inserida no *ledger* é uma transação válida. Este mecanismo é chamado de consenso. O consenso é um protocolo de tolerância a falhas executado por cada nó da blockchain para, assim, assegurar que todos entrem em um acordo com relação às transações adicionadas à cadeia de blocos (CACHIN; VUKOLIĆ, 2017). O processo de mineração depende do algoritmo de consenso adotado. O algoritmo de consenso da Bitcoin é a Prova de Trabalho (PoW - do ingês *Proof of Work*) e os nós da rede que participam deste processo são chamados de mineradores. Estes nós são computadores com grande poder computacional para resolver um desafio que envolve essencialmente a força bruta (GREVE et al., 2018). Cada minerador irá tentar resolver este problema computacional e, o primeiro que resolver, é o que receberá a permissão para adicionar o seu bloco no *ledger* com algumas transações realizadas até o momento, recebendo uma remuneração em troca.

O consenso PoW, porém, tem suas limitações. Seu processo de mineração, acaba exigindo um alto gasto de energia e, como a validação de cada transação precisa ser feita por uma grande quantidade de nós da rede, o intervalo de tempo para que o próximo bloco seja adicionado no *ledger* é alto, chegando a 10 minutos na Bitcoin (VUKOLIĆ, 2015).

2.1.4 Contrato Inteligente

O contrato inteligente é um conceito proposto inicialmente por Szabo (1997) e implementado por Buterin et al. (2014) com a criação da plataforma Ethereum. A ideia de Szabo, era traduzir cláusulas contratuais em código, e de alguma forma controlar tais informações com a mínima intervenção de intermediários em transações entre partes (CHRISTIDIS; DEVETSIKIOTIS, 2016). Nesta época, ainda não existia uma tecnologia para desenvolver este conceito, com o surgimento da blockchain, isto tornou-se possível.

Na Ethereum, os SmCs são escritos na linguagem Solidity, uma linguagem Turing-completa, e são compilados para uma linguagem de baixo nível específica, para que sejam executados na EVM, a qual está configurada em cada nó da rede. Na EVM, cada instrução tem um custo, por isso, a implantação do SmC na rede gera um custo em *ethers* (moeda virtual da Ethereum) que depende dos tipos e quantidade de instruções utilizadas. Da mesma forma, depois de implantado na rede, toda operação de escrita no SmC deve ser paga em *ethers*, no entanto, a operação de leitura é livre. Toda vez que um SmC é implantado na rede, a fim de preservar a característica da imutabilidade da blockchain, este nunca mais poderá ser alterado.

Os SmCs residem dentro da rede Ethereum, e assim como os endereços públicos e únicos que cada pessoa na rede possui, cada SmC tem seu próprio endereço, carteira e saldo. Todo SmC é capaz de enviar e receber transações e, ao receber uma nova transação, pode ser ativado ou desativado. Todo SmC, uma vez implantado na rede, qualquer integrante desta pode acessar suas funções, todavia, nestas funções podem ser inseridas características de segurança, prevenindo acessos não autorizados (SINGHAL; DHAMEJA; PANDA, 2018).

Com relação à implementação, os SmCs possibilitaram o aperfeiçoamento de DApps, de forma que mais aplicações e sistemas baseados em blockchain surgiram para diversas áreas diferentes, como agricultura, a exemplo de Prause e Boevsky (2019), energia, explorado por Li et al. (2019), governo eletrônico, estudado por Santiago et al. (2020) e IoT, tal como em Abijaude et al. (2019). Para o desenvolvimento destas aplicações, existem diversas ferramentas que foram sendo propostas desde o surgimento da plataforma Ethereum. Muitas destas ferramentas e *frameworks* estão listados no site da Ethereum[1]. Mas os principais aparatos oferecidos oficialmente são o web3[2], que é um pacote para implantação e interação com os SmCs disponível para várias linguagens, o solc[3], que é o compilador de código Solidity e o editor de códigos Remix[4], para a escrita, compilação e implantação de contratos. Baseando-se em Mukhopadhyay (2018), o processo de criação de um contrato segue o seguinte fluxo:

- Primeiro é desenvolvido o código do contrato em Solidity;
- Depois utiliza-se o solc com a correta versão para compilar o código e gerar duas estruturas, a Interface Binária de Aplicação (ABI - do inglês *Application Binary*

[1]https://ethereum.org/pt-br/developers/
[2]https://web3js.readthedocs.io/
[3]https://docs.soliditylang.org/
[4]https://remix.ethereum.org/

Interface) e os *bytecodes*. Os *bytecodes* são as instruções em baixo nível do código que a EVM entende, a ABI contém os protótipos das funções criadas para utilizar como interface de interação com o SmC;

- Utilizando o *Infura*[5] (ou outro serviço semelhante), um nó da rede Ethereum que oferece um serviço de API para comunicação com a rede, pode-se obter um *endpoint* que será utilizado para comunicação com a blockchain;

- Podendo utilizar o Remix ou web3, a partir dos *bytecodes* e do *endpoint* do *Infura*, é possível implantar o contrato na rede Ethereum, sendo a rede principal ou uma rede de teste (criada para testar o funcionamento dos contratos sem gastar *ethers* reais);

- Depois de implantado na rede, é possível utilizar a ABI para realizar a comunicação com este contato e acionar funções anteriormente programadas.

2.1.5 Vulnerabilidade de Contratos Inteligentes

Embora os SmCs tenham trazido vários benefícios e contribuído para o crescimento e maior adoção da tecnologia em diferentes áreas, estes trouxeram de forma associada, alguns desafios a serem solucionados. O principal dos desafios, que ainda encontra-se em aberto, são as vulnerabilidades encontradas em diversos contratos na Ethereum. Um ataque muito conhecido, que explorou uma certa vulnerabilidade existente em um contrato implantado, foi o "ataque DAO", no qual o atacante conseguiu roubar em torno de 60 milhões de dólares ao encontrar uma falha em uma função do contrato e fazer esta ser executada recursivamente como um *loop*, fazendo com que o contrato enviasse *ethers* sem limites para seu endereço blockchain (PRAITHEESHAN et al., 2019). Outras falhas aconteceram em diversos contratos explorando diferentes vulnerabilidades, como geração de números randomizados utilizando mecanismos que são controlados por mineradores ou armazenando dados secretos em variáveis privadas, sendo que tudo na blockchain é público (ARGAÑARAZ et al., 2020). Para melhor entender o que internamente pode gerar tais falhas, algumas destas vulnerabilidades citadas foram detalhadas abaixo.

- Reentrância: Esta foi a vulnerabilidade explorada no citado ataque ao contrato inteligente DAO e é, portanto, uma das mais importantes e conhecidas. Tal vulnerabilidade pode acontecer quando um código Solidity possibilita o usuário de armazenar e sacar valores em *ether* do contrato de forma a obedecer uma ordem de instruções suscetível a falha. O Código 2.1 é um exemplo de código com falha de reentrância, onde na linha 4 é criado um mapeamento de valores em *ether* que podem ser sacados por endereços (podendo ser um usuário ou um outro SmC) utilizando a função da linha 6. O problema é que, antes de zerar o saldo do mapeamento *shares* do endereço que executou a função, o código primeiro executa a transferência de *ethers*. Isto permite que, por exemplo, outro contrato que esteja chamando esta função, possa chamá-la novamente antes mesmo de finalizar as instruções, e, como

[5] https://infura.io/

a linha 8 não seria executada, o contrato iria erroneamente informar que aquele endereço possui o mesmo saldo que anteriormente, possibilitando sacar o mesmo saldo várias vezes, como aconteceu no "ataque DAO" (LIU et al., 2018). Uma resolução para este problema, seria atualizar o saldo daquele endereço antes de realizar a transferência em si, como mostra o Código 2.2.

Código 2.1 Exemplo de trecho de código Solidity com falha de reentrância

```
1  // THIS CONTRACT CONTAINS A BUG - DO NOT USE
2  contract Fund {
3      /// @dev Mapping of ether shares of the contract.
4      mapping(address => uint) shares;
5      /// Withdraw your share.
6      function withdraw() public {
7          if (payable(msg.sender).send(shares[msg.sender]))
8              shares[msg.sender] = 0;
9      }
10 }
11
```

Fonte: Documentação do Solidity[6]

Código 2.2 Exemplo de trecho de código Solidity sem falha de reentrância

```
1  contract Fund {
2      /// @dev Mapping of ether shares of the contract.
3      mapping(address => uint) shares;
4      /// Withdraw your share.
5      function withdraw() public {
6          uint share = shares[msg.sender];
7          shares[msg.sender] = 0;
8          payable(msg.sender).transfer(share);
9      }
10 }
11
```

Fonte: Documentação do Solidity[6]

- Envio não verificado: Esta falha ocorre quando o desenvolvedor utiliza um método de envio de *ether* e não verifica se esta operação foi corretamente finalizada. No Solidity existem duas formas para que o envio seja realizado, utilizando a função *"transfer"* ou a função *"send"*. A primeira, já faz a verificação automática da correta execução do envio, porém, utilizando a segunda, é preciso fazer este processo manualmente, caso esqueça de fazer, isso pode permitir que um atacante possa executar um código malicioso e obter todo o saldo (DIKA, 2017).

- Dependência do timestamp: O Solidity permite acessar uma variável que retorna o *timestamp* do bloco em que aquela transação está. Por permitir este acesso,

[6]https://docs.soliditylang.org/en/latest/security-considerations.html

o desenvolvedor pode utilizá-la para vários fins, como dentro de uma operação de envio de *ethers* ou geração de números aleatórios. Porém, é exatamente em operações como estas que torna o código vulnerável. O *timestamp* do bloco é definido pelo minerador logo no momento em que finaliza a mineração, portanto, variáveis como estas, são completamente controladas por este nó. Por isso, não é recomendado utilizá-las para fins como este, uma vez que pode ser manipulada intencionalmente a favor do nó que minerou aquele bloco (JIANG; LIU; CHAN, 2018).

- DelegateCall Perigosa: O *delegateCall* é uma função que pode ser utilizada dentro do Solidity para executar uma função de um contrato recebendo os dados da execução desta função via parâmetro. Este parâmetro pode ser enviado através da variável da linguagem chamada *"msg.data"*. No momento em que um contrato implementa uma *delecateCall* o cuidado que precisa ter é que, neste momento, uma pessoa pode utilizar esta operação para executar qualquer função que seja pública não só do contrato, mas também das bibliotecas que este utiliza. Portanto, se existir alguma função pública que não poderia ser executada por qualquer endereço, um atacante pode aproveitar-se da abertura da *delegateCall* e acionar uma determinada função que possa beneficiá-lo (JIANG; LIU; CHAN, 2018).

- Profundidade da pilha de chamada: A profundidade da pilha de chamada ou *call-stack depth*, é o máximo de vezes que uma função pode ser chamada iterativamente. Cada vez que uma função é chamada, a *call-stack depth* é acrescida de um. Na EVM, o limite da *call-stack* é de 1024 chamadas, ou seja, se esse limite for excedido, a EVM irá emitir um erro. Sabendo desta informação, um atacante pode aproveitar para beneficiar-se. Um possível ataque, é criar um contrato que chama ele mesmo 1023 vezes antes de chamar a função de um outro contrato. O que acontece neste caso é que a EVM irá emitir um erro, e, dependendo de como o contrato lida com esse tipo de erro, este pode tornar-se vulnerável e o atacante aproveitar-se disso (PRAITHEESHAN et al., 2019; LUU et al., 2016).

Como na blockchain, uma vez implantado o contrato, é inviável realizar alterações, é necessário que os próprios desenvolvedores certifiquem que aquele contrato não apresenta nenhuma falha de segurança. Como listado acima, vários dos tipos de vulnerabilidades são falhas geradas pelos próprios desenvolvedores, não são erros da linguagem e muito menos da blockchain em si, neste caso, a Ethereum.

Hoje, existem diversas ferramentas para ajudar na verificação de possíveis falhas nos contratos antes de serem implantados na rede, ferramentas que permitem realizar desde testes estáticos e formais, para verificação da escrita do código, até testes de execução simbólica, para avaliar o correto funcionamento das funções do contrato (SINGH et al., 2020). Entretanto, por mais que existam tais ferramentas, o Solidity, por ser uma linguagem Turing-completa, acaba limitando a qualidade das análises, e é possível que linguagens não Turing-completas possam diminuir tais vulnerabilidades e permitir realizar análises mais completas (ATZEI; BARTOLETTI; CIMOLI, 2017).

2.2 PROGRAMAÇÃO ORIENTADA A OBJETOS

A OOP é um paradigma de programação que adota o conceito de "objetos" para representar artefatos da programação como objetos da vida real. Por ser uma abstração do mundo real, torna o código mais claro para o desenvolvedor, facilita modificações de código, reduz riscos de programação e encoraja o reuso de componentes (BOOCH et al., 2008). As principais características da OOP são: (1) objetos, que são representações de objetos reais, pessoas ou animais expressos em uma linguagem de programação; (2) classes, que funcionam como grupos de objetos que compartilham os mesmos atributos; (3) encapsulamento, o qual é a capacidade de esconder informações internas que não são necessárias para o conhecimento do usuário; (4) agregação ou composição, que é a habilidade de combinar vários objetos dentro e um objeto só a fim de dividir um problema; (5) herança, é um a forma de reuso de código onde uma classe pode herdar características de outras classes; e (6) polimorfismo, onde objetos, por exemplo, de uma mesma classe sobrescrevem tipos de ações em comum de modo que uma mesma ação pode ser expressa de forma diferente para cada objeto (STEFANOV, 2008). A OOP é hoje adotada em diversos tipos diferentes de aplicações, dentre eles estão as bibliotecas e *frameworks*.

2.3 BIBLIOTECAS

Ao programar um software, muitas vezes os desenvolvedores precisam lidar com uma mesma sequência de operações ou funcionalidades, as quais podem também ser reaproveitadas para outros softwares. A biblioteca é uma forma de encapsular funcionalidades que serão reutilizadas variadas vezes. Ou seja, é uma coleção de códigos pré-escritos que, através de uma API exportada, facilitam a resolução de algumas tarefas e também deixam o código mais limpo, de forma que uma sequência de operações podem ser resumidas em uma única função ou método da biblioteca (MCFARLAND, 2011). São alguns exemplos de bibliotecas na linguagem JavaScript: JQuery[7], React.js[8] e Web3.js[9].

2.4 FRAMEWORKS

No site da Ethereum, um conjunto de *frameworks* para a construção de SmCs e DApps está listado. O Truffle[10], Embark[11] e Waffle[12] são alguns deles. Mas, além desses, é possível encontrar muitas outras propostas na literatura, como FSolidM (MAVRIDOU; LASZKA, 2018), um *framework* para desenvolvimento de contratos inteligentes, e Vandal (BRENT et al., 2018), um *framework* para testes de contratos inteligentes. Como este trabalho de dissertação é uma nova contribuição para essa comunidade com um novo *framework*, é importante construir um melhor entendimento sobre as características de um *framework*.

[7] https://jquery.com/
[8] https://reactjs.org/
[9] https://web3js.readthedocs.io/
[10] https://www.trufflesuite.com/
[11] https://framework.embarklabs.io/
[12] https://getwaffle.io/

Um *framework* pode ser definido como um conjunto de classes abstratas ou componentes que podem ser reutilizados para resolver problemas similares fazendo uso de designs diferentes para melhor encapsular recursos (JOHNSON; FOOTE, 1988). Ou seja, realizando uma comparação mais compreensível, o *framework* pode ser comparado a um motor sem energia para ser acionado, e o seu gerador de energia é a aplicação que utiliza os artefatos abstratos do *framework* para um domínio específico (MARKIEWICZ; LUCENA, 2001).

Diferente das bibliotecas, os *frameworks* são mais genéricos a ponto de, não só oferecer operações encapsuladas em funções, como também trazer algumas facilidades para o processo de desenvolvimento do programador, como organização de diretórios do projeto e ferramentas de desenvolvimento que podem ser acessadas via linha de comando do terminal. Ou seja, os *frameworks* englobam as bibliotecas e outros artefatos para oferecer ao programador um ambiente de desenvolvimento mais completo conforme um determinado domínio.

De acordo com o Fayad e Schmidt (1997), um *framework* pode ser classificado com base na técnica utilizada: *white-box* ou *black-box*. Eles explicam que os *frameworks white-box* dependem muito da OOP para estruturar as classes adotando herança e métodos sobrescritos para reutilizar o código. Além disso, o uso destes *frameworks* requer um certo entendimento de alguns relacionamentos internos. Os *frameworks black-box* também podem utilizar OOP, mas não dependem muito de artefatos da OOP para exportar seus elementos, eles utilizam componentes definidos para realizar uma tarefa específica, e estes componentes podem ser utilizados por uma aplicação via composição de objeto. Estes componentes também encapsulam recursos internos, oferecendo interfaces externas para serem usadas sem a necessidade de entender como o componente funciona. Como Johnson (1988) explica, *black-box* é ideal para a evolução do sistema, pois, além da relação dos componentes tornar-se mais reutilizável, a organização do *framework* também fica mais compreensível, por isso, esta foi a técnica escolhida para o desenvolvimento do *framework* proposto.

Para melhor organizar a maneira como os componentes do *framework* são projetados, é importante fazer o uso de padrões de projeto. Os padrões de projeto permitem padronizar um vocabulário na definição de soluções e criar código otimizado, considerando as necessidades do problema (OSMANI, 2012). Três dos padrões de projeto utilizados nesta dissertação são o *Factory*, *Module* e *Builder*. O padrão *Factory* é usado para quebrar a dependência de uma classe/componente A para criar uma instância de outra classe/-componente B, neste caso, é criada uma fábrica C que sabe tudo o que é necessário para instanciar B e retornar a instância para A (DIAZ; HARMES, 2008). O padrão *Module* torna possível encapsular elementos públicos e privados de uma classe. O padrão *Builder* é usado para representar a construção de um objeto de diferentes maneiras até sua configuração final (GAMMA et al., 1995).

2.5 PRINCÍPIOS SOLID

Os princípios SOLID trazem um conjunto de orientações que podem ser seguidas para que um software possa ter uma ótima escalabilidade, padronização, manutenibilidade,

dentre outras características. A sigla é uma junção de cinco grandes princípios que surgiram com o intuito de guiar os desenvolvedores a construírem sistemas que vão além de somente cumprir com o objetivo para o qual foram projetados (SINGH; HASSAN, 2015). As linguagens hoje oferecem diversos recursos que facilitam no desenvolvimento de um sistema, como as classes, heranças e interfaces, mas nem sempre a forma como um sistema foi implementado é ideal. Um software pode funcionar corretamente, mas se suas classes ou componentes não estão devidamente estruturados e organizados, uma futura manutenção neste código pode tornar-se uma dor de cabeça (JOSHI, 2016). Para explicar mais detalhadamente o propósito destes princípios, todos os cinco foram descritos a seguir.

- P1 - Princípio da responsabilidade única: Este princípio define que "uma classe deve ter apenas um motivo para mudar", ou seja, uma classe deve ter uma única responsabilidade. Isto porque, quando uma classe fere este princípio, uma determinada alteração no código do sistema pode ser um grande problema, uma vez que esta classe estaria realizando várias funcionalidades diferentes, e isto dificultaria o entendimento e modificação da mesma (SINGH; HASSAN, 2015).

- P2 - Princípio do aberto e fechado: Estabelece que uma classe ou método deve estar aberta para extensão e fechada para modificação. Assim, para seguir este princípio, uma classe deve ser desenvolvida de forma que dificilmente seja alterada, mas que ainda assim, ofereça uma possibilidade de customização. Pode parecer um paradoxo, mas, se a pessoa que projetou aquela classe prever as possíveis extensões, o seu desenvolvimento muito provavelmente estará de acordo com o princípio (VALENTE, 2020).

- P3 - Princípio de substituição de Liskov: Explicita algumas regras de redefinição de métodos de classes mãe em classes filhas. A violação deste princípio surge no momento em que uma classe filha acaba fugindo das regras de sobrescrita de um método da classe a qual esta está herdando. Ao sobrescrever este método de forma não condizente com o princípio, no momento em que for necessário substituir a classe filha pela classe mãe, a alteração no comportamento do método será uma surpresa, e poderá causar uma confusão no correto funcionamento do código. De fato, em algumas situações pode ser necessário sobrescrever um método, e é possível sim fazer a sobrescrita sem violar este princípio, basta seguir as regras estabelecidas pelo mesmo.

- P4 -Princípio da segregação de interface: Assim como o primeiro princípio, este define que as interfaces devem ter responsabilidades únicas. Uma interface que contém protótipos de métodos que representam funcionalidades muito divergentes, pode acontecer de precisar implementar métodos que não são necessários dentro de classes que implementam esta interface. O interessante então é segregar estas interfaces em interfaces com funcionalidades mais específicas, tendo cada uma métodos direcionados para funcionalidades semelhantes, e que com certeza serão necessários nas classes que irão implementar estas interfaces (JOSHI, 2016).

- P5 - Princípio da inversão de dependência: Este último princípio recomenda não depender das classes concretas, mas sim de abstrações, ou seja, interfaces. As interfaces são mais estáveis do que o tipo de uma classe. Ao depender da interface, realizando alterações na classe como o seu próprio nome, não irá influenciar em quase nada no resto do código, pois as dependências são todas de abstrações das classes e não da classe em si (JOSHI, 2016).

2.6 JAVASCRIPT, JSON E TYPESCRIPT

JavaScript é, a princípio, uma linguagem feita especialmente para a Web. É a linguagem que sempre fez parte da tríade de tecnologias mais utilizadas no desenvolvimento Web: o HTML, a linguagem de marcação utilizada para definir a estruturação das páginas; o CSS, que define a estilização desta estrutura; e o *JavaScript* que é utilizada para determinar como esta estrutura deve se comportar. É importante destacar que esta é uma linguagem onde os tipos de dados como *string*, números inteiros, *boolean*, dentre outros, são omitidos do código fonte, e o desenvolvedor, portanto, não precisa definir de forma explícita o tipo de um objeto ao utilizá-lo no código. Além disso, permite a criação de classes, podendo usufruir também dos artefatos de orientação a objetos como herança, polimorfismo, etc. Por mais que tenha surgido como uma linguagem para criação de *scripts* que determinam o comportamento de componentes da Web, acabou tornando-se uma linguagem bem robusta e utilizada no desenvolvimento de *softwares* de grande porte (FLANAGAN, 2004).

Uma das ferramentas que fez esta linguagem ser ainda mais utilizada, foi o Node.js, um ambiente que possibilita executar *JavaScript* do lado servidor. Ou seja, com esta ferramenta, é possível executar uma aplicação *JavaScript* em uma máquina sem a necessidade de um navegador. Assim, tornou-se possível utilizar *JavaScript* não só para definir comportamentos de componentes HTML através de *scripts*, mas também para criação de serviços em servidor, aplicações *desktop*, jogos e até mesmo *frameworks* (TEIXEIRA, 2012).

Outro avanço trazido com a linguagem *JavaScript*, foi o JSON. O JSON é uma linguagem utilizada por aplicações Web como formato de transferência de dados, que acabou facilitando este processo para as aplicações que anteriormente utilizavam Linguagem Extensível de Marcação (XML - do inglês *Extensible Markup Language*). O processo necessário para transferir dados via XML é um pouco exaustivo, por precisar de algumas transformações dos dados tanto do lado servidor quando do lado cliente pelo navegador. Já os dados enviados via JSON, são facilmente transformados, pois além de ter *JavaScript* do lado cliente, com o Node.js já é possível utilizar *JavaScript* do lado servidor. Como o conteúdo de um JSON recebido vem no formato de *string*, é necessário utilizar um artifício que transforme esse JSON para poder usá-lo dentro da linguagem como objeto. Uma forma antiga de realizar este processo no *JavaScript* é utilizando uma função (*eval*) que executa de forma direta o código que aquela *string* representa. Porém, utilizar este processo para transformar o JSON em objeto *JavaScript* torna o sistema inseguro, uma vez que o *eval* permite executar código *JavaScript* em geral e, consequentemente, permite injetar código malicioso que seria executado direto pelo interpretador. A forma

mais segura hoje é utilizar o *parser* (analisador) nativo da linguagem através do método *JSON.parse*, pois, neste caso, o método somente transforma o JSON em objeto *JavaScript* e não permite qualquer outro tipo de código diferente do formato que o JSON aceita (SRIPARASA, 2013). O JSON é estruturado em um modelo de chave-valor, onde cada chave é a identificação de um certo valor configurado e, cada valor, podendo ser de algum tipo de dado padrão, ou até mesmo um objeto JSON, como demonstrado no Código 2.3, no qual consta uma chave *data:* que tem como valor um objeto JSON que contém duas chaves *message:* e *date:* cujos valores são respectivamente um texto como uma *string* e outra *string* representando uma data.

Código 2.3 Exemplo de dado estruturado como JSON

```
{
  "data": {
    "message": "Hello World!",
    "date": "2021-05-20T23:40:27.552Z",
  },
};

```

Embora o JSON tenha surgido do *JavaScript*, é importante evidenciar que um dado estruturado com o JSON não é exatamente do mesmo formato do que quando estruturado como um objeto *JavaScript*, por isso a existência do *parser*. Existem certas diferenças entre estes formatos que são importantes para o correto entendimento desta dissertação. O JSON suporta somente os seguintes tipos de dados: *strings*, números, *booleans* (verdadeiro ou falso), *arrays* (listas), valor nulo e um outro objeto JSON. Já o objeto nativo do *JavaScript* suporta estes e mais outros tipos de dados como o tipo Date da própria linguagem, funções ou até tipos criados pelo próprio desenvolvedor (SRIPARASA, 2013). Outra diferença é que as chaves do JSON são todas definidas com aspas duplas (como "date"), já no objeto *JavaScript* podem ser definidas sem nenhuma aspas. O Código 2.4 mostra o exemplo do Código 2.3 representado como objeto *JavaScript*, adicionando também uma função como outra chave somente para ilustrar que no momento da transformação de objeto *JavaScript* para JSON, as funções são removidas, uma vez que no JSON as funções não podem ser usadas como tipos de dados, e o objeto do tipo Date é transformado automaticamente para string.

Código 2.4 Exemplo de dado estruturado como objeto *JavaScript*

```
// Estrutura anterior em objeto JavaScript
const javascriptObject = {
  data: {
    message: "Hello World!",
    date: new Date(),
    foo: function (){return "bar"},
  },
};

```

```
10    // Imprimindo objeto no console
11    console.log(javascriptObject);
12
13    // Retorna:
14    // { message: 'Hello World!', date: 2021-05-21T00:18:50.425Z,
15    // foo: [Function: foo] }
16
17    // Transformando objeto para JSON em forma de string
18    console.log(JSON.stringify(data));
19
20    // Retorna:
21    // {"message":"Hello World!", "date":"2021-05-21T00:18:50.425Z"}
22
```

A única questão do *JavaScript* é o fato de ser uma linguagem de tipagem implícita. Esse fato pode dificultar o processo de desenvolvimento uma vez que o desenvolvedor muitas vezes não sabe que tipo de chaves aquele objeto possui e também qual o tipo do valor daquela chave, e isso ocorre por ele não ter informações sobre o tipo daquele objeto (NANCE, 2014). O Código 2.5 mostra um exemplo de uma função no *JavaScript* que recebe como parâmetro um objeto chamado *bar* e cria internamente um objeto chamado *internalObject*. Sabe-se de todas as chaves de *internalObject* pois este foi criado dentro da própria função e, por isso, torna-se fácil saber que a chave *data:* de *internalObject* retorna um valor que é uma *string*. Mas, considerando o objeto *bar* recebido como parâmetro, percebe-se uma dificuldade maior em saber que tipo de chave existe ali dentro e ainda que tipo de valor esta chave retorna. Uma forma de conhecer estas informações seria analisar qual a parte do código que realiza a chamada desta função e analisar qual o tipo de parâmetro que este trecho de código envia para ela. Outra forma seria imprimir a variável no console para saber o resultado que foi retornado para então prosseguir com o desenvolvimento. Tudo isso demanda tempo e ainda dificulta na leitura do código.

Código 2.5 Exemplo de dificuldade com objetos não tipados no *JavaScript*

```
1     function foo(bar){
2         const internalObject = {data: "Hello"}
3         console.log(internalObject.data);
4         bar. // Difícil saber as chaves que este objeto possui
5     }
6
```

Foi visando resolver este problema que surgiu a linguagem *TypeScript*, que, embora muito semelhante ao *JavaScript*, pois baseia-se nesta linguagem, traz a possibilidade da tipagem estática dos recursos da linguagem de forma explícita. Assim, qualquer variável e retorno de função deve ser de um tipo de dado explicitamente definido. Este tipo de dado pode ser um tipo padrão como *string* e número, ou um tipo criado a parte, como uma classe ou uma interface (FENTON; FENTON; SPEARING, 2014). O Código 2.6 demonstra um exemplo do mesmo Código 2.5, porém, agora com as tipagens bem definidas. Neste caso, é possível saber claramente que o objeto *bar* é do tipo da interface

Date e, portanto, contém todos os métodos definidos dentro da interface, dentre eles, o *getTime()* que retorna a data em formato de *timestamp*. Além de conhecer o tipo do parâmetro, também é definido o tipo do retorno da função, para que quando esta função seja chamada, o outro trecho de código terá de forma definida qual o tipo de dado que aquela função retorna. Como a função *getTime()* retorna um número, o tipo de retorno é *number*.

Código 2.6 Exemplo de tipagem de objetos no *TypeScript*

```
1    function foo(bar: Date) : number{
2        const internalObject = {data: "Hello"}
3        console.log(internalObject.data);
4        return bar.getTime() // getTime é um método da interface Date
5    }
6
```

2.7 GERAÇÃO AUTOMÁTICA DE CÓDIGO

É verdade que os SmCs trouxeram uma nova forma de automatizar processos, onde, algumas cláusulas definidas podem ser executadas sem intervenção humana. Mas, como explicado na Seção 2.1.5 (Vulnerabilidade de SmCs), eles também trouxeram alguns desafios em paralelo. Um grande desafio que relaciona-se diretamente com o problema das vulnerabilidades apresentado, é que, nem todos são capazes de entender e desenvolver um SmC usando uma linguagem de programação, sendo este um grande desafio para pessoas da área jurídica, por exemplo, mapearem cláusulas dos contratos legais em SmCs (FRANTZ; NOWOSTAWSKI, 2016).

Portanto, com este alto nível de complexidade na construção de SmCs, até mesmo os desenvolvedores acabam construindo SmCs vulneráveis, contendo *bugs* e falhas de segurança (MAVRIDOU et al., 2019), como muitos trabalhos também destacam (GAO et al., 2020; ZHANG; XIAO; LUO, 2020; SO et al., 2020; PACE; SÁNCHEZ; SCHNEIDER, 2020). Diante desses fatos, muitos autores estão propondo formas de facilitar o desenvolvimento do SmC, de forma que o seu criador não precise ter muito conhecimento técnico sobre programação de SmCs.

Um tema de interesse é a geração automática de código, que na engenharia de *software* já é um tema bem estudado (CHOUDHURY et al., 2018). Para chegar a essa adoção, os autores partem da definição de uma estrutura ou modelo que será utilizada para representar o SmC. Esse modelo geralmente é algo que o usuário (pessoa que usará a ferramenta para criar o contrato) sentir-se-á confortável para projetar um SmC com ele. Depois de criar o modelo, o usuário pode acionar um mecanismo que irá traduzir o modelo para código de SmC e, eventualmente, compilar este código (GARCÍA-BAÑUELOS et al., 2017). O modelo é uma representação abstrata do SmC, e o nível de abstração utilizada para este modelo vai depender dos objetivos do projeto. Quanto maior o nível de abstração de um modelo, maior é a compreensão de um ser humano em utilizar tal modelo. Quanto menor o nível de abstração, menor é o grupo de pessoas que conseguem utilizar tal modelo. A próxima Seção apresenta alguns trabalhos como exemplos concretos desse processo.

2.8 TRABALHOS RELACIONADOS

Nesta seção, alguns trabalhos relacionados são apresentados para evidenciar o estado da arte na área. As Tabelas 2.1 e 2.2 apresentam um comparativo de todos os 28 trabalhos encontrados com base em nove critérios de avaliação, onde, a primeira tabela compara os trabalhos com base em 5 critérios: (C1) se o trabalho propõe a criação de um framework; (C2) se permite a escrita automática de código de SmC; (C3) se permite a compilação do código gerado; (C4) se permite a implantação do contrato gerado; e (C5) se permite gerar o código de forma dinâmica, sem o uso de *templates*. A segunda tabela compara os trabalhos de acordo com 4 critérios: (C6) qual o modelo de SmC usado; (C7) qual a lingagem de SmC que permite gerar; (C8) se gera os contratos de forma genérica; e (C9) se permite gerar em tempo de execução. Estes critérios são essenciais para verificar quais as características dos trabalhos relacionados condizem com os mesmos objetivos do Gifflar.

Frantz e Nowostawski (2016) apresentam uma forma de gerar código Solidity a partir da estrutura ADICO, uma gramática para instituições criada por Crawford e Ostrom (1995). Seu objetivo é aumentar a acessibilidade da criação SmC, traduzindo a estrutura ADICO, criada por um ser humano, em código legível por máquina. Então, os autores primeiro modelam algumas regras institucionais com a estrutura e, em seguida, as frases são mapeadas e transcritas para o Solidity. No final, eles aplicam a proposta em dois exemplos de casos de uso.

Mavridou e Laszka (2018) apresentam o FSolidM, uma ferramenta realmente completa que usa máquina de estados finita como estrutura para gerar código Solidity. Uma vez que a estrutura é projetada graficamente, é possível gerar o código do Solidity a partir da máquina de estados finita, já incluindo métricas para prevenir ataques e também permitindo implantá-los na rede Ethereum.

O trabalho de Qin et al. (2019) vai um pouco além, propondo um *framework* conceitual para gerar código SmC a partir de linguagem natural. Para isso, utilizam um dicionário denominado CoDic, criado pelos próprios autores, e também uma Linguagem Natural de Máquina (MNL) para criar frases a partir do CoDic para representar o código SmC. Eles não mostram nenhuma implementação desta estrutura e não especificam a linguagem SmC que podem gerar.

A Caterpillar é uma ferramenta criada por López-Pintado et al. (2019), capaz de gerar código Solidity, e para isso, os autores utilizam a Notação e Modelo de Processos de Negócios (BPMN) para representar os contratos, então, um componente denominado "*Compiler*" é capaz de traduzir BPMN em Solidity, permitindo também implantar estes contratos a uma rede privada Ethereum e interagir com eles.

A Tabela 2.1 apresenta uma comparação da proposta com 28 trabalhos da literatura que abordam geração de código de SmC, incluindo os quatro trabalhos citados. Nesta tabela, pode-se observar se as ferramentas propostas seguem a estrutura de *framework*, quais as etapas de criação de SmC (escrita, compilação e implantação) que elas atendem e se fazem isto de forma dinâmica. O primeiro trabalho, refere-se ao projeto em questão, o qual, além de seguir a estrutura de um *framework*, oferece todas as etapas de criação de SmC, desde a escrita do código até a implantação da instância na blockchain e de forma

dinâmica, ou seja, sem utilizar códigos de contratos pré definidos para então gerar novos SmCs baseados nestes *templates*. Dos trabalhos relacionados, 18 não são *frameworks* e a maioria não oferece todas as etapas de criação à implantação do SmC (C2, C3 e C4), somente 9 oferecem todas as três etapas. Os trabalhos 20, 21 e 24 são os únicos que utilizam *templates* para ajudar na criação dos contratos (C5). Dos trabalhos que propõem *frameworks*, somente três deles (12, 16 e 19) se igualam à proposta desta dissertação considerando todas as características (C1 a C5).

Tabela 2.1 Comparação de trabalhos com base no tipo de ferramenta, etapas de criação do SmC e dinamicidade

N°	**Proposta**	**C1**	**C2**	**C3**	**C4**	**C5**
0	Gifflar	✓	✓	✓	✓	✓
1	(JURGELAITIS; BUTKIENĖ et al., 2022)		✓			✓
2	(RASTI, 2022)		✓			✓
3	(DWIVEDI; NORTA, 2022)		✓			✓
4	(JURGELAITIS et al., 2021)		✓			✓
5	(QASSE; MISHRA; HAMDAQA, 2021)		✓			✓
6	(ZHU et al., 2021)		✓			✓
7	(MERLEC; LEE; IN, 2021)	✓	✓			✓
8	(WÖHRER; ZDUN, 2020)		✓			✓
9	(EID, 2020)		✓			✓
10	(SKOTNICA; PERGL, 2020)		✓			✓
11	(HAMDAQA; METZ; QASSE, 2020)	✓	✓			✓
12	(LADLEIF; FRIEDOW; WESKE, 2020)	✓	✓	✓	✓	✓
13	(FU; ZHU, 2020)		✓	✓	✓	✓
14	(TAN et al., 2020)		✓	✓	✓	✓
15	(QIN et al., 2019)	✓	✓			✓
16	(LUO et al., 2019)	✓	✓	✓	✓	✓
17	(LÓPEZ-PINTADO et al., 2019)		✓	✓	✓	✓
18	(MAO et al., 2019)		✓			✓
19	(MAVRIDOU; LASZKA, 2018)	✓	✓	✓	✓	✓
20	(TRAN; LU; WEBER, 2018)		✓	✓	✓	✓
21	(CHOUDHURY et al., 2018)	✓	✓			
22	(GARAMVÖLGYI et al., 2018)		✓			
23	(ASTIGARRAGA et al., 2018)	✓	✓		✓	✓
24	(TRAN et al., 2017)	✓	✓	✓	✓	
25	(LIAO et al., 2017)	✓	✓	✓		✓
26	(FRANTZ; NOWOSTAWSKI, 2016)		✓	✓		✓
27	(WEBER et al., 2016)		✓	✓	✓	✓
28	(PETTERSSON; EDSTRÖM, 2016)		✓	✓		✓

Considerando a Tabela 2.2 e analisando melhor as três propostas mais relacionadas, Luo et al. (2019) apresentam a ferramenta de uma forma mais conceitual, apesar de apresentarem detalhadamente todas as estratégias e apresentar um exemplo de aplicação baseada em um cenário, não explicam muito sobre a implementação e como esse *framework* pode ser utilizado, além de não ser genérico (C8), devido ao tipo de estrutura de escrita ser no formato textual, o que dificulta o processo por precisar tratar ambiguidades como em Qin et al. (2019). Mavridou e Laszka (2018) propõem uma ferramenta

que é bem consolidada e envolve verificação de vulnerabilidades de contratos a partir da formalização em máquinas de estados, entretanto, não demonstra nenhuma forma de avaliação do *framework*, apenas mostra o funcionamento da plataforma. Ladleif, Friedow e Weske (2020) apresentam um trabalho mais completo, envolvendo tanto uma profunda explicação da metodologia, quanto da implementação, construindo um protótipo e realizando um estudo de caso como avaliação da proposta. Porém, não é um *framework* que gera contratos de forma genérica e, apesar de incluir um componente gerador de contratos a partir de diagramas BPMN, o objetivo do trabalho é mais voltado para a arquitetura da blockchain, ou seja, vai além do escopo de aplicação dos SmCs e envolve também arquitetura de nós e consenso entre eles. Somente os últimos dois citados coincidem com esta dissertação em relação ao tipo de linguagem gerada (Solidity), ainda assim, um dos objetivos do Gifflar é também integrar outras linguagens de SmC (mais detalhes no próximo Capítulo).

Tabela 2.2 Comparação de trabalhos com base na estrutura de escrita, linguagens de SmC, genericidade e geração em tempo de execução

N°	Proposta	C6	C7	C8	C9
0	Gifflar	JSON	Solidity	✓	✓
1	(JURGELAITIS; BUTKIENĖ et al., 2022)	UML	Solidity	-	
2	(RASTI, 2022)	SIMBOLEO	Chaincode	✓	
3	(DWIVEDI; NORTA, 2022)	SLCML	Solidity	-	
4	(JURGELAITIS et al., 2021)	PSM	*Chaincode*	-	
5	(QASSE; MISHRA; HAMDAQA, 2021)	Textual	Solidity	-	
6	(ZHU et al., 2021)	TA-SPESC	Solidity		
7	(MERLEC; LEE; IN, 2021)	Blocos	Solidity		
8	(WÖHRER; ZDUN, 2020)	Cláusulas	Solidity	-	
9	(EID, 2020)	XML	Solidity, C#		
10	(SKOTNICA; PERGL, 2020)	DEMO	Solidity	✓	
11	(HAMDAQA; METZ; QASSE, 2020)	Diagrama	Solidity, JS e Chaincode	✓	
12	(LADLEIF; FRIEDOW; WESKE, 2020)	BPMN	Solidity	-	
13	(FU; ZHU, 2020)	-	-	✓	
14	(TAN et al., 2020)	Interface Web	Solidity		
15	(QIN et al., 2019)	CoDic	-		
16	(LUO et al., 2019)	Textual	JavaScript		
17	(LÓPEZ-PINTADO et al., 2019)	BPMN	Solidity		
18	(MAO et al., 2019)	Blocos	Solidity		
19	(MAVRIDOU; LASZKA, 2018)	Máquina Estados	Solidity	✓	
20	(TRAN; LU; WEBER, 2018)	Diagrama	Solidity	-	
21	(CHOUDHURY et al., 2018)	Go	Go	✓	
22	(GARAMVÖLGYI et al., 2018)	UML	Solidity		
23	(ASTIGARRAGA et al., 2018)	Regras	Chaincode	✓	
24	(TRAN et al., 2017)	Interno	Solidity		
25	(LIAO et al., 2017)	Gherkin	Solidity		
26	(FRANTZ; NOWOSTAWSKI, 2016)	ADICO	Solidity		
27	(WEBER et al., 2016)	BPMN	Solidity		
28	(PETTERSSON; EDSTRÖM, 2016)	Idris	Serpent	-	

Apesar de todos os trabalhos citados serem ferramentas de geração de código SmC, todos eles focam em uma estrutura que primeiro será elaborada por uma pessoa e depois traduzida em código, sendo um processo que sempre depende do desenvolvedor, pois, todas as vezes que uma aplicação necessitar de um novo SmC em determinada situação, precisará do desenvolvedor para modelar o novo SmC, gerar o código, compilar, implantar na rede e integrar com a aplicação. A principal contribuição desta dissertação, é oferecer aos desenvolvedores a possibilidade de construir sistemas capazes de criar os modelos, gerar o código SmC, compilar, implantar e interagir por conta própria em tempo de execução, sem se preocupar com erros sintáticos, tudo de forma dinâmica, ou seja, sem o uso de *templates* como fragmentos de códigos já prontos.

Dessa forma, diferente dos trabalhos relacionados, a necessidade do desenvolvedor é só na etapa de criação do sistema que integra a proposta, depois disto, o próprio sistema terá a autonomia de gerar os códigos dinamicamente. Tal proposta é alcançada a partir da criação de um *framework* que permite sistemas gerarem códigos de SmC em tempo de execução. Assim, apresenta as seguintes funcionalidades: (i) criação do modelo do contrato, (ii) geração do código, (iii) compilação, (iv) implantação do SmC na blockchain, e (v) interação com contratos em tempo de execução.

2.9 RESUMO DO CAPÍTULO

Este Capítulo primeiramente apresentou uma introdução à blockchain, uma tecnologia que oferece uma rede P2P onde os nós podem realizar transações de forma rastreável, transparente, segura e sem a necessidade de uma terceira parte confiável, construindo uma explicação do funcionamento desta tecnologia a partir da primeira blockchain proposta, a Bitcoin. A segurança desta tecnologia é garantida devido à diversos artefatos implementados, dentre eles, o *ledger* distribuído entre os nós, que é estruturado em blocos interligados, sendo quase impossível adulterar o valor de uma transação em toda a blockchain e quebrar o protocolo de consenso entre os nós.

Outro conceito importante são os SmCs que surgiram com a plataforma blockchain Ethereum. Os SmCs são escritos na linguagem Solidity, uma linguagem Turing-completa, e são compilados para uma linguagem de baixo nível específica, para que sejam executados na EVM. Este novo paradigma permitiu que a blockchain possa transferir não só moedas digitais como também informação, impulsionando o uso da blockchain em diversas áreas.

No capítulo também foi abordado sobre a OOP, a qual hoje é bastante utilizada em várias aplicações. A OOP além de trazer uma abstração do mundo real, torna o código mais compreensível para o desenvolvedor, simplifica a manutenibilidade do código reduzindo os riscos de falhas e possibilita o reuso de componentes do código. As bibliotecas e *frameworks* são exemplos de aplicações que empregam esta abordagem.

A biblioteca é uma forma de encapsular funcionalidades que serão reutilizadas várias vezes, de forma que uma sequência de operações podem ser resumidas em uma única função ou método. Com as possibilidades que os SmCs concederam, a comunidade da plataforma Ethereum investiu bastante em facilitar o desenvolvimento de SmCs e DApps a partir da criação de *frameworks*. Um *framework* é um conjunto de abstrações que podem ser reutilizadas para resolver problemas similares, diferente das bibliotecas eles

são mais genéricos podendo conter várias bibliotecas. Com base na técnica utilizada, o Gifflar, proposto nesta dissertação, é classificado como *black-box*, o qual não depende excessivamente de artefatos da OOP e utiliza componentes que podem ser utilizados por uma aplicação via composição de objeto. Com relação à organização do desenvolvimento, o Gifflar apoia-se nos padrões de projeto, que são utilizados para padronizar um vocabulário no desenvolvimento de uma aplicação considerando as necessidades do problema, os padrões *Factory*, *Module* e *Builder* são exemplos utilizados neste projeto.

Por este projeto envolver também a Engenharia de Software, é indispensável falar sobre os princípios SOLID. Estes princípios trazem um conjunto de orientações que podem ser seguidas para que um software possa ter uma ótima escalabilidade, padronização, manutenibilidade, dentre outras características. A sigla é uma junção de cinco grandes princípios que surgiram com o intuito de guiar os desenvolvedores a construírem sistemas que vão além de somente cumprir com o objetivo para o qual foram projetados. Todos os cinco princípios foram explicados, destacando a melhor forma de desenvolver um código de qualidade.

Para melhor entendimento do próximo Capítulo, foram introduzidas as linguagens de programação *JavaScript* e *TypeScript* e também a estrutura JSON. O *JavaScript* é uma linguagem que foi feita especialmente para a Web e era utilizada inicialmente para determinar a forma como os componentes de um site se comportavam. Mas, com o Node.js, foi possível utilizar o *JavaScript* não só do lado cliente, mas também do lado servidor, possibilitando desenvolver aplicações *JavaScript* em uma máquina sem a necessidade de um navegador. O JSON foi uma grande contribuição desta linguagem, uma vez que facilitou bastante a troca de dados entre serviços. Este é bastante utilizado como formato de transferência de dados que acabou tomando o lugar do XML, o qual necessitava de algumas transformações de dados que tornava o processo mais exaustivo do que utilizar o JSON. A desvantagem do *JavaScript* é a omissão da tipagem dos dados. Por isso, surgiu o *TypeScript*, que, embora muito semelhante ao *JavaScript* por basear-se nesta linguagem, traz a tipagem explícita dos dados, o que facilita bastante durante o desenvolvimento e em futuras manutenções de código.

Os SmCs, apesar de oferecerem diversos benefícios, são acompanhados de um grande desafio. Por causa do alto nível de complexidade na construção dos SmCs, os próprios desenvolvedores acabam construindo SmCs vulneráveis, contendo *bugs* e falhas de segurança. Diante disto, vários autores estão propondo formas para facilitar o desenvolvimento dos SmCs. A geração automática de código, é uma destas formas, onde, a partir da modelagem de uma estrutura utilizada para representar os SmCs, o desenvolvedor pode acionar uma ação de traduzir este modelo para código de SmC e então compilá-lo. O Gifflar propõe empregar esta estratégia e, depois de encontrados 28 trabalhos relacionados, foi possível destacar a principal contribuição desta dissertação, que é a possibilidade de os desenvolvedores construírem sistemas capazes de criar os modelos, gerar o código de SmC, compilar, implantar e interagir com os SmCs por conta própria, em tempo de execução, sem preocupar-se com erros sintáticos, automatizando todo o processo de desenvolvimento de SmCs.

Capítulo 3

GIFFLAR: GERAÇÃO DE CONTRATOS INTELIGENTES EM TEMPO DE EXECUÇÃO

Este capítulo apresenta toda a estrutura do *framework* desenvolvido, começando com a explicação geral sobre o Gifflar, seguindo com a definição do núcleo do *framework* e da biblioteca Solgen. Sobre o núcleo é explicado como as ferramentas internas são organizadas assim como quais são os comandos que podem ser executados no terminal. Posteriormente são introduzidos os artefatos e componentes da Solgen, a aplicação dos princípios SOLID, demonstração de uso da ferramenta, limitações, código-fonte do *framework*, versão e propriedades suportadas, e, por fim, é apresentado um fluxograma que define de forma clara em que situações o Gifflar pode ser aplicado.

3.1 ESTRUTURA DO GIFFLAR

Como explicado anteriormente, o Gifflar visa facilitar o desenvolvimento de SmCs aos desenvolvedores objetivando, principalmente, possibilitá-los construir sistemas capazes de escrever, compilar, implantar e interagir com tais contratos de forma dinâmica. Portanto, a ideia do Gifflar é oferecer uma interface mais clara possível, abstraindo o desenvolvimento direto em linguagem de contrato, mas, ao mesmo tempo, permitindo ao máximo o desenvolvedor criar qualquer tipo de contrato que poderia desenvolver caso estivesse implementando um contrato diretamente com uma linguagem de programação. Ou seja, a abstração criada pelo Gifflar pretende simplificar, porém, não limitar o desenvolvimento de contratos a ponto de criar estruturas de código prontas, mas sim conceder liberdade de desenvolver qualquer lógica de SmC que deseja-se criar. Para isso, é necessário uma estrutura bem definida a fim de oferecer esta característica e também preparar o código do *framework* para simplificar futuras manutenções e atualizações de componentes.

É importante destacar também que o escopo do Gifflar é a nível de aplicação. Ou seja, o *framework* em si não é influenciado pelo funcionamento da blockchain que o sistema irá utilizar, nem os diferentes tipos de algoritmos de consenso. O sistema que utilizar o Gifflar pode até ser a criação de uma própria blockchain com seu algoritmo de consenso específico e sua própria estratégia em tratar sobre a tolerância a falhas bizantinas, mas

o *framework* neste caso seria utilizado somente para a estratégia de geração do código dos contratos e *deploy* deles nesta rede blockchain, toda a lógica da blockchain em si estará no sistema que está utilizando o *framework*. As próximas subseções explicam todo o planejamento e organização elaborados para o desenvolvimento do *framework*.

3.1.1 Núcleo (Core)

O núcleo do Gifflar é responsável por unir todos os recursos que foram desenvolvidos para resolver um problema específico. Ou seja, este foi projetado para integrar as diferentes ferramentas e bibliotecas responsáveis por conter todos os componentes necessários para a modelagem dos contratos. Hoje existem diferentes linguagens de programação para SmCs, por isso, o Gifflar deve conter diferentes bibliotecas, cada uma desenvolvida para uma linguagem de SmC diferente, porém, todas seguindo um mesmo padrão de desenvolvimento. Na versão atual, o Gifflar somente integra a biblioteca para criação de contratos em Solidity, a Solgen, mas futuramente pretende-se integrar bibliotecas para outras linguagens. A Figura 3.1 demonstra a estrutura do núcleo do *framework*.

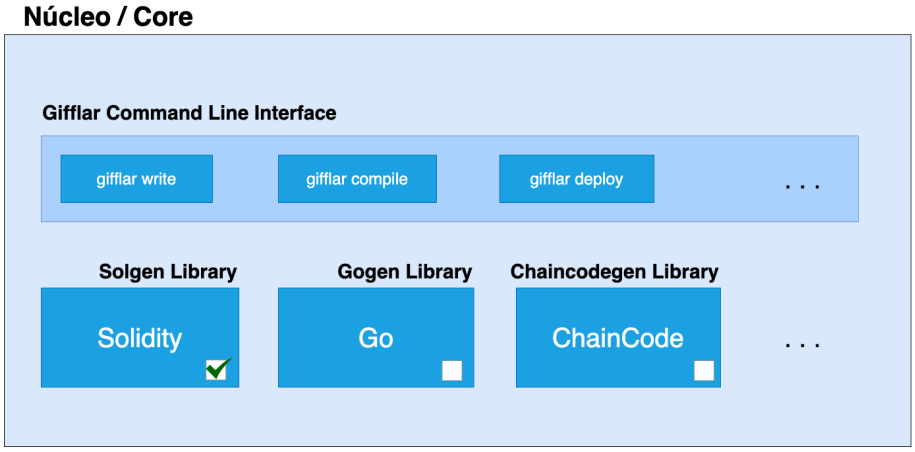


Figura 3.1 Estrutura do Gifflar *core*

Uma forma de facilitar o gerenciamento de um projeto criado com um *framework* e de utilizar algumas ferramentas oferecidas pelo mesmo, é ter uma interface para linha de comandos. O Gifflar oferece alguns comandos que auxiliam o desenvolvedor a utilizar os recursos providos pelo *core* do *framework*. Atualmente, os comandos disponíveis são:

- *gifflar init*: Inicia um novo projeto do Gifflar criando a estrutura de pastas e instalando as dependências necessárias.
- *gifflar make:model* [*file*]: Cria um novo arquivo para modelagem de um SmC. Já inclui um conteúdo padrão somente para guiar o programador a como iniciar com a modelagem. O *file* é o nome de arquivo do modelo a ser criado.
- *gifflar make:script* [*file*]: Cria um novo arquivo de *script* para implantação de um SmC. Já inclui um conteúdo padrão somente para guiar o programador a como

iniciar com a criação do *script*. O *file* é o nome de arquivo do *script* a ser criado. Este *script* deve conter algumas configurações necessárias para a implantação do contrato, como argumentos do construtor e o endereço de carteira que irá pagar a taxa de implantação do contrato.

- *gifflar make:service* [*file*]: Cria um novo arquivo de serviço para geração de SmCs. Já inclui um conteúdo padrão somente para guiar o programador a como iniciar com o serviço. O *file* é o nome de arquivo do serviço a ser criado. O serviço integra as funcionalidades de modelagem, escrita, compilação e implantação em um só lugar para acionar de forma interna em uma aplicação.

- *gifflar write*: Escreve o código Solidity dos modelos criados.

- *gifflar compile*: Compila os códigos Solidity dos modelos criados.

- *gifflar deploy*: Executa os *scripts* para implantação dos contratos inteligentes na rede blockchain que está configurada como padrão no arquivo de configuração do Gifflar.

- *gifflar help*: Imprime no console todos os comandos disponíveis do Gifflar.

O Gifflar foi projetado de forma a oferecer duas opções de desenvolvimento para o programador. De forma (a) estática ou (b) dinâmica. Na forma (a) estática, o programador utiliza o Gifflar para desenvolver contratos estaticamente, ou seja, desenvolve contratos manualmente, porém, utilizando a facilidade da modelagem do Gifflar (ver seção 3.1.3). Além disso o programador pode utilizar os comandos responsáveis por criar os modelos (*make:model*), criar os *scripts* (*make:script*) para realizar o *deploy* dos contratos, etc. A forma (b) dinâmica permite a criação de serviços, que são onde os desenvolvedores podem criar suas condições para geração automática de contratos e, por integrar as funções de escrita, compilação e implantação, portanto, utiliza somente o comando *gifflar make:service*.

3.1.2 Solgen (Solidity Generator)

3.1.2.1 Linguagem de Contrato Inteligente

O Gifflar foi desenvolvido inicialmente com somente uma biblioteca, a Solgen, a qual permite gerar códigos na linguagem *Solidity*. Foi escolhida esta linguagem de SmC porque a plataforma Ethereum já oferece atualmente diversas ferramentas que podem ajudar nas próximas etapas de aprimoramento da proposta. Outra vantagem de utilizar esta linguagem, é o fato de ela ser utilizada por várias blockchains além da Ethereum, e como futuramente pretende-se adicionar mais bibliotecas ao Gifflar, a fim de gerar códigos de contratos para diferentes blockchains, ou seja, diferentes linguagens de SmC, essa característica do *Solidity* já possibilita conquistar parcialmente esta funcionalidade.

3.1.2.2 Modelo para Geração de Contratos

A linguagem utilizada para representação dos SmCs foi o JSON. O motivo da escolha dessa estrutura, é que muitos sistemas passaram a utilizar o JSON como estrutura de mensagem, principalmente por oferecer um desempenho melhor que a XML, torna-se assim, mais fácil de comunicar-se com estes sistemas usando o mesmo formato de mensagem. Além disso, o *framework* é desenvolvido na linguagem TypeScript que é completamente baseada em JavaScript e, como o JSON surge do JavaScript, é muito mais simples trabalhar com ele e fornecer uma interface mais intuitiva para os desenvolvedores.

É importante destacar que o JSON pode ser a linguagem utilizada para representar os contratos, mas a partir do momento em que é recebido dentro do TypeScript, é analisado pelo *parser* e transformado em objeto JavaScript nativo. Poderia também ser utilizado diretamente o JavaScript nativo como modelo de representação de um SmC, mas isto limitaria o *framework* a operar somente com esta linguagem, não podendo expandir seu uso por outras linguagens. Com o JSON isso é possível. Por ser um formato utilizado para transferência de dados, facilita o processo em que um código em uma outra linguagem possa gerar o modelo do contrato e enviar o JSON, por exemplo, para um servidor que implementa o *framework*.

A Figura 3.2 ilustra o modelo de representação de um contrato a ser gerado pelo Gifflar. Seguindo a linguagem JSON, inicialmente é configurado a definição inicial de um SmC (chave *contract:*), em seguida o nome do contrato a ser criado (chave *name:*) e depois o conteúdo do contrato em si . Dentro do *contract:* são definidas as variáveis que este contrato deverá ter (chave *variables:*). Como em um contrato mais de uma variável pode ser declarada, a chave *variables:* recebe uma lista de objetos, onde cada objeto é a representação da criação de uma variável no SmC, considerando o tipo de dado, nome da variável, scopo (público, privado, etc) e valor de inicialização da variável (que é opcional).

Assim como a chave de variáveis, a chave *functions:* funciona da mesma maneira, porém, contendo uma lista de "objetos funções", onde são definidos o nome da função, escopo, se é um construtor (e, caso seja, não é necessário definir o nome, devido a alterações em versões mais novas do *Solidity*), os *inputs*, *outputs*, conteúdo da função, dentre outras configurações. Todas as chaves *inputs* recebem uma lista de argumentos, definindo o tipo e nome da variável de entrada da função. Já nas chaves *outputs*, caso a função tenha um retorno, somente é necessário informar o tipo das variáveis a serem retornadas, o nome delas é opcional. O conteúdo *content:* da função recebe uma lista de objetos *statements*, que são quaisquer ações que podem ser realizadas dentro de uma função, como atribuição de variáveis, *loops*, *if/else*, etc.

É certo que, comparado com os trabalhos relacionados, o JSON é um modelo o qual o nível de abstração é menor do que o nível de abstração utilizado nos modelos dos trabalhos relacionados. Assim, não é qualquer pessoa que consegue entender e modelar SmCs utilizando o Gifflar, somente um grupo específico: desenvolvedores de software. Como o objetivo do *framework* é permitir que desenvolvedores criem sistemas que geram códigos de SmCs em tempo de execução, utilizar um modelo de baixo nível é ideal para facilitar a representação do SmC em código TypeScript, principalmente o JSON, pelo motivo apresentado no parágrafo anterior.

```
{
  "contract": {
    "name": "MyContract",
    "variables": [
      {
        "type": "string",
        "name": "message",
        "scope": "public"
      }
    ],
    "functions": [
      {
        "name": "setMessage",
        "scope": "public",
        "isConstructor": false,
        "inputs": [{ "type": "string", "name": "_message" }],
        "outputs": [],
        "modifiers": [],
        "content": [
          {
            "statement": "assignment",
            "variable": "message",
            "expressionValue": { "customExpression": "_message" }
          }
        ]
      }
    ]
  }
}
```

Figura 3.2 Exemplo do modelo JSON padrão do Gifflar

Além disso, o fato de o modelo do Gifflar ser voltado para desenvolvedores permite que projetos semelhantes aos trabalhos relacionados possam se beneficiar deste *framework*. Ou seja, os desenvolvedores destes sistemas poderiam utilizar o Gifflar internamente para construir seus sistemas. Assim, o sistema permitiria os usuários desenvolverem SmCs em modelos de alto nível, mas internamente, o sistema estaria utilizando a modelagem do Gifflar, uma vez que foi criada especificamente para desenvolvedores utilizarem. Isso facilitaria o desenvolvimento destes sistemas uma vez que o Gifflar abstrai a sintaxe da linguagem de SmC.

Além disso, considerando que dois projetos utilizam o mesmo padrão de modelo (modelo do Gifflar), onde cada um dos projetos cria em cima do modelo do Gifflar, um modelo de alto nível. Como os dois utilizam o mesmo padrão internamente, existe a possibilidade de o modelo específico de um projeto poder ser traduzido no modelo específico de outro projeto que também utiliza o padrão de modelo JSON do Gifflar. Tal processo depende apenas do tipo de versão do *framework* que cada projeto está utilizando.

3.1.2.3 Aplicação dos Princípios SOLID

O Gifflar foi desenvolvido utilizando OOP, que é um paradigma já utilizado a muito tempo em diversos sistemas, e que apresenta suas vantagens como simplificar o entendimento do código, dentre várias outras apresentadas no Capítulo 2. Além de utilizar OOP, a organização de sua biblioteca Solgen está de acordo com os princípios SOLID. Esta Seção explica sobre esta adequação a estes princípios.

As classes do Gifflar são tratadas neste projeto como “componentes”. Estes foram desenvolvidos de modo que cada um tenha uma única responsabilidade que será explorada

por um componente superior como forma de reuso de código, estando desta forma de acordo com o primeiro princípio (P1).

Cada componente da Solgen foi criado com o intuito de dificilmente precisar abrir seu código para ser modificado. Isto porque cada componente representa uma estrutura da linguagem (e.g. estrutura de repetição, *if/else* e criação de variáveis), que neste projeto são chamadas de *statements*, e como cada *statement* já possui sua estrutura bem definida, raramente será necessário abrir o código do componente para realizar alguma alteração. Um dos motivos para modificar um componente seria a alteração da versão do compilador do Solidity, que acaba trazendo algumas diferenças na gramática da linguagem. Mas, como estas alterações podem impactar no uso dos métodos da Solgen, estas devem ser disponibilizadas em uma nova versão do *framework*, a qual também irá conter uma documentação informando quais foram as principais alterações que impactam no uso da biblioteca. Além disso, a Solgen permite que alguns componentes possam ser estendidos, de forma que possibilita alterar o comportamento dos seus componentes internos. Todas estas características estão de acordo com o princípio P2.

Mesmo que a Solgen foi desenvolvida com OOP, nenhum método foi sobrescrito. Todos os métodos foram criados de forma específica àquele componente e seguem especificações definidas por interfaces considerando as regras do princípio P3. Por não depender de classes concretas, mas sempre de interfaces, está também de acordo com o princípio P5.

Todas as interfaces da Solgen foram criadas de forma específica para cada componente. Dessa forma, é possível aproveitar todas as especificações da interface, não havendo a possibilidade de vários objetos serem somente de uma interface mais genérica e acabarem contendo métodos inutilizáveis. Ou seja, assim como os componentes, cada interface também possui responsabilidade única, o que está de acordo com o princípio P4.

3.1.2.4 Componentes da Solgen

A biblioteca Solgen possui uma organização bem estruturada a respeito dos relacionamentos de cada componente e faz o uso de padrões de projeto para definir, de forma clara, as responsabilidades de cada um. A Figura 3.3 mostra a organização das dependências dos componentes. As setas tracejadas na imagem indicam "dependência". Os primeiros componentes (Contract Manager e Contract) são componentes externos que um código poderá importar e usar. O componente Contract ou Gifflar Contract possui todas as "habilidades" para modelar, escrever, compilar e implantar um SmC, portanto, depende de todos os componentes abaixo. O Contract Manager ou Gifflar Contract Manager pode conter uma lista de instâncias de Contracts, de forma que seja possível, por exemplo, compilar vários contratos num mesmo momento.

Cada componente abaixo do Contract pode conter subcomponentes internos (por exemplo, Function Writer e Content Writer, no caso do componente Contract Writer) que são criados para resolver pequenas partes do problema e fornecer melhor organização de código seguindo o princípio P1, também facilitando testes de código. Conforme explicado antes, a proposta usa três tipos de padrões de projeto. Cada componente é um Module e também uma Factory, onde cada um, com o mesmo propósito, tem a mesma estrutura, e nenhum pode influenciar o outro. O componente Contract é um

Builder, que pode representar o SmC de diferentes formas (JSON, código *Solidity*, ABI e bytecode e instância SmC).

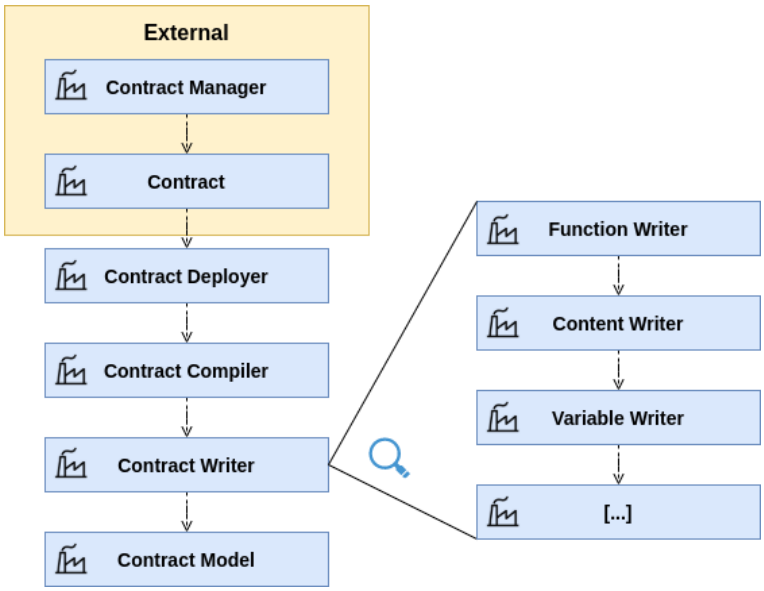


Figura 3.3 Relação de componentes da Solgen

O Solidity também permite a criação de interfaces e bibliotecas para utilizar dentro da criação de um contrato. Assim, um contrato pode implementar uma interface que determina as funções que este contrato deve ter, e este mesmo contrato pode utilizar uma biblioteca para uma determinada função como, por exemplo, uma biblioteca que exporta funções para cálculos matemáticos seguros. Por isso, o Gifflar também possui componentes para permitir a criação de interfaces e bibliotecas. São o Gifflar Interface e o Gifflar Library. Algumas diferenças em relação à criação de contrato é que interfaces e bibliotecas no Solidity não possuem construtores, e interfaces não possuem variáveis de estado. Todos os três tipos de componentes, Gifflar Contract, Gifflar Interface e Gifflar Library podem ser administrados dentro do Gifflar Contract Manager, também existe uma opção no Gifflar Contract para herdar as funções de uma interface.

3.1.2.5 Fluxo de Etapas

A Solgen está organizada em quatro etapas que seguem um determinado fluxo. A Figura 3.4 ilustra como cada etapa está relacionada e os recursos necessários para que o fluxo continue. As setas tracejadas significam "dependência" e a sólida demonstra a saída de cada estágio. A primeira etapa é a modelagem (*Modeling*), que, através do componente Contract Model, o qual poderá depender de outros componentes específicos (como Variable Model, Function Model, etc) representados pelo bloco Models, gera o JSON que é a linguagem de representação do SmC modelado.

O Contract Writer, na etapa de *escrita* (*Writing*), além de depender de componentes específicos, também precisa do JSON criado no estágio da modelagem. Uma vez

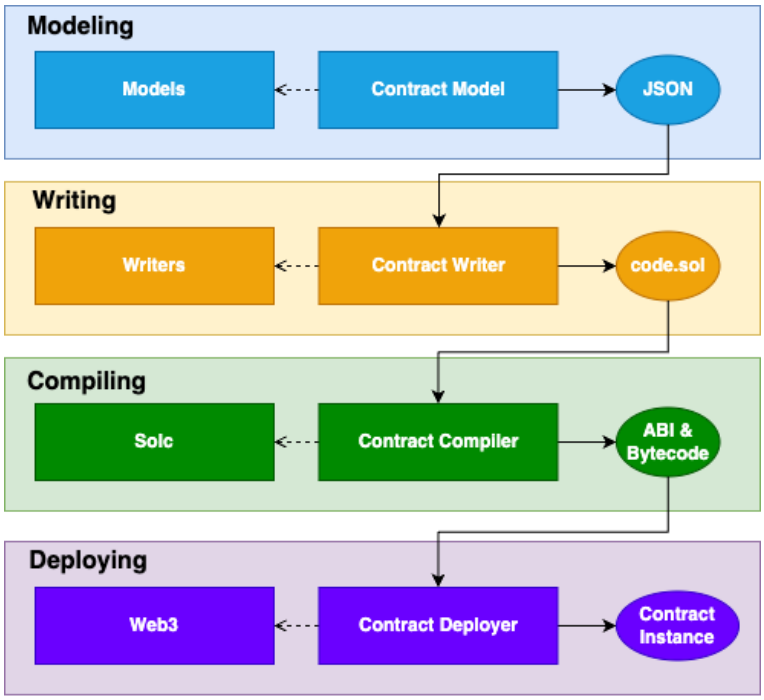


Figura 3.4 Organização de etapas da Solgen

que o CONTRACT WRITER escreve o código SmC (code.sol), o componente CONTRACT COMPILER, na etapa de compilação (*Compiling*), irá usar o compilador *Solidity*[1] (Solc) para compilá-lo e gerar a ABI e o *bytecode*, que são duas estruturas principais que serão usadas para implantar o SmC na blockchain (*bytecode*) e para interagir com ele (ABI).

Na etapa de implantação (*Deploying*), usando o *bytecode*, o CONTRACT DEPLOYER implantará o contrato na rede usando a biblioteca Web3.js[2] e irá retornar a instância do contrato para permitir interações com ele. Não é necessário passar por todas as etapas de uma vez, o chamador poderá parar em qualquer etapa e obter diferentes saídas de representação do contrato.

3.1.2.6 Fluxo Interno de Componentes na Etapa de Escrita

Para ilustrar melhor a sequência de criação dos componentes ao executar o estágio de escrita, a Figura 3.5 apresenta um diagrama de sequência reduzido para explicar como funciona o fluxo da etapa de tradução do JSON para o código *Solidity*. O SYSTEM, que importa a fábrica `createGifflarContract()` da Solgen, pode usá-la para criar uma nova instância de GIFFLARCONTRACT, que, por sua vez, usa as fábricas para criação do CONTRACTMODEL, CONTRACTWRITER, CONTRACTCOMPILER e CONTRACTDEPLOYER, mas para uma ilustração mais clara, apenas o CONTRACTWRITER foi incluído. Com o CONTRACTWRITER FACTORY, é criada a instância CONTRACTWRITER e re-

[1] https://github.com/ethereum/solc-js
[2] https://github.com/ethereum/web3.js

tornada para a instância do GifflarContract. Uma vez que o System já possui o objeto completo, ele usa alguns métodos para modelar o contrato (explicado ainda nesta Seção com a Figura 3.7) e então usa o método `write()` para submeter o modelo criado ao ContractWriter, que irá interpretá-lo e retornar o respectivo código gerado de volta ao System. Lembrando que este código gerado também é salvo no próprio objeto GifflarContract, de forma que pode ser solicitado novamente a qualquer momento além de retirar a necessidade de informar para o método de compilação qual o código que deve ser compilado.

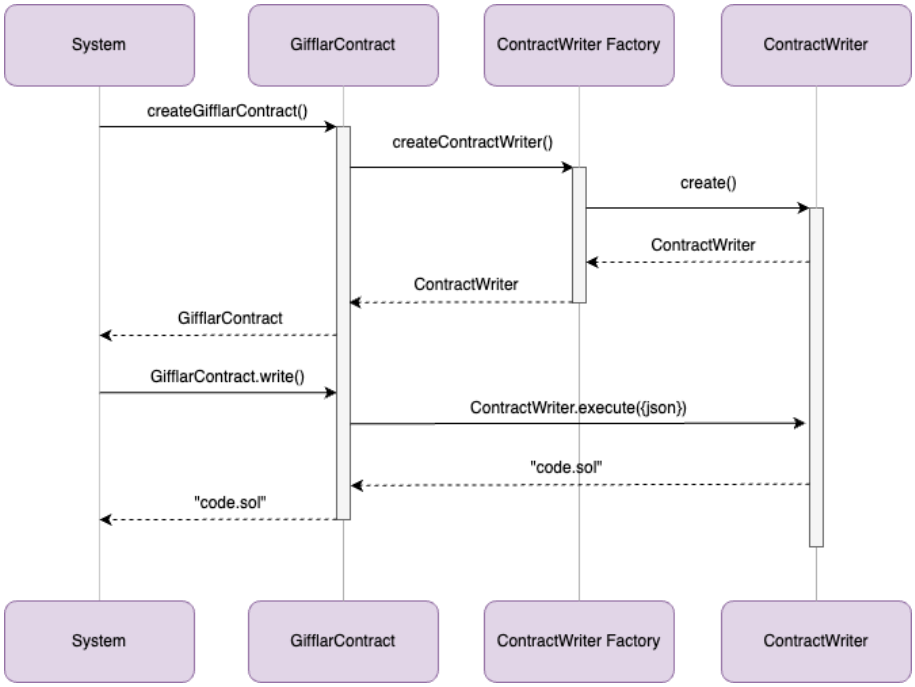


Figura 3.5 Diagrama de sequência da etapa de escrita do Gifflar

3.1.3 Uso do Framework

Na prática, o usuário utiliza o comando *gifflar init* para criar um novo projeto. Este comando também cria um arquivo de configuração *gifflarconfig.json* que define o caminho padrão das pastas do Gifflar, chave privada da carteira que realiza a implantação dos contratos, lista de configuração de redes blockchain que podem ser utilizadas e qual a configuração de rede padrão a ser utilizada nas implantações. A Figura 3.6 demonstra um exemplo de configuração de três redes blockchain no arquivo de configuração. Neste caso, foi configurada uma rede local, a rede principal da Ethereum e uma rede de teste da Binance Smart Chain (BSC), mas a rede padrão selecionada é a *local_network* (em *defaultNetwork*). Cada configuração de rede define uma chave de seleção da rede (*key*), o identificador da rede[3] (*networkId*), o quanto de taxa (GAS) o usuário está disposto a

[3]Podemos encontrar o identificador de uma rede no site https://chainlist.org/

pagar pelas transações, o preço que quer pagar por estas taxas (*gasPrice*) e o link do nó da rede onde irão ser solicitadas as execuções das transações (*nodeLink*).

```
"defaultNetwork": "local_network",
"networks": [
  {
    "key": "local_network",
    "networkId": 0,
    "gas": 3000000,
    "gasPrice": "10000000000",
    "nodeLink": "http://localhost:8545"
  },
  {
    "key": "ethereum_mainnet",
    "networkId": 1,
    "gas": 3000000,
    "gasPrice": "10000000000",
    "nodeLink": "https:// ... "
  },
  {
    "key": "bsc_testnet",
    "networkId": 97,
    "gas": 3000000,
    "gasPrice": "10000000000",
    "nodeLink": "https://data-seed-prebsc-1-s1.binance.org:8545/"
  }
]
```

Figura 3.6 Exemplo de trecho do *gifflarconfig.json*

Depois de criado o projeto com o Gifflar, o usuário pode utilizar os outros comandos para criação de modelos, *scripts* e/ou serviços. A criação dos arquivos é feita pelo núcleo do Gifflar através dos comandos executados pelo usuário, já a modelagem dos SmCs é feita a partir da importação da biblioteca Solgen. Os arquivos para serviços criados pelo Gifflar seguem um *template* com algumas importações e partes de código já definidos para guiar o usuário, porém, a Solgen pode ser utilizada da forma que melhor se adapta à aplicação. Já a criação de modelos e *scripts* estáticos precisam seguir a estrutura definida pelo Gifflar, pois os comandos de escrita, compilação e implantação dependem desta estrutura.

A Figura 3.7 mostra um exemplo da criação de um modelo de SmC utilizando a Solgen como pacote. Após a importação, o desenvolvedor pode definir a estratégia de modelagem do contrato, que será utilizada pelo sistema, a fim de gerar o JSON a partir de métodos do próprio objeto GifflarContract, para então prosseguir com a etapa de escrita. Seguindo o código de exemplo, depois de configurar o modelo do contrato, o método `.toString()` somente retorna o código JSON estruturado (ilustrado na Figura 3.2) em formato de *string*, então o `.write()` interpreta o JSON e retorna o código *Solidity* escrito (ilustrado na Figura 3.8), em seguida o método `.compile()` compila o contrato escrito, gerando os *bytecodes* e ABI e, finalmente, o método `deploy(...)` realiza a implantação do SmC na blockchain utilizando a biblioteca Web3.js e algumas configurações necessárias como o endereço que irá executar a transação e os argumentos do construtor do contrato. Outras configurações de implantação são definidas no arquivo de configuração *gifflarconfig.json*.

A Figura 3.8 mostra um exemplo de o resultado a ser gerado pelo Gifflar ao interpretar e traduzir o JSON da Figura 3.2 em código *Solidity*. A primeira linha mostra a versão

```typescript
// Factory
import { createGifflarContract } from "@gifflar/solgen";

// Creating contract model
const myContract = createGifflarContract("MyContract");

// Creating a contract variable
myContract.createVariable({ regularType: "string" }, "message", "public");

// Creating a contract function
myContract
  .createFunction("setMessage", "public")
  .setInput({ regularType: "string" }, "_message")
  .setAssignment("message", { customExpression: "_message" });

//Returning stringified model
myContract.toString();

//Returning JSON model
myContract.toJson();

//Writing contract solidity code
myContract.write();

// Compiling the solidity code
myContract.compile();

// Deploying the contract
myContract.deploy({ from: "0x ... ", args: ["Hello World!"] });
```

Figura 3.7 Exemplo de uso da Solgen como pacote no TypeScript

do compilador, que a propósito, será atualizada acompanhando cada nova versão do *framework*. As próximas linhas são a criação do código do contrato a partir do JSON recebido, separando cada espaço para cada chave definida no JSON, seguindo a sequência normalmente seguida por convenção, sendo, por exemplo, as declarações de variáveis primeiro, então possíveis definições de eventos, construtor da função, e demais funções do contrato. Além, disso, o Gifflar insere comentários e espaçamentos em cada fragmento de código, para caso de o usuário ter a intensão de parar na etapa de *escrita* e somente obter o código gerado, facilitando assim o seu entendimento sobre este código.

```solidity
pragma solidity 0.6.0;

contract MyContract{
//VARIABLES
string public message;

//FUNCTIONS
function setMessage(string memory _message) public{
message = _message;
}

}
```

Figura 3.8 Código gerado pelo modelo da Figura 3.2

3.2 DESAFIOS DA GERAÇÃO AUTOMÁTICA DE SMC

A geração automática de SmC em tempo de execução oferecida pelo Gifflar é uma abordagem que traz consigo alguns desafios. O primeiro deles é a garantia de uma correta sintaxe do código gerado, o que já é esperado por um *framework* para geração de códigos. Um segundo desafio é a semântica do código gerado, que já depende um pouco mais da

forma como o próprio desenvolvedor organizou as instruções. O último desafio, inclusive muito importante, é a garantia de que os códigos gerados são seguros, sem nenhuma vulnerabilidade. Este último é um pouco mais complicado, uma vez que, no caso do Gifflar, os contratos são gerados e implantados em tempo de execução, e não é possível esperar uma validação e correção manual dos contratos antes da implantação, tudo precisa ser de forma dinâmica. Esta seção faz uma discussão sobre estes três desafios mostrando como foram ou ainda serão considerados no desenvolvimento do Gifflar e também apresentando outras soluções.

O erro de sintaxe de um código pode ser qualquer uso incorreto da gramática da linguagem no momento da escrita do código. A sintaxe do código gerado pelo Gifflar é garantida através dos testes unitários realizados em cada componente. Além desses testes, o próprio compilador do Solidity permite obter uma resposta de falha caso houve algum erro de sintaxe no código gerado. É claro que, este último caso é bastante raro, uma vez que os testes unitários cobrem boa parte dos erros de sintaxe, portanto, caso este aconteça, será porque houve algum caso especial que não foi considerado pelos testes automáticos e, assim, no momento em que o usuário tentar compilar o contrato criado utilizando os modelos do Gifflar, este irá emitir um erro e, por isso, não irá permitir que o usuário realize a implantação do contrato na blockchain. Ou seja, se este caso acontece, pode-se dizer que seria uma falha benigna onde, por um lado, prejudica a usabilidade do *framework*, mas por outro, ainda impede o usuário de pagar a taxa de implantação de um contrato com um erro de sintaxe, pois a etapa de implantação só pode ser executada depois de obter a ABI e *bytcodes* da etapa de compilação.

Já o erro de semântica de um código vai além do uso incorreto da gramática. Podendo ser uma ordem de instruções ou operações que deveriam comportar-se de uma forma, mas não executam como esperado, muitas vezes por falta de atenção do desenvolvedor no momento de desenvolvimento, como por exemplo, uma divisão por zero ou uso de uma mesma variável contadora para laços de repetição que estão aninhados. A semântica dos contratos gerados pelo Gifflar apresenta-se ainda como um desafio, uma vez que o *framework* não possui nenhuma inteligência internamente, como por exemplo, uso de um modelo de aprendizado de máquina treinado, mas somente predefinições de cada elemento da linguagem Solidity que são utilizadas pelo usuário como uma ferramenta para livre personalização do SmC. Por oferecer uma liberdade para o usuário do Gifflar, a depender da organização dos dados feita na etapa de modelagem, pode-se gerar um erro de semântica que não é captada pelo *framework* em nenhuma das etapas de criação do SmC (modelagem, escrita, compilação e implantação), de forma que o contrato pode acabar sendo implantado na blockchain com um erro interno, o qual será revelado somente ao tentar executar certa operação e esta falhar.

Uma possibilidade de trazer esta verificação a nível semântico com o Gifflar em produção, seria adicionando certa inteligência para interpretar informações vindas de uma ferramenta de análise de SmCs e o próprio *framework* realizar as devidas correções do código baseando-se nestes resultados gerados pela ferramenta. São perspectivas que podem ser exploradas em trabalhos futuros. Assim, a nível de semântica de código, ainda existe uma demanda por parte dos usuários do Gifflar em ter atenção na organização das instruções do código na etapa de modelagem, ou seja, continua a mesma atenção à

semântica que precisa ter no desenvolvimento de SmC com Solidity.

Outro desafio que deve ser destacado é a prevenção de falhas de segurança que tornam os SmCs vulneráveis. Assim como a semântica, a detecção de possíveis vulnerabilidades no contrato ainda não é explorada dentro do Gifflar. A liberdade oferecida ao desenvolvedor que usa o *framework* faz com que este precise ter atenção ao desenvolver os contratos. Ou seja, como fundamentado no Capítulo 2, grande parte das vulnerabilidades dependem da forma como o próprio desenvolvedor irá utilizar nas instruções. O Gifflar não impede, por exemplo, de utilizar o *timestamp* do bloco para gerar números aleatórios, o que ocasionaria a falha de segurança "dependência do *timestamp*".

Porém, existem hoje vários trabalhos na literatura que apresentam o conceito de correção automática de SmCs vulneráveis, o que encaixa exatamente na necessidade do Gifflar. Zhang, Yuyao, et al. (2020) desenvolvem o SmartShield, um sistema que corrige automaticamente SmC com relação a três vulnerabilidades mais comuns, a verificação é feita a nível de *bytecode*, também realizam um experimento em 28.621 contratos com falhas na Ethereum e o sistema foi capaz de corrigir 91.5% deles. Nguyen, Tai D., Long H. Pham, e Jun Sun (2021) apresentam a sGuard, uma abordagem que possibilita corrigir de forma automática quatro tipos de vulnerabilidades mais comuns, a verificação e correção é feita a nível de código Solidity, também fazem um experimento em 5000 SmCs da rede Ethereum e todos os contratos que tinham falhas foram corrigidos. Rodler, Michael, et al. (2021) apresentam um framework chamado EVMPatch que também corrige SmC de forma automática, mas inclui a estratégia de permitir corrigir um SmC mesmo depois de implantado na rede, aplicando o padrão *Upgradeable Proxy* já utilizado pela comunidade, ao fim do estudo realizam um experimento em 14.107 contratos e o *framework* conseguiu corrigir aproximadamente 99.76% deles. Ferreira Torres, Christof, e Hugo Jonker (2022) introduzem a Elysium, uma ferramenta para reparo automático de contratos a nível de *bytecode*, que hoje é capaz de corrigir sete tipos diferentes de vulnerabilidades de forma automática que pode também ser estendida com novos modelos para localização de diferentes *bugs*, realizam um experimento comparando com outras ferramentas e os resultados mostram que a ferramenta consegue corrigir 30% mais contratos do que as outras. O interessante é que algumas destas ferramentas corrigem o contrato não só no sentido das vulnerabilidades, mas também com relação à semântica.

A maior garantia do Gifflar então está na sintaxe. Como visto em exemplos deste capítulo, o Gifflar facilita a implementação de SmCs sem a necessidade de o desenvolvedor conhecer muito sobre a linguagem do contrato que está utilizando. A liberdade oferecida pelo *framework* faz com que a semântica e a detecção de vulnerabilidades sejam consideravelmente comprometidas, porém, uma possível generalização da implementação de códigos de SmCs a ponto de utilizar partes de código já prontas, apesar de permitir um melhor controle da semântica e detecção de vulnerabilidades, fugiria dos objetivos e limitaria os sistemas a gerarem códigos de SmCs com um baixo grau de personalização. O desafio mais preocupante é a garantia de geração de contratos seguros, o que torna indispensável uma inclusão futura da abordagem de correção automática de SmCs dentro do Gifflar, uma vez que, no caso do Gifflar, não é possível realizar este processo de forma manual como tradicionalmente é realizado.

Tabela 3.1 Propriedades do *Solidity* atualmente suportadas ou não pelo *Gifflar*.

Propriedade	**Suportada**	**Exemplo**
Declaração de Variáveis	✓	string name; / string name = "Bob";
Atribuição	✓	name = "Bob" / name = getName()
Criação de eventos	✓	event myEvent(string name);
Chamada de eventos	✓	emit myEvent(name);
Criação de Funções	✓	function myFuntion() public {...}
Criação de Construtor	✓	function constructor() public {...}
Estrutura IF	✓	if(count == 1){...}
IFs aninhados	✓	if(){if(){...}}
Estruturas	✓	struct Person {string name;}
Modificadores	✓	modifier onlyOwner(){[...]; _; }
Enumerações	✓	enum Choice {One, Two, Three}
Mapeamentos	✓	mapping(address => uint) public mappingList;
Loops	✓	for(i=0;i<count;i++){...}
Herança	✓	contract Dog is Animal{...}
Assembly	✗	assembly {...}

3.3 CÓDIGO, VERSÃO E PROPRIEDADES SUPORTADAS

O Gifflar foi desenvolvido está com seu código-fonte disponível na plataforma GitHub[4]. Durante o desenvolvimento, foram também criados testes automatizados para certificar-se do correto funcionamento de cada componente. A Tabela 3.1 lista algumas das propriedades do *Solidity* que a Solgen suporta (representadas pelo símbolo ✓) e quais ainda serão adicionadas (representadas com um ✗). Esta tabela está considerando a versão do *Solidity* a qual o *framework* utiliza nesta primeira versão, que é a 0.6.0. Embora esteja definido na tabela que o *framework* não suporta instruções *assembly* (instruções de baixo nível do Solidity), a versão atual da Solgen fornece um método no qual o desenvolvedor consegue inserir instruções *assembly* ou qualquer outra diretamente em código Solidity, estas instruções somente não possuem métodos específicos para elas como as outras suportadas.

3.4 QUANDO UTILIZAR O GIFFLAR

As primeiras concepções do Gifflar surgiram como uma necessidade do projeto IoTCocoa (um dos projetos apresentados na Seção 5.4 do Capítulo 5), o qual, em síntese, busca usar uma ferramenta para gerar SmCs para dispositivos IoT através do próprio sistema. Entretanto, o *framework* não se limita ao escopo do IoTCocoa, ele pode ser aplicado em outras áreas dependendo somente das necessidades do projeto a ser desenvolvido, como por exemplo o segundo projeto que será apresentado no Capítulo 5 que é voltado para cadeia de suprimentos para produtos farmacêuticos. Visto que o Gifflar propõe um modo

[4]https://github.com/GifflarJS-Framework

inovador no desenvolvimento de SmCs, é interessante não só entender o funcionamento, como também saber em que momento é possível utilizá-lo. Para facilitar a explicação e também a decisão de futuros projetos em utilizar o Gifflar, foi criado um fluxograma (Figura 3.9).

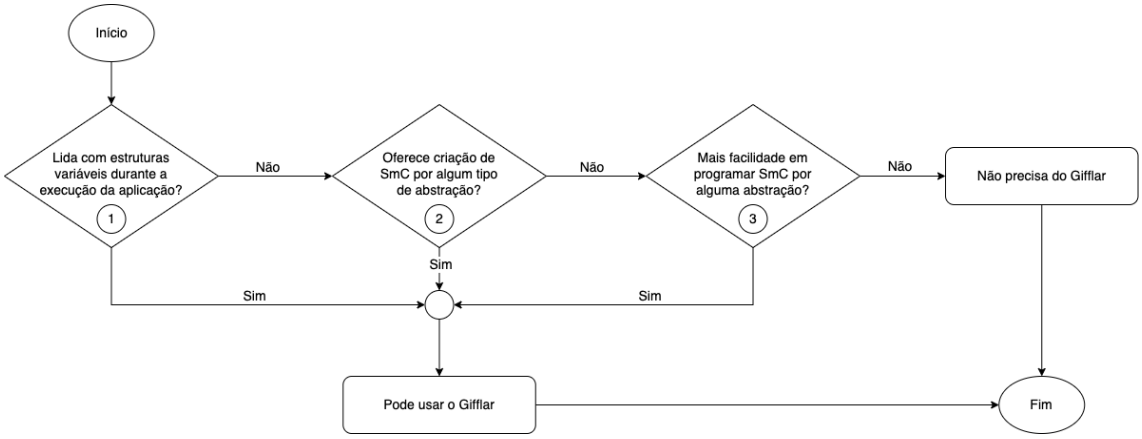


Figura 3.9 Fluxograma de decisão para uso do Gifflar

Esse fluxograma mostra de forma prática e resumida quais as principais condições que justificam o Gifflar ser utilizado em um projeto. Explicando com mais detalhes cada condição do fluxograma:

- Condição 1: Refere-se a projetos em que a aplicação proposta lida ou tende a lidar futuramente com estruturas as quais suas propriedades variam durante a execução da aplicação. Um exemplo prático é o projeto IoTCocoa, que faz o uso de dispositivos IoT, sendo que o sistema pretende lidar com futuros dispositivos até então desconhecidos pela aplicação, de forma que estes dispositivos possam "cadastrar-se" no middleware em produção (a Seção 5.4 do Capítulo 5 apresenta com mais detalhes o uso do Gifflar no IoTCocoa).

- Condição 2: Remete-se a projetos cujo objetivo é criar uma abstração na programação em Solidity. Por exemplo, um dos trabalhos futuros deste projeto é a criação de uma aplicação web do Gifflar, onde um usuário consegue acessar esta página e criar um SmC utilizando componentes habituais de formulários como botões e seleções. Outro exemplo são os próprios projetos relacionados citados na Seção 2.8 do Capítulo 2, que utilizam uma abstração de alto nível para criar SmCs. Até projetos como estes podem utilizar o Gifflar a fim de facilitar o desenvolvimento do próprio sistema.

- Condição 3: Trata de desenvolvedores que preferem programar de forma mais abstraída do que no próprio código-fonte. Esta condição refere-se ao uso do Gifflar de forma estática (ver Seção 3.1.1). É fato que um desenvolvedor que já possui uma experiência avançada em Solidity provavelmente não usaria a modelagem do

Gifflar por escolha, a não ser que tenha motivos como as condições 1 e 2. Entretanto, desenvolvedores com menos experiência nesta linguagem, e até mesmo em blockchain, buscam alternativas simples de iniciar na área, e o Gifflar pode ser uma escolha adequada para este momento (relatos de alguns participantes da avaliação no Capítulo 5 confirmam que o Gifflar contribuiu positivamente na evolução do conhecimento prático em SmCs).

3.5 RESUMO DO CAPÍTULO

Este capítulo descreveu toda a estrutura do *framework* para geração automática de SmCs, o Gifflar. Apesar de ele ser voltado principalmente para este fim, o Gifflar foi também projetado de forma a oferecer duas opções de desenvolvimento, de modo (a) estático ou (b) dinâmico. No modo (a) estático, o programador pode desenvolver SmCs de forma manual utilizando a modelagem do Gifflar. O modo (b) dinâmico permite a criação de serviços que integram todas as funcionalidades para um sistema requisitar a escrita de um contrato, compilação e implantação na rede de forma automática. Para isso, o núcleo do Gifflar (*core*) é dividido em vários integrantes, sendo os principais a interface de linha de comandos que é bastante utilizada na programação (a) estática e as bibliotecas que são muito bem exploradas na programação (b) dinâmica.

A intenção é que o Gifflar integre várias bibliotecas para várias linguagens de SmC. Porém, nesta versão inicial, o Gifflar possui somente uma biblioteca, a Solgen, que exporta uma API para gerar contratos na linguagem Solidity e implantar estes SmCs em plataformas blockchain que aceitam esta linguagem. A programação de SmCs com a Solgen utiliza o JSON como modelo. A modelagem do SmC nesta biblioteca então é feita inicialmente utilizando o JSON, depois de configurar o código JSON com todas as propriedades que o contrato terá (variáveis, funções, eventos, etc.), o programador pode acionar a funcionalidade de escrita do código para gerar o código-fonte Solidity através do JSON. Uma vez com o código-fonte gerado, este pode ser compilado e implantado na rede blockchain utilizando as funcionalidades da Solgen correspondentes a estas ações.

Toda a estrutura do Gifflar foi criada utilizando OOP, por ser um paradigma já consolidado e que simplifica o entendimento por outros desenvolvedores. Além disso, também está de acordo com os princípios SOLID. As classes do Gifflar seguem todos os 5 princípios SOLID fazendo o uso também de interfaces que, além de facilitar o desenvolvimento interno do *framework* também auxilia os usuários da biblioteca a entenderem quais as propriedades de um certo objeto que internamente implementa esta interface. Isto é possível porque o Gifflar é desenvolvido em TypeScript que permite a tipagem dos recursos da biblioteca.

A estrutura interna da Solgen é dividida em componentes, cada um responsável por uma funcionalidade específica. Um exemplo são os componentes Contract Model e Contract Writer que são responsáveis respectivamente por modelar o SmC através do TypeScript e escrever o código-fonte do SmC a partir do modelo JSON criado. Os componentes Gifflar Contract e Gifflar Contract Manager são componentes externos que podem ser importados em um projeto para modelar os SmCs. O Gifflar Contract integra todos os componentes internos do Gifflar responsáveis por modelar,

escrever, compilar e implantar SmCs; o Gifflar Contract Manager permite administrar vários Gifflar Contracts em um mesmo objeto. A sequência lógica das etapas de modelagem até a implantação do contrato na rede utilizando a Solgen é a seguinte: (a) o desenvolvedor modela o SmC utilizando métodos do objeto Gifflar Contract; (b) o método *write()* do objeto é acionada para gerar o código-fonte do SmC através do JSON configurado na etapa de modelagem; (c) o método *compile()* é invocado para compilar o código-fonte gerado na etapa anterior, originando a ABI e os *bytecodes* do contrato; (d) é realizada a implantação do contrato na rede blockchain acionando o método *deploy()*.

Foram identificados três desafios importantes de serem destacados. O primeiro deles é a garantia da sintaxe. Caso haja algum erro de sintaxe ao compilar o SmC, o Gifflar já emite um erro de compilação, impedindo que o SmC seja implantado. O segundo é assegurar que não há nenhum erro de semântica. Este ainda não é garantido pelo Gifflar, uma vez que depende muito da forma como o usuário irá organizar as instruções na etapa de modelagem. Porém, existem algumas possibilidades que podem ser exploradas em trabalhos futuros, como utilizar uma ferramenta de análise estática para analisar e então corrigir os erros semânticos. Também é de suma importância prevenir falhas de segurança nos SmCs antes de serem implantados. Para isso, foi encontrado uma abordagem proposta por vários artigos da literatura a qual permite corrigir falhas em um código-fonte de forma automática, algumas vezes até mesmo erros semânticos. Estes artigos serão estudados e futuramente será proposto uma integração desta abordagem com o Gifflar, a fim de garantir a segurança dos contratos gerados.

O Gifflar foi desenvolvido e está com seu código disponível na plataforma GitHub. A estrutura da linha de comandos foi criada e biblioteca Solgen já suporta parte da gramática do Solidity, as instruções que ainda não suporta podem ainda assim ser inseridas através de um método onde o desenvolvedor pode injetar instruções diretamente em código Solidity. Com o *framework* já bem estruturado, é possível também definir quando pode-se aplicá-lo a um projeto. Para isso, o Capítulo apresentou um fluxograma que mostra de forma prática quais as condições que justificam utilizar o Gifflar em um projeto.

Capítulo 4

ESTRATÉGIA DAS AVALIAÇÕES

A maior atenção de um projeto como o desta dissertação deve ser com a perspectiva do usuário. Não adianta somente desenvolver o projeto e certificar de que funciona, é necessário entender do próprio usuário do sistema quais são suas necessidades e dificuldades, se é coerente aplicar o *framework* em seus projetos empresariais ou pessoais. São questões como estas que formam a base para avaliar a usabilidade de um sistema (NIELSEN, 1994). O usuário final do Gifflar é o(a) desenvolvedor(a) de sistemas. Com o intuito de entender o ponto de vista destes usuários sobre a usabilidade do Gifflar, foi realizada uma avaliação de usabilidade em que estes puderam experimentar o Gifflar e opinar sobre a qualidade de sua usabilidade. Esta dissertação refere-se aos usuários que fizeram parte do experimento como "usuários", "participantes", "desenvolvedores" ou "programadores". A fim de verificar a aplicabilidade do Gifflar em outras abordagens, foi realizada uma revisão da literatura em busca de projetos que utilizam blockchain e que poderiam integrar com Gifflar para aprimorar suas funcionalidades no cenário em que se encontram. As próximas subseções explicam como foi planejada cada estratégia.

4.1 ESTRATÉGIA DE AVALIAÇÃO DE USABILIDADE

Sabe-se que o Gifflar possui um *core* onde estão definidos todos os possíveis comandos da interface de linha de comandos e também é onde está integrada a Solgen. Contudo, a linha de comandos é utilizada pelo usuário somente como uma facilidade durante o desenvolvimento do sistema com o *framework*, a maior parte do tempo o usuário estará utilizando a Solgen para a integração com o sistema. Por isso, a biblioteca Solgen foi o artefato de foco na avaliação de usabilidade.

Essa avaliação foi embasada no trabalho do Piccioni, Furia e Meyer (2013). Neste trabalho, os autores propõem um estudo de usabilidade de uma API. A metodologia do estudo consiste em 3 etapas. A primeira etapa é a definição das questões de pesquisa que guiam o estudo a fim de alcançar o objetivo de obter um feedback mais claro e preciso sobre a API. Na segunda etapa é explicado sobre como o estudo foi construído, também são definidas as duas ferramentas necessárias para a execução do protocolo, que

é o formulário de perguntas a serem respondidas pelo usuário e os tokens de utilidade a serem observados e determinados pelo condutor do experimento. A última etapa é a explicação da execução do estudo em si, onde é explicada toda a sequência de atividades do protocolo que foi seguida durante o experimento com cada usuário. Com base nestas três etapas do trabalho de referência, foram criadas três subseções (Questões de pesquisa, Elaboração da estratégia e Execução da estratégia) para apresentar respectivamente as três etapas no contexto da proposta em questão.

4.1.1 Questões de pesquisa

As questões de pesquisa são as mesmas do trabalho de referência. Neste, os autores criaram quatro questões que guiaram o estudo, mas são questões suficientes para serem aplicadas na avaliação do Gifflar, uma vez que a biblioteca Solgen também é considerada uma API.

- **QP1**: Qual é o esforço necessário para entender a semântica dos recursos da API com base em seus nomes e documentação?
- **QP2**: O nível de abstração da API atende à usabilidade?
- **QP3**: A estrutura da API facilita a reutilização e a concisão no código do cliente?
- **QP4**: O uso da API pode ser aprendido de forma fácil e incremental?

QP1 busca um entendimento melhor sobre o que o usuário precisa fazer para entender o fluxo e funcionamento da biblioteca e também como que cada método deve ser executado. Ela envolve questões como: a documentação dos métodos está suficientemente clara? Os nomes dos métodos estão intuitivos? O fluxo e a forma de acesso aos recursos (criação de objetos, chamada de método, etc) são compreensíveis?

QP2 procura verificar se o nível de abstração está adequado para a usabilidade da biblioteca. Ou seja, os usuários precisam ser capazes de utilizar a biblioteca sem a necessidade de entender implementações internas dos recursos. Além disso, a abstração deve seguir as convenções e padrões já utilizados por desenvolvedores de software, caso contrário, a biblioteca iria abstrair as implementações internas, porém iria reduzir a clareza do acesso aos recursos da API.

QP3 avalia se o código criado utilizando a biblioteca é conciso, permite uma manutenção sem muita complexidade e permite estender com facilidade. Ou seja, caso para uma certa aplicação o uso da biblioteca gera uma quantidade de código muito grande e complexa, a manutenção futura deste código pode ser bem complexa.

QP4 foca em avaliar o processo de aprendizado do usuário. A intenção é que o aprendizado seja gradual, de forma que naturalmente um usuário consiga lembrar de utilizar um recurso que já utilizou antes em uma outra funcionalidade, sem muito esforço, mesmo que estas funcionalidades sejam diferentes. Por mais que seja uma questão de pesquisa relacionada à QP1 e QP3, ela traz um destaque maior no processo de aprendizado do que em tarefas mais práticas.

4.1.2 Elaboração da Estratégia

Avaliar a usabilidade de um sistema é uma tarefa um tanto complexa. Atingir resultados totalmente confiáveis não é trivial, uma vez que cada desenvolvedor tem sua forma e hábitos de programar, além disso, cada pessoa tem seu método de aprendizado. Por isso, os autores Piccioni, Furia e Meyer basearam-se em um *framework* de dimensões cognitivas (BLACKWELL et al., 2001), que a partir deste ponto será apelidado de CDF (do inglês *Cognitive Dimensions Framework*). O CDF baseia-se em apresentar aos usuários *tasks* (tarefas) de programação, nas quais eles vão seguir as requisições de cada *task* com o objetivo de criar um código que utiliza os recursos da biblioteca. O intuito deste processo é realizar uma comparação da expectativa que o usuário tem com a realidade encontrada ao utilizar a API. Como uma forma de reduzir a subjetividade nesta avaliação de usabilidade, as diferenças dos dois lados da comparação são capturadas de duas fontes diferentes.

Logo depois de o participante completar as *tasks*, ele participa de uma rápida entrevista com dezoito perguntas. Cinco destas perguntas são sobre sua experiência em JavaScript, Blockchain e Solidity, e as outras treze são sobre a experiência de utilizar a Solgen. Este questionário permite entender sob a perspectiva do usuário quais foram suas dificuldades, se houve alguma necessidade que não foi atendida por nenhuma funcionalidade da API e também qual sua opinião sobre os recursos utilizados. A seção 4.1.2.1 apresenta todas as questões em mais detalhes e as relaciona com as questões de pesquisa da seção 4.1.1.

Além das questões da entrevista, outros dados foram obtidos implicitamente de eventos que ocorreram com os usuários durante a execução das *tasks*. Foi realizada uma chamada de vídeo online com cada participante. Cada chamada foi gravada com o consentimento dos participantes, e nenhum deles foi obrigado a ligar a câmera, foi necessário somente o áudio e o compartilhamento da tela do código onde estava executando as *tasks*. Também foi solicitado a eles para que seguissem o protocolo *Thinking aloud* (Pensando alto). Este é um protocolo baseado no trabalho de Ericsson e Simon (1993), e consiste em requisitar aos participantes para expressar em voz alta pensamentos que tiver sobre alguma ação que está realizando durante a execução das *tasks*, seja alguma dificuldade, uma dúvida ou uma escolha justificada. O uso deste protocolo foi importante para também obter dados destes eventos sob a perspectiva do condutor do experimento, não limitando-se somente a dados respondidos pelo participante, até porque podem existir dados importantes sobre o uso da API que surgiram durante o processo e que depois foram esquecidas por ele de evidenciá-las no questionário. Como as chamadas foram gravadas, o condutor conseguiu rever as gravações e registrar tudo aquilo que não foi registrado durante a chamada ao vivo. Estes dados obtidos pelo condutor foram classificados em tokens de usabilidade, os quais expressam justamente as diferenças entre expectativa e realidade para o participante durante a chamada, de modo que é possível apresentar de forma mais organizada, além de permitir relacionar cada token com as questões de pesquisa.

4.1.2.1 Questões da Entrevista

A entrevista consiste em dezoito perguntas. Cinco delas são questões sobre o conhecimento de JavaScript, Blockchain e Solidity do participante. Doze são questões baseadas no trabalho de referência que, neste caso, diz respeito ao Gifflar, e a última é uma pergunta opcional sobre qualquer recomendação de melhoria que o participante possa ter depois de completar o experimento.

Questões relacionadas ao participante:

1. Como você nivela seu conhecimento em JavaScript?
2. Você tem quantos anos de experiência com o JavaScript?
3. Como você nivela seu conhecimento teórico em Blockchain?
4. Como você nivela seu conhecimento em Solidity?
5. Você tem quantos anos de experiência com o Solidity?

Questões relacionadas à QP1:

6. Você acha que os tipos da API condizem com os conceitos do domínio como você esperava?[1]
7. Você precisou buscar com frequência informações na documentação sobre os recursos da API para resolver as tarefas?
8. O código necessário para completar as tarefas condiz com suas expectativas?

Questões relacionadas à QP2:

9. Você acha que o nível de abstração da API é apropriado para resolver as tarefas?
10. Você precisou adaptar a API para estar de acordo com as suas necessidades?
11. Você acha que precisou entender a implementação interna da API para poder utilizá-la?

Questões relacionadas à QP3:

12. A quantidade de código necessária para cada tarefa parece exagerada ou insuficiente para você?
13. Você acha que foi fácil avaliar seu próprio progresso na resolução das tarefas?
14. Você acha que teve que escolher uma forma (dentre várias outras) para resolver uma tarefa?
15. Você acha que teria que alterar muito seu código para mudar o tipo de rede blockchain?[2]

[1] Leia-se "tipos": nomes dos métodos, nome dos parâmetros, a forma de acesso aos recursos, etc.
[2] Questão 15 é uma pergunta modificada para adaptar-se ao projeto em questão.

Questões relacionadas à QP4:

16. Uma vez que você completou as primeiras duas tarefas, foi mais fácil completar as outras?

17. Você acha que teve que aprender várias classes e dependências dos componentes da biblioteca para resolver as tarefas?

Questão relacionada à sugestões:

18. Alguma recomendação de melhoria?

As primeiras cinco questões tem o objetivo de verificar o quão básico é o conhecimento de um participante. As perguntas sobre o nível de conhecimento, no caso do JavaScript, apresentou como possíveis respostas: "Básico", "Intermediário" e "Avançado", no caso da blockchain e Solidity, apresentou as opções: "Nenhum", "Básico", "Intermediário" e "Avançado". A intenção de saber esta informação é porque isto também pode influenciar nos resultados da pesquisa. É esperado que quanto menos experiência um desenvolvedor tem, mais dificuldade ele pode ter em entender o funcionamento da biblioteca. Por outro lado, espera-se que um desenvolvedor com conhecimento avançado pode sentir menos complexidade no entendimento dos recursos da biblioteca. Ou seja, são questões que podem ajudar a verificar possíveis casos que podem ser ameaças aos resultados finais da avaliação.

As próximas treze perguntas visam capturar dados sobre o ponto de vista do usuário em relação à biblioteca depois de ter a experimentado. Para cada pergunta foram apresentadas as opções de resposta: "Sim", "Não" e "Algumas vezes" (são opções de respostas baseadas naquelas utilizadas no trabalho de referência). O resultado destas questões foi essencial para compreender, sob a visão do usuário, as possíveis melhorias necessárias para a biblioteca, as quais não poderiam ser obtidas pelo próprio desenvolvedor da biblioteca, pois seriam informações que não estariam considerando o usuário final. Cada pergunta está relacionada com uma questão de pesquisa, e por isso foram classificadas em grupos diferentes. Por fim, a última questão consiste em obter qualquer recomendação que não foi abordada pelas outras perguntas, portanto é uma pergunta subjetiva onde o usuário tem liberdade para expressar quaisquer observações e/ou sugestões.

4.1.2.2 Tokens de Usabilidade

Para cada evento gerado pelo usuário durante a chamada, foram criados tokens que representam a ocorrência de cada evento. Um evento pode ser qualquer reação do participante sobre algum recurso da biblioteca que não ficou claro ou não condiz com suas expectativas. Com os tokens de usabilidade é possível representar cada ocorrência e apresentar de forma organizada. A criação dos token aconteceu no momento em que o condutor revisitou cada gravação e observou o acontecimento de cada evento. Foram detectados 28 eventos (uma síntese é apresentada no Capítulo 5), sendo vários compartilhados por participantes diferentes. Cada evento foi observado e classificado em tokens, totalizando em sete classificações de tokens, alguns destes aproveitados do trabalho de referência

(“surprise”, “choice”, “missed”, “incorrect” e “unexpected”). Cada classificação de tokens é explicada abaixo, sendo que os exemplos utilizados nesta explicação são exemplos genéricos.

- Token “surprise”: O desenvolvedor encontra algo na API que vai contra suas expectativas. Por exemplo, um determinado objeto principal da biblioteca Solgen é criado a partir de uma fábrica e não instanciando uma classe com a palavra reservada “new”, o que pode ir contra a expectativa de alguns usuários, uma vez que grande parte das vezes um objeto principal é criado a partir de uma classe importada.

- Token “choice”: O usuário se depara com opções diferentes de desenvolvimento, e deve entender e escolher uma das opções para prosseguir. Por exemplo, o elemento de uma *array* pode ser acessado por um método “*arr.at(0)*” ou pela própria notação de *array* “*arr[0]*”. O usuário então escolhe uma delas e justifica.

- Token “missed”: É observado que o usuário esqueceu de alguma funcionalidade ou abstração que já utilizou antes. Por exemplo, utilizou em alguma *task* o método “*filter()*” para filtrar elementos de um array, mas quando precisou novamente, demorou um tempo buscando mais uma vez na documentação ou implementou de uma outra forma.

- Token “incorrect”: O desenvolvedor implementa uma funcionalidade da API incorretamente, de forma que reproduza um erro. Por exemplo, aciona um método que recebe como parâmetro uma *string*, mas está repassando como *int*, então um erro pode surgir no próprio editor de código-fonte ou depois que executar o código.

- Token “unexpected”[3]: O usuário completou uma *task* mas de forma não esperada pelo condutor. Por exemplo, o usuário deveria criar uma variável e inicializá-la com um valor na mesma linha de instrução, porém ele criou a variável em uma linha e depois realizou uma atribuição; isto não resulta em nenhum erro, somente foge do que era esperado.

- Token “read_doc”: O desenvolvedor leu a documentação mas não entendeu como utilizar uma certa funcionalidade. Este token representa dois eventos observados. No primeiro, o desenvolvedor achou a documentação sobre a funcionalidade, leu e não entendeu. No segundo, o desenvolvedor tem muita dificuldade em encontrar a documentação sobre a funcionalidade. Os dois eventos podem ter surgido por três fatores: (a) a documentação não está clara; (b) o próprio usuário não é acostumado com este tipo de documentação; ou (c) O nível de inglês do desenvolvedor não foi o suficiente para entender ou encontrar sobre a funcionalidade (estes fatores são explicados com mais detalhes no Capítulo 5).

[3]No trabalho de referência o token “unexpected” representa um evento em que o usuário utiliza a API de forma não documentada, porém, na avaliação do Gifflar, nenhum usuário resolveu as *tasks* de forma diferente de como está na documentação. Mesmo assim, o token foi reaproveitado para uma outra finalidade.

- Token "not_found": Uma certa funcionalidade não foi encontrada pelo usuário. Este evento foi detectado em momentos em que o próprio desenvolvedor, depois de tentar encontrar a funcionalidade, verbalizava frases como "Não consegui encontrar esta funcionalidade" ou "Vou precisar de uma ajuda para encontrar este método".

4.1.3 Execução da Estratégia

Esta secção descreve com detalhes a execução do protocolo da avaliação de usabilidade do Gifflar. Aqui são listadas as *tasks* que os desenvolvedores realizaram para guiar o teste de usabilidade do Gifflar. Também é apresentado o ambiente de desenvolvimento online que os participantes utilizaram durante a execução das *tasks*; todos os participantes utilizaram este mesmo ambiente para contribuir com a uniformidade da avaliação de forma que nenhum participante teve vantagem em relação a este fator. A seção também trata sobre a origem e experiência dos participantes. Por fim, é detalhado o protocolo seguido pelo condutor da avaliação com cada participante.

4.1.3.1 Tasks

As *tasks* seguiram uma dificuldade gradual, para proporcionar um aprendizado também gradual. Além disso, como explicado no início do capítulo, o foco das *tasks* foi em testar a usabilidade da Solgen, então os comandos do Gifflar não foram muito explorados, apenas um comando foi utilizado na última *task*, mas somente para fins de conhecimento. Esse comando foi o *gifflar deploy*, que foi acionado somente para permitir o participante ter uma interação e entendimento melhor ao ver o SmC na rede blockchain sem ter programado em Solidity, portanto, o comando não foi considerado em nenhuma das questões da entrevista ou mesmo na classificação dos tokens, embora todos os usuários ficaram fascinados com este último exercício. As *tasks* são apresentadas nesta seção de forma resumida, a versão original seguida pelos usuários está no Apêndice A.

- **Criando Gifflar Model**: Esta *task* consiste em importar a fábrica de criação do objeto de modelagem do Gifflar (Contract Model) que é uma abstração do SmC do Solidity.

- **Desenvolvendo Contrato Inteligente com Gifflar Contract Model**: Esta é a *task* mais desafiadora para o desenvolvedor. Ela possui *subtasks* para dividir melhor cada operação. Nesta, o usuário deve modelar o SmC utilizando métodos do Contract Model criado na primeira *task*.

- **Criando Gifflar *Script* para implantação do contrato na blockchain:** Nesta etapa, o desenvolvedor já modelou o SmC e deve criar um arquivo que servirá para escrever o script de implantação do contrato na rede blockchain.

- **Configurando escrita e compilação do contrato:** Ainda utilizando o Contract Model, o qual já é injetado dentro do *script* de implantação, nesta *task*

o usuário deve primeiramente encontrar o método do CONTRACT MODEL para escrever o código Solidity a partir da modelagem realizada e, em seguida, o método que representa a compilação deste código. As duas ações são divididas em *subtasks*.

- **Construindo *script* de implantação na rede blockchain *testnet*:** Nesta última *task* o desenvolvedor deve realizar 4 *subtasks*. A primeira é acionar o método de implantação do contrato na rede, passando os parâmetros definidos pela própria *task* (argumento do construtor do contrato e carteira blockchain responsável por implantar o contrato). Depois deve definir dentro do arquivo de configuração do Gifflar a chave privada da carteira blockchain e alterar a rede onde será feito o *deploy* (mais detalhes sobre a rede na Seção 4.1.3.2). Posteriormente deve encontrar o método que retorna a instância deste contrato na rede blockchain. Em seguida deve copiar um trecho de código que não depende da Solgen, mas sim da biblioteca Web3.js. Por fim, o participante aciona o comando *gifflar deploy* para executar o código criado.

É importante destacar que nenhum participante usou sua carteira pessoal da blockchain para realizar este processo, a carteira utilizada para a implantação dos contratos na rede foi uma carteira criada pelo próprio condutor, que inclusive já tinha definido por padrão a chave privada desta carteira na configuração, de modo que o desenvolvedor somente precisou encontrar a propriedade onde definia a chave privada, mas não precisou inserir outra chave.

O arquivo de apresentação das *tasks* sofreu atualizações durante as avaliações com os primeiros participantes. Estas atualizações foram relacionadas à clareza de algumas frases. Conforme os primeiros participantes foram executando as tasks, o condutor detectou algumas interpretações erradas do usuário ao ler a sentença da *task*; nestes casos o condutor explicou de forma mais clara para o usuário, posteriormente atualizou o texto das *tasks* para os próximos participantes. Este fator aconteceu quatro vezes, ou seja, com quatro participantes.

4.1.3.2 Ambiente de Desenvolvimento

Todos os usuários utilizaram um mesmo ambiente de desenvolvimento, a fim de não permitir que um participante possa ter uma vantagem sobre os outros pelo fato de o seu ambiente de desenvolvimento local já fornecer algumas facilidades. Por isso, foi utilizada uma máquina Linux online que permite o desenvolvimento colaborativo chamada GitPod[4]. Este sistema online disponibiliza um editor de código-fonte semelhante ao editor Visual Studio Code[5], um editor bastante conhecido por vários desenvolvedores, inclusive pelos próprios participantes deste experimento. O GitPod permite criar um *workspace* (espaço de trabalho) onde é possível gerenciar um projeto, permitindo acesso também à linha de comandos do terminal. Porém, um *workspace* do GitPod só pode ser criado se for conectado a um repositório do GitHub, por isso, foi necessário inicialmente criar um

[4]https://gitpod.io/
[5]https://code.visualstudio.com/

projeto localmente utilizando o Gifflar (*gifflar init newProject*) e então salvar este projeto em um repositório do GitHub. Com isso, foi possível conectar o repositório do projeto no GitPod e criar um novo *workspace* onde também foi instalado o Gifflar utilizando o Node.js versão v14.19.1. Além disso, os participantes precisaram acessar o GitPod com sua própria conta do GitHub, mas isto de forma alguma permitiu que o condutor tivesse acesso a qualquer repositório do participante, mas sim o participante que teve acesso ao repositório criado pelo condutor.

O *workspace* uma vez configurado, foi possível definir no arquivo de configuração do Gifflar a chave privada da carteira utilizada para realizar o deploy dos SmC de cada participante, e também os dados da rede blockchain utilizada[6]. Com o intuito de mostrar uma segunda opção de rede blockchain que pode ser usada no Gifflar, foi utilizada a rede de teste da blockchain Binance Smart Chain (BSC), uma blockchain que foi criada com base na Ethereum (portanto também utiliza Solidity como linguagem de SmC), mas que implementa um outro protocolo de consenso. Como foi utilizada uma rede de teste, não houve gastos reais, somente gastos de moedas de teste, que foram obtidas através do site oficial da BSC específico para solicitar moedas para este fim[7].

4.1.3.3 Participantes

A avaliação teve a participação de 22 desenvolvedores (homens e mulheres), sendo que 9 são estudantes de bacharelado no curso de ciência da computação na Universidade Estadual de Santa Cruz (UESC), 1 estudante de bacharelado no curso de análise de sistemas na Universidade Estácio de Sá, 1 estudante de doutorado em ciência da computação na UFBA e 12 programadores que já trabalham na área de computação (um deles é estudante e também trabalha na área). Todos participaram como voluntários e a única restrição é que tivessem no mínimo conhecimento básico em JavaScript. Leia-se "conhecimento básico" aquele programador que sabe criar variáveis, objetos (assim como desestruturação de objetos), funções e que já utilizou estrutura de classes em algum momento. Foi definido dessa forma para alcançar um público mais abrangente, pois como a blockchain e o Solidity são tecnologias relativamente novas, buscar desenvolvedores que conheçam todas elas seria complicado e demorado. Por isso, foi preferível realizar uma introdução dos conceitos necessários para todos os usuários, de forma que todos eles conseguiram completar o experimento conhecendo os conceitos essenciais da blockchain e Solidity. Experiência em TypeScript não foi necessário porque o experimento foi criado de forma que não precisasse de nenhuma tipagem explícita, ou seja, a experiência em JavaScript era o suficiente.
Desses participantes, 1 pessoa tem experiência em JavaScript de 8 anos, 4 pessoas de 5 anos, 11 pessoas de 1 a 4 anos e 6 pessoas usam a linguagem menos de 1 ano. Com relação ao Solidity, 1 pessoa tem experiência de 4 anos, 1 possui experiência de 3 anos, 4 participantes adquiriram conhecimento básico de Solidity em uma matéria da universidade, e informaram que conhecem a menos de 1 ano, as outras 16 pessoas não tinham

[6] O arquivo de configuração do Gifflar pode conter dados de várias redes blockchain, cada uma com um identificador que foi utilizado pelos participantes para selecionar a rede blockchain correta

[7] https://testnet.binance.org/faucet-smart

Tabela 4.1 Estatísticas de conhecimento dos participantes.

	Nenhum		**Básico**		**Intermediário**		**Avançado**	
	#[8]	%	#	%	#	%	#	%
JavaScript	0	0%	9	41%	7	32%	6	27%
Blockchain	6	27%	13	59%	2	9%	1	5%
Solidity	16	73%	4	18%	1	5%	1	5%

Nota: Porcentagens arredondadas para uma melhor visualização.

nenhuma experiência na linguagem.

A Tabela 4.1 mostra algumas estatísticas em relação ao conhecimento dos participantes em JavaScript, Blockchain (conhecimento teórico) e Solidity. Considerando os dados apresentados anteriormente e os dados da tabela, percebe-se que a variação dos participantes com relação à experiência na linguagem JavaScript foi bem dividida, sendo que a maioria considera ter um conhecimento básico. O mesmo não pode ser afirmado para as competências "Blockchain" e "Solidity", o que já era esperado, considerando que estas são tecnologias ainda novas que ainda estão conquistando os estudantes e profissionais.

4.1.3.4 Protocolo

Cada chamada de vídeo foi realizada na plataforma Meet do Google. O protocolo inicia com o condutor realizando uma introdução de conceitos básicos sobre blockchain, SmCs e o objetivo do Gifflar. Ainda nesta introdução, o condutor mostra um exemplo de código Solidity (Apêndice B) e explica alguns conceitos básicos da linguagem essenciais para entender alguns recursos da biblioteca. Este processo durou em torno de 20 minutos, em alguns casos em que o participante teve dúvidas demorou mais tempo, porém o foco desta etapa era o entendimento do participante sobre tais conceitos para então prosseguir com as próximas etapas.

Após a introdução básica, o condutor explica que criou *tasks* para guiar o teste de usabilidade e também fala sobre o protocolo *Thinking aloud*, solicitando para que o participante possa proferir em voz audível suas intenções na resolução de cada *task*. O condutor então pergunta ao participante sobre a possibilidade de gravar a chamada e prossegue informando que este não deve preocupar-se em resolver as *tasks* em tempo hábil, pois o foco do teste é a biblioteca Solgen e não as habilidades do próprio participante. O participante também recebe a informação de que ao final do teste o condutor irá realizar algumas perguntas sobre seu ponto de vista em relação à usabilidade do Gifflar.

Nesta próxima etapa, o condutor compartilha com o participante o link de acesso ao *workspace* do GitPod e em seguida explica a organização de diretórios e arquivos do Gifflar e o que cada um representa. O participante neste momento é instruído que terá acesso à documentação de cada recurso da biblioteca dentro do próprio código e também é solicitado para compartilhar a tela do navegador onde se encontra o *workspace* do GitPod. Uma vez que o ambiente de desenvolvimento do participante está pronto, o

[8]'#' é utilizado nas tabelas para referir-se a 'quantidade'.

condutor compartilha o link de onde as *tasks* estão definidas. A fim de melhor preparar o participante, o condutor explica como as *tasks* estão organizadas e também esclarece de forma geral o propósito de cada *task*. O condutor então inicia a gravação e libera o participante para iniciar da primeira *task*.

Durante o experimento, o condutor manteve o microfone desligado para não atrapalhar o participante e somente em alguns casos em que o participante levantava uma dúvida sobre a *task* ou algum recurso da biblioteca o condutor se manifestava. Todas as dúvidas sobre as *tasks* foram tiradas de forma clara e direta, mas qualquer dúvida que surgiu sobre o uso dos recursos da biblioteca foram tiradas pelo condutor incentivando o próprio participante a encontrar na documentação, tendo o cuidado para não prejudicar a avaliação.

Depois que os participantes finalizam todas as *tasks* e realizam a implantação do contrato na rede acionando o comando *gifflar deploy*[9], o condutor explica brevemente como visualizar os dados do contrato no explorador de blocos da rede de teste da BSC[10] e prossegue para a entrevista. Após a entrevista, a gravação é encerrada e todo o procedimento é finalizado. Após rever as gravações, ainda assim foi possível ter uma média de quanto tempo os participantes levaram para finalizar as *tasks*. Em média, o tempo para finalizar as *tasks* foi de 56 minutos, sendo que o menor tempo foi 27 minutos e o maior tempo foi 109 minutos. O Apêndice C mostra um exemplo de código que representa o código final depois que os participantes finalizaram todas as *tasks*.

4.2 ESTRATÉGIA DE AVALIAÇÃO DE APLICABILIDADE

Como forma de explorar a aplicação do Gifflar em outros projetos, foi realizada uma avaliação mais conceitual. Para isso, conduziu-se uma revisão da literatura em busca de artigos que propõem projetos práticos e implementados que utilizam blockchain. Como a versão atual do Gifflar abrange somente a linguagem Solidity, a revisão foi mais direcionada para projetos que utilizam esta linguagem. Dessa forma, é possível definir duas questões de pesquisa específica para esta revisão, as quais foram nomeadas **QPR1** e **QPR2** a fim de diferenciar da nomenclatura das questões de pesquisa da avaliação de usabilidade.

- **QPR1**: Quais os projetos em blockchain onde o uso do Gifflar poderia beneficiá-los?
- **QPR2**: De forma conceitual, como o Gifflar pode ser aplicado nestes projetos?

4.2.1 Condução da Pesquisa

A revisão foi realizada no Google Acadêmico e foram utilizadas as palavras chave *"decentralized application ethereum"* e *"decentralized application solidity"*, com o objetivo de encontrar projetos que propuseram a implementação de aplicações descentralizadas e

[9]Transações podem ser observadas no endereço:
https://testnet.bscscan.com/address/0xc49d80472ffa30a9a7b1c7b137dd05ff528f4e1d
[10]https://testnet.bscscan.com/

que utilizam a Ethereum como blockchain ou que somente utilizam Solidity como linguagem de programação. A restrição de período foi de artidos entre 2018 e 2022, a fim de encontrar projetos mais recentes.

Para cada uma das duas buscas (uma busca para cada string de busca) foram considerados os 50 primeiros artigos, totalizando 100 artigos. Isto porque durante as duas buscas, as próximas páginas do Google Acadêmico apresentavam muitos artigos que fugiam do objetivo da revisão.

4.2.2 Triagem dos Artigos

A revisão contou com a definição de alguns critérios. Em primeiro lugar, o próprio fluxograma apresentado na Seção 3.4 do Capítulo 3 foi utilizado como critério de inclusão dos artigos. Por outro lado, houve a necessidade de definir os seguintes critérios de exclusão:

- Foram desconsiderados *whitepapers*[11] de projetos ou plataformas blockchain. Por ser um documento mais voltado para marketing.
- Livros ou slides. Pois geralmente só apresentam exemplos de projetos geralmente mais genéricos.
- Artigos duplicados.
- Artigos em idiomas diferentes do inglês.

4.2.3 Etapas da Pesquisa

O processo de desenvolvimento da pesquisa em busca de responder as questões **QPR1** e **QPR2** seguiu as seguintes etapas:

- A primeira etapa consistiu em pesquisar os artigos na base de dados Google Acadêmico utilizando as *strings* de busca. Nesta etapa foram lidos os títulos de cada artigo e aplicados os critérios de exclusão, caso fosse necessário entender melhor o objetivo do trabalho, o resumo também era explorado. Os artigos com títulos e resumos candidatos foram selecionados para a próxima etapa.
- Depois de coletar os artigos candidatos, foi realizada uma leitura completa a fim de aplicar o fluxograma de uso do Gifflar e verificar se o Gifflar pode ou não ser aplicado. É uma etapa diretamente relacionada com a questão de pesquisa **QPR1**.
- Na última etapa já foram selecionados os projetos nos quais certamente podem beneficiar-se o Gifflar. A fim de responder a questão **QPR2**, para cada artigo é realizada uma releitura completa com o objetivo de entender o projeto de forma mais técnica, encontrar e descrever um cenário em que o uso do Gifflar contribuiria de forma significativa ao projeto.

[11] Diferente de um artigo científico, o whitepaper é mais um documento informativo das características de um certo produto. É muito utilizado por projetos da área da Blockchain.

Como informado na Seção 3.4 do Capítulo 3, o Gifflar surgiu também como uma contribuição para o projeto IoTCocoa, por isso, foi definido incluir o IoTCocoa como um dos projetos eleitos a uma descrição mais detalhada de um cenário de integração com o Gifflar.

Capítulo 5

RESULTADOS DAS AVALIAÇÕES

Este capítulo apresenta e discute os resultados obtidos na avaliação de usabilidade e de aplicabilidade do Gifflar. Com relação aos resultados da primeira avaliação, são reveladas as respostas de cada participante a respeito das perguntas da entrevista, além de listar quais os tokens de usabilidade detectados de acordo com as reações de cada participante. Estes dois resultados são confrontados gerando uma discussão sobre as falhas de usabilidade da biblioteca Solgen que serão corrigidas nas próximas atualizações do *framework*. Também são evidenciadas as possíveis ameaças à validade da avaliação e o que foi realizado para diminuir seus impactos. Os resultados da segunda avaliação expõem a quantidade de projetos encontrados e quais foram selecionados para uma possível aplicação do Gifflar. Ao fim do capítulo, são apresentados dois projetos selecionados demonstrando em detalhes como o Gifflar poderia ser utilizado e em que sentido o projeto seria aprimorado.

5.1 AVALIAÇÃO DE USABILIDADE

Os resultados da avaliação são apresentados nas próximas subseções na sequência das questões de pesquisa. Cada subseção inicia com um parágrafo resumindo os resultados relacionados à devida questão de pesquisa e em seguida apresenta os dados quantitativos da entrevista e tokens de usabilidade, os quais estão resumidos respectivamente nas Tabelas 5.1 e 5.2. A duas tabelas apresentam as porcentagens formatadas sem casas decimais utilizando arredondamento para uma melhor visualização dos dados, porém, no Apêndice D as mesmas tabelas são apresentadas de forma mais completa e considerando as porcentagens mais precisas em duas casas decimais.

A Tabela 5.1 organiza as respostas de cada participante em “Sim”, “Não” e “Algumas vezes”. Estas respostas são relacionadas às doze perguntas sobre o Gifflar, e são apresentados na tabela a quantidade absoluta de participantes que responderam de acordo com aquela categoria seguido da porcentagem que esta quantidade representa em relação ao total de participantes. A questão 18 (sobre recomendações de melhoria) não foi inserida nesta tabela por ser mais subjetiva, por isso, ela será melhor abordada no conteúdo de

Tabela 5.1 Respostas das perguntas da entrevista.

Q[1]	QP[2]	Sim		Não		Algumas vezes	
		#	%	#	%	#	%
6	1	15	68%	0	0%	7	32%
7	1	10	46%	4	18%	8	36%
8	1	21	96%	1	5%	0	0%
9	2	22	100%	0	0%	0	0%
10	2	0	0%	22	100%	0	0%
11	2	0	0%	21	96%	1	5%
12	3	0	0%	19	86%	3	14%
13	3	22	100%	0	0%	0	0%
14	3	1	5%	16	73%	5	23%
15	3	0	0%	22	100%	0	0%
16	4	21	96%	0	0%	1	5%
17	4	0	0%	22	100%	0	0%

Nota: Porcentagens arredondadas para uma melhor visualização.

cada seção. Após verificar as gravações do experimento com cada participante, observou-se que não houve nenhuma ambiguidade ao responder as questões, somente foram detectadas respostas mais detalhadas em algumas questões, as quais serão mencionadas no decorrer da leitura.

A Tabela 5.2 apresenta os tokens de usabilidade detectados ao rever as gravações do teste realizado pelos participantes. Para cada token é apresentado um identificador para melhor referir-se ao token durante a discussão, uma descrição rápida para entender do que se trata a ocorrência do evento que o token representa, também a relação deste token com as questões de pesquisa da Seção 4.1.1 do Capítulo 4, a quantidade absoluta de participantes que reproduziram este evento e a porcentagem que essa quantidade representa em realação à quantidade total de participantes. Esta tabela apresenta somente os tokens mais frequentes, a tabela completa com todos os 27 tokens detectados pode ser encontrada no Apêndice D.

5.1.1 QP1: Compreensibilidade

O esforço necessário para compreender os recursos da biblioteca é mínimo, a maioria dos participantes consideraram que os recursos foram bem definidos e a documentação estava bem explicativa. Mesmo assim, existem algumas dificuldades enfrentadas por determinados desenvolvedores que resultarão em futuros aperfeiçoamentos.

Quase todos os participantes (96%) concordaram que o código necessário para completar as *tasks* condiz com as expectativas (questão 8). Isso significa que o código cliente criado ao utilizar o Gifflar pode ser considerado um código compreendido com facilidade

[2]'Q' é utilizado nas tabelas para referir-se a 'questão'.

[3]'QP' é utilizado nas tabelas para referir-se a 'questão de pesquisa'.

Tabela 5.2 Tokens de usabilidade mais frequentes

Q	Token[4]	Descrição	QPs	#	%
T1	S	não acostumado a esse tipo de doc.	1	17	77%
T2	S	tipo de dado incomum.	1, 2	13	59%
T3	N	não encontrado sobre 'memory'.	1, 2	12	55%
T4	R	'dataLocation' ao criar variável local não entendido.	1, 2	8	36%
T5	S	esperado 'setEmit' não 'setEventCall'.	1	8	36%
T6	R	exemplo do padrão builder não está claro.	1, 2	7	32%
T7	C	definir 'inputs' por argumento ou método 'setInput'.	1, 3	5	23%
T8	S	fábrica é um objeto?	1, 2	4	18%
T9	N	método 'deploy' não encontrado.	1	4	18%
T10	S	criar a variável pelo TypeScript.	1, 2	4	18%
T11	I	criar variável local nos *inputs* de 'createFunction'.	1, 2, 4	4	18%
T12	I	tentou criar variável definindo somente o nome.	1	4	18%
T13	U	atribuição depois de criação de variável.	1, 4	3	14%
T14	S	achou que 'createVariable' cria variável local.	1	3	14%
T15	R	difícil encontrar método 'deployed'.	1	3	14%
T16	I	tentou criar o construtor pelo TypeScript.	1, 2	3	14%
T17	N	método 'deployed' não encontrado.	1	3	14%

Nota: Porcentagens arredondadas para uma melhor visualização.

pelos desenvolvedores, de acordo com este resultado. Somente um usuário informou que o código criado utilizando a biblioteca não condiz com as expectativas, pois escreveu muito mais do que se estivesse escrito em Solidity e, portanto, esperava um código mais claro e limpo (isto também fere a QP3, inclusive este ponto é retomado com mais detalhes na seção relacionada a esta questão de pesquisa).

Quase 70% dos participantes consideram que os nomes e a forma de acesso dos recursos condizem com os conceitos do domínio (questão 6). Porém, 32% declararam que isto é válido somente algumas vezes. Alguns tokens refletem essa declaração; no token T5, 36% dos desenvolvedores ao final da *task* 2, na instrução de emissão de evento, procuraram o método que gera a emissão pelo nome "*setEmit()*", quando na verdade era chamado "*setEventCall()*"; 14% tentaram acionar o método "*createVariable()*" da biblioteca para criar uma variável local de uma função do Solidity (token T14), sendo que este método cria variáveis de estado e o "*setVariable()*" cria variáveis locais; os tokens T9 e T17 destacam eventos em que, respectivamente, 18% dos programadores não encontraram o método "*deploy()*" e 14% não encontraram o método "*deployed()*", precisaram ser melhor instruídos pelo condutor. Porém, estes são usuários que não tinham quase nenhuma experiência em programação de SmCs, observou-se que os experientes na linguagem Solidity encontraram estes métodos com facilidade justamente por lidar com tais nomenclaturas de funcionalidades com mais frequência.

Dos participantes, 46% precisaram buscar informações com frequência na documentação para entender cada recurso da Solgen (questão 7); 36% comunicaram que algumas vezes precisaram consultar a documentação de forma mais frequente; e 18% informaram que aprenderam os recursos mais rapidamente, pois segue os paradigmas de programação

[4]Cada classificação de token da Seção 4.1.2.2 do Capítulo 4 é representada pela inicial maiúscula.

semelhante às ferramentas que utilizam e, portanto, não precisaram constantemente consultar a documentação. Alguns tokens refletem este resultado. O token T15 diz respeito à dificuldade que 14% dos participantes tiveram de encontrar o método "*deployed()*", responsável por verificar se o SmC foi implantado na rede e retornar a instância do mesmo para permitir acionar suas funções (diferente do token T17, não houve a necessidade do condutor intervir, o participante conseguiu encontrar o método, porém com mais dificuldade do que os outros que encontraram rapidamente); o token T12 evidencia uma falha de alguns participantes (14 %) ao tentar utilizar o método da biblioteca para criar variável de estado e passar somente o nome da variável no parâmetro, mesmo depois de olhar a documentação do método. Estes dois tokens são resultados do evento que o token T1 representa. Grande parte dos participantes (77%) não estavam acostumados a olhar a documentação pelo próprio código, eles informaram que normalmente utilizavam documentações disponibilizadas pelo site da biblioteca ou alguma página na qual é possível observar os recursos de forma mais organizada e clara. Em alguns momentos também foi observado que o participante não lia direito a documentação, com o intuito de terminar as *tasks* em tempo hábil, mesmo informado que tinha o tempo necessário para finalizar.

Por fim, a questão dezoito, referente a recomendações de melhoria da biblioteca, gerou algumas respostas que surgiram das dificuldades apresentadas nos parágrafos anteriores. Alguns participantes (14%) que sentiram dificuldades em encontrar o método "*setEventCall()*" recomendaram alterar o nome para algo semelhante a "*setEmit()*" ou "*setEmitEvent()*". Outros (9%) recomendaram alterar o nome do método "*deployed()*" para algo semelhante a "*getInstance()*". Sobre a documentação, no geral, todos elogiaram a forma como está explicando e exemplificando o uso dos recursos, porém 14% recomendaram criar uma página web com uma documentação mais completa, além disso, um participante recomendou melhorar a documentação sobre os métodos aninhados para criação de conteúdo de funções (token T6).

5.1.2 QP2: Abstração

De acordo com os resultados, o nível de abstração da biblioteca está adequado e não houve nenhuma necessidade de adaptação da API. Porém, existem algumas abstrações para serem revisitadas, de modo a verificar se existe uma forma mais clara de implementação do recurso devido à algumas confusões que surgiram dos participantes durante a execução das *tasks*.

Todos os participantes informaram que o nível de abstração da biblioteca é apropriado para resolver as atividades (questão 9) e que também não precisaram adaptar a API para estar de acordo com suas necessidades (questão 10). Porém, foram observados alguns tokens que vão contra este resultado. O token T2 representa um evento que aconteceu com mais da metade dos participantes (59%), os quais consideraram que a abstração das tipagens do Solidity ficou um pouco incomum. Ao criar uma variável de estado no Solidity pela biblioteca, por exemplo, o primeiro parâmetro do método responsável por esta criação é um objeto que define a tipagem da variável, dentro deste objeto é escolhida a propriedade desejada para esta tipagem: "*regularType*", para tipos regulares como "*string*"; "*array*", para representar a criação de um *array*; ou "*customType*", para inserir

um tipo personalizado, como por exemplo, um *struct* criado. A primeira intenção do participante após ler o primeiro item da *task* 2 é tentar criar a variável pelo TypeScript (token T10), ou acionar o método correto e inserir no primeiro parâmetro uma tipagem como no TypeScript ou como *string*:

```
- string message;
- createVariable(string, ...)
- createVariable(‘‘string’’, ...)
```

mas ao olhar a documentação, verifica que na verdade é como um objeto:

```
- createVariable({ regularType:  ‘‘string’’ }, ...)
```

por isso surge esta surpresa no participante, inclusive, 18% dos desenvolvedores recomendaram rever esta abstração da tipagem justamente por esta dificuldade. A mesma intenção acontece com 14% dos participantes ao tentarem criar o construtor do Solidity pelo próprio TypeScript ao invés de utilizar um método da biblioteca (token T16).

Uma outra dificuldade foi a surpresa inicial no uso da fábrica "*createGifflarContract()*", que recebe como parâmetro o nome do contrato e retorna o objeto de modelagem para construção do contrato através deste objeto. 18% dos participantes acharam que esta fábrica seria um objeto no qual deveriam selecionar uma propriedade ou método para criar um contrato. Isto pode ter acontecido por falta de clareza da *task* ou atenção do participante, o qual ao entender melhor a documentação conseguiu corrigir o erro.

A maioria dos participantes (96%) não precisaram entender a implementação interna da API da biblioteca para utilizá-la (questão 11), tudo foi encontrado dentro da documentação de cada método. Somente um participante buscou a implementação interna das tipagens, mas o mesmo explicou que é algo que usualmente faz com outras bibliotecas também, ou seja, o próprio participante já é acostumado a buscar mais informações na implementação interna dos recursos, portanto, não influencia negativamente na abstração oferecida pela interface, uma vez que nenhum outro participante fez o mesmo.

Alguns participantes demonstraram dificuldade em um recurso da criação de variável local com a biblioteca. A criação de variável local, para determinados tipos de dado no Solidity, necessita definir se a variável representa um valor localizado na memória ou no armazenamento fixo do contrato, por isso, nestes casos é necessário utilizar o termo "*memory*" ou "*storage*". No método de criação de variável local, esta localização de dado é abstraída inserindo a propriedade "*dataLocation*" como uma opção no método de criação de variável local. Com base nisto, 55% dos participantes não encontraram como definir a abstração "*dataLocation*" mesmo lendo a documentação (pode-se considerar também falta de atenção do participante, como comentado antes, foi observado em alguns deles a intenção em completar as *tasks* de forma rápida); já 36% deles encontraram o "*dataLocation*", porém não entenderam como utilizar. Esta funcionalidade em específico estava descrita, porém, não estava nos exemplos fornecidos pela documentação propositalmente, como uma maneira de estimular os participantes a pensar da forma como a linguagem Solidity está abstraída sem o apoio do exemplo de uso.

5.1.3 QP3: Reusabilidade

Os participantes concordam que conseguiram escrever um código claro, conciso e de forma incremental. Também consideram que a alteração do tipo de rede blockchain utilizada não é complexa. Muitos participantes declararam que o uso da biblioteca foi bem direcionado, enquanto uma outra parte deles encontrou alternativas de implementação as quais algumas vezes geraram ambiguidade, originando alguns eventos representados pelos tokens de usabilidade.

Todos os participantes assumiram que foi fácil de avaliar seu próprio progresso na resolução das *tasks* (questão 13). Foi percebido também nas gravações que, mesmo surgindo os eventos representados pelos tokens, o progresso foi gradual e que alguns recursos utilizados em *tasks* anteriores foram necessários para as próximas, assim como o próprio conhecimento em si.

Nenhum usuário considerou difícil alterar o código para utilizar outro tipo de rede blockchain (questão 15), justamente porque é somente alterar o arquivo de configuração e realizar a seleção pelo identificador da rede. Neste sentido, um dos participantes recomendou na questão 18 permitir alterar a rede blockchain também pelo método "*deploy()*", para caso o sistema lide com várias redes em tempo de execução. A biblioteca já possui um método para definir a configuração de rede pelo código, mesmo assim, é uma recomendação pertinente e que será considerada nas próximas modificações da biblioteca.

Quase 90% dos participantes afirmaram que a quantidade de código não está exagerada nem insuficiente (questão 12). Entretanto, 14% acreditam que o código é um pouco exagerado, e que escrevendo o mesmo código em Solidity ficaria bem menor. Por outro lado, um dos participantes justificou sua resposta como "Não" porque considerou que se o objetivo do *framework* é gerar SmCs de forma dinâmica, a quantidade de código está adequada, uma vez que se fosse desenvolver este artifício manualmente, a quantidade seria ainda maior do que a quantidade do código feito com a Solgen.

Acima de 70% dos participantes não perceberam nenhuma alternativa de uso de algum recurso (questão 14), informaram que a documentação da biblioteca foi bem direta ao ponto. Porém, na verdade existem sim algumas alternativas, as quais foram percebidas por 27% dos participantes (considerando os que responderam "Sim" e "Algumas vezes"). E este fator é confirmado pelo token T7, onde 23% dos desenvolvedores se depararam com a possibilidade de definir os *inputs* de uma função do Solidity direto no argumento do método "*createFunction()*" como uma lista ou acionando um método interno "*setInput()*" para definir cada *input* separadamente. Alguns tokens refletem a dificuldade de alguns participantes em entender esta alternativa, como o T11, e outros representados na tabela completa de tokens no Apêndice D, como o T24, T18, T26 e T27.

Algumas recomendações de melhoria relacionam-se com esta questão de pesquisa. Dois dos participantes recomendaram melhorar a explicação sobre o "*setInput()*" na documentação. Um outro participante sugeriu a criação de um método que escreve o código Solidity e compila em um mesmo momento (na biblioteca Solgen existem os métodos *write* e *compile* separados voltados para estas ações), como uma forma de reutilizar e reduzir as duas funcionalidades em uma só, uma vez que a compilação não acontece se não houver código-fonte gerado; ele também fez uma comparação com os métodos da

biblioteca Scikit-Learn, uma biblioteca da linguagem Python voltada para aprendizado de máquina, onde existem os métodos "*fit()*" para treinar o modelo, o método "*transform()*" para escalar o treinamento com mais dados e existe o método "*fit_transform()*" que reutiliza os dois métodos e resume as duas funcionalidades em uma só.

5.1.4 QP4: Aprendizagem

Nenhum participante precisou entender as relações entre os componentes da biblioteca nem conhecer as classes internas. O aprendizado dos participantes ao executar as *tasks* foi gradual, de modo que o conhecimento adquirido em uma *task* foi reaproveitado nas próximas. Ainda assim, foram observados alguns eventos durante os testes que, em parte, comprometem esta afirmação.

Todos os participantes informaram que não precisaram entender nenhuma relação entre os componentes nem conhecer as classes internas para resolver as *tasks* (questão 17), somente a interface da biblioteca foi o suficiente. Porém, dentro da interface foi necessário o entendimento de algumas relações. Um exemplo é a última *subtask* da *task* 2, que consiste na criação de uma função do Solidity pela biblioteca. Nesta *subtask*, compreender que o conteúdo da função é definido através de métodos aninhados é essencial (padrão BUILDER), inclusive entender que os métodos subsequentes do método de criação de função ("*createFunction()*") são diferentes dos métodos gerais acionados pelo objeto do GIFFLARCONTRACTMODEL. Este conhecimento citado é inicialmente estimulado na *subtask* 3 da *task* 2, a qual solicita ao participante para criar um construtor do Solidity utilizando a biblioteca e inserir uma atribuição como conteúdo do construtor. A documentação propositalmente não apresentou informações detalhadas sobre os métodos aninhados, justamente para verificar se seria uma funcionalidade de fácil aprendizado, ou se haveria a necessidade de maiores explicações, que foi o caso. Inclusive esta dificuldade dos participantes gerou um token de usabilidade T6, que refere-se à falta de clareza na explicação de como o padrão BUILDER é utilizado na Solgen.

Quase 100% dos participantes concordam que depois de finalizar as primeiras duas *tasks*, foi menos complexo completar as outras (questão 16). Todos os que responderam "Sim" a esta pergunta, afirmaram que o conhecimento adquirido em uma *task* foi reutilizado nas próximas, de modo que o aprendizado foi sendo gradual. O desenvolvedor que respondeu "Algumas vezes" também declarou o mesmo que os outros, porém, percebeu que a partir da *task* 3, a perspectiva muda para uma visão que abstrai a modelagem realizada a fim de acionar os métodos que irão escrever o código a partir da modelagem criada, compilá-lo e implantá-lo da rede blockchain. Esta foi uma das pessoas que não tem experiência em programação de SmC e, de fato, para pessoas que não são da área específica, esta forma de programação é um pouco diferente, inclusive os nomes dos métodos como o "*deploy()*" e o "*deployed()*", assim como seus argumentos, são incomuns para eles. Apesar disso, alguns participantes apresentaram comportamentos que evidenciaram uma interrupção no progresso do conhecimento adquirido. Depois de, na *task* 2, passar pela *subtask* de criação de um construtor e seguir para a última *subtask* a fim de criar uma função no Solidity, 18% dos participantes tentaram realizar uma atribuição interna da função nos argumentos do método "*createFunction()*" responsáveis por definir os *inputs*

da função (token T11), mesmo depois de ter utilizado o método "*setAssignment()*" para este fim no conteúdo do construtor. Outro evento observado é representado pelo token T13, onde o participante deveria criar uma variável local na última *subtask* e inicializá-la com um valor utilizando o método "*setVariable()*", porém, 14% destes participantes resolveram esta *subtask* criando a variável local sem a inicialização pelo método "*setVariable()*" e posteriormente utilizou o método "*setAssignment()*" para atribuir o valor inicial da variável na próxima linha (token T13), o que não está errado, porém, foi um evento inesperado pelo condutor do teste de avaliação.

5.2 DISCUSSÃO

De um modo geral, apesar das inconsistências encontradas, a Solgen teve bons resultados no teste de usabilidade realizado. Além disso, todas as inconsistências encontradas serão essenciais para a evolução do Gifflar. Por isso, esta seção retoma alguns prontos principais de cada tópico das questões de pesquisa (compreensibilidade, abstração, reusabilidade e aprendizagem) e traça um plano preliminar de atualizações na Solgen que objetivam diminuir as principais dificuldades encontradas pelos participantes ao utilizar a biblioteca.

No tópico da compreensibilidade, um dos pontos principais é o nome do método "*setEventCall()*". Sem dúvida, o nome deste método acabou fugindo do padrão dos nomes dos outros métodos. A intenção é que a nomenclatura defina de forma mais clara possível a funcionalidade daquele método. Como uma quantidade considerável de participantes apresentaram esta dificuldade e ainda recomendaram a alteração do nome na última questão da entrevista, o nome deste método será então revisado e alterado para um nome que coincide com a funcionalidade de emissão de evento no Solidity. Outros dois métodos que tiveram seus nomes questionados foram o "*deploy()*" e "*deployed()*". Nos dois casos, os poucos participantes que tiveram dificuldades em encontrar os métodos não tinham experiência em programação de SmCs; já a maioria ou tinha experiência na área, ou leu a documentação com mais atenção. Além disso, o método "*deploy()*" descreve de forma clara a ação que será realizada (traduzindo para o português: "implantar"); o "*deployed()*" pode não estar diretamente claro, mas considerando que a instância pode somente ser obtida depois que o *deploy()* for realizado, este seria um método candidato a ter sua descrição da documentação lida. Uma opção para o "*deployed()*" seria a criação de um outro método chamado "*getInstance()*" que faria a mesma ação que o "*deployed()*" já conhecido pelos desenvolvedores da área, porém, isso geraria uma ambiguidade no uso da biblioteca. Portanto, o nome destes dois métodos serão mantidos. Um outro fator citado foi sobre melhorias na documentação. Todas as explicações da documentação serão revisadas e aperfeiçoadas para estar ainda mais compreensível, além disso, uma documentação no formato de página Web já está sendo produzida.

Um dos eventos mais frequentes relacionado à abstração da biblioteca, foi a forma como as tipagens dos recursos do Solidity foram abstraídas. Realmente não é confortável abrir uma chave de objeto do JavaScript todas as vezes que, por exemplo, for criar uma simples variável. Esta foi uma forma de permitir o usuário escolher o modelo de tipagem que deseja utilizar (tipo elementar, *array* ou um tipo personalizado), e também de possibilitar selecionar o tipo elementar sem precisar escrever, o que ainda evita erros de

digitação. Definir a tipagem diretamente como uma *string* não possibilita esta seleção do tipo elementar diretamente no argumento da função, seria somente um modo de tipagem: a tipagem personalizada sem nenhuma apoio da biblioteca. Mesmo assim, foi observado que esta seleção não foi utilizada por quase nenhum desenvolvedor. Por este motivo e por ter sido um evento bem frequente no teste de usabilidade, a tipagem dos recursos da biblioteca será reavaliada. Uma ideia é criar uma utilidade na biblioteca que exporta essa tipagem com a seleção, assim, o desenvolvedor pode escolher se quer utilizar ou não este modo de tipagem para garantir a escrita correta. A desvantagem desta ideia é que acaba gerando uma ambiguidade, mas mesmo assim, não seria uma ambiguidade grave, uma vez que não seria um recurso dentro do método, mas sim um recurso que precisaria ser importado para utilizar. A criação do objeto de contrato através da fábrica "*createGifflarContract()*" também gerou uma certa dificuldade em poucos participantes, esta não foi considerada tão preocupante pois também foi influenciada pelo fato de que grande parte dos participantes não estavam acostumados em consultar a documentação pelo próprio código. Já o caso da definição do "*dataLocation*" na criação de variável, a qual não teve um código de exemplo inserido na documentação propositalmente, agora será melhor descrita na documentação.

A alternativa de inserir os *inputs* de uma função do Solidity pelo argumento do método "*createFunction()*" ou pelo método interno "*setInput()*" gerou algumas confusões em poucos participantes. Além disso, outros tokens de usabilidade relacionaram-se com o não entendimento do objetivo destes recursos. Porém, foi observado durante o teste de cada um destes participantes que faltou atenção à documentação, pois está claro em um trecho da documentação uma frase que descreve para que serve o "*setInput()*" e deixa claro que é uma alternativa de definir os inputs da função. Existem três fatores que podem ter influenciado esta falta de atenção. Primeiro é que estes participantes não tinham costume em utilizar a documentação pelo próprio código. Segundo é que a documentação estava toda em inglês, e foi notado que alguns desenvolvedores não conheceram alguns termos. Para tentar resolver este segundo fator, que inclusive já era previsto, o condutor do teste deixou claro a cada participante que poderia pedir ajuda caso precisasse traduzir algum trecho ou termo do inglês para o português. Porém, mesmo assim, existe um outro fator, o qual será melhor abordado na próxima seção, em que algumas pessoas acabam hesitando em pedir auxílio ou realizar perguntas ao condutor, que a princípio é uma pessoa desconhecida, e sobretudo porque o teste de avaliação realizado lembra uma entrevista de emprego, onde o desconforto é bem maior e o programador deve ser capaz de desenvolver um código com propriedade e no final será julgado se está apto para a vaga ou não. Por estes motivos, a alternativa do "*setInput()*" será mantida, até porque os participantes que detectaram este método e prestaram atenção à documentação, conseguiram entender e até escolher qual das duas alternativas sentia-se melhor em utilizar. Existem duas recomendações na Seção 5.1.3 que serão consideradas nas próximas atualizações do Gifflar, são a inclusão de um argumento para definir a rede blockchain no método "*deploy()*" e adicionar um método na biblioteca que realiza as duas ações de escrever e compilar o código Solidity.

No tópico de aprendizagem, surgiram três eventos importantes que devem ser analisados e evitados nas próximas atualizações do Gifflar. O primeiro deles é a dificuldade que

poucos participantes tiveram em entender o padrão BUILDER. De fato, na documentação não está totalmente claro que os métodos aninhados na criação de uma função são diferentes dos métodos do objeto GIFFLARCONTRACTMODEL, nem que cada método destes representa uma instrução interna da função, somente apresenta um exemplo de como utilizar os métodos aninhados e informa que é assim que define um conteúdo de uma função. Nas próximas atualizações do Gifflar, essa parte da documentação da biblioteca será mais explicativa. Esta melhoria na documentação irá também contribuir na diminuição da ocorrência dos outros dois eventos, onde alguns participantes tentaram realizar uma atribuição interna da função nos argumentos do método "*createFunction()*" e outros participantes que definiram o valor inicial da variável local inserindo uma nova instrução de atribuição quando poderia ter inicializado na mesma instrução da criação da variável.

Adicionalmente, 18% dos participantes obtiveram erros de compilação de código do SmC ao executar o comando de implantação. E outros 18% também obtiveram erros, porém após a implantação. Quase todos foram por causa de parâmetros que estavam diferentes do que a task pedia, por exemplo, nome de variável ou nome de função. Somente um erro de implantação foi por causa de instabilidades na rede de teste da BSC (discutido com mais detalhes na próxima seção). Os erros de compilação foram apresentados de forma amigável, porém mostra os erros relacionados ao código Solidity e não ao modelo do criado com o Gifflar. Já os erros após a implantação não foram apresentados de forma amigável. Por isso, o tratamento e apresentação dos erros do Gifflar também precisam ser aperfeiçoados.

Em síntese, estas são as principais alterações a serem realizadas na Solgen:

- Alterar o nome do método "*setEventCall()*" para um nome que corresponde de forma mais clara à funcionalidade representada.
- Disponibilizar a documentação em formato de *website*.
- Reavaliar e simplificar o formato da tipagem dos recursos.
- Avaliar se a a adição da definição o tipo de rede blockchain no método "*deploy()*" mantém a consistência do método.
- Adicionar um método que escreve e compila o código Solidity do SmC.
- Melhorar o tratamento e apresentação de erros.
- Revisar e aprimorar a documentação em todos os sentidos.

As alterações não se limitarão a estas, até porque existem outros tokens na tabela do Apêndice D que também serão revisados e podem fazer surgir outras atualizações.

5.3 AMEAÇAS À VALIDADE

Esta seção discute as principais ameaças à validade do teste de usabilidade do Gifflar e apresenta as ações realizadas para minimizar seus impactos.

5.3.1 Validade de Construção

O teste da usabilidade de um artefato de software possibilita obter dados variados e bastante subjetivos. Uma possível falha seria apresentar estes dados de manteira muito objetiva. Para evitar esta falha, foram utilizadas abordagens já validadas pela literatura que foram essenciais para guiar a avaliação da forma mais correta possível. A metodologia de Piccioni, Furia e Meyer (2013) foi fundamental para dirigir a avaliação e interpretar os resultados subjetivos encontrados de forma adequada. Esta metodologia também envolve outras abordagens que também foram importantes, como o CDF proposto por Blackwell et al. (2001) e o *Thinking aloud protocol* criado por Ericsson e Simon (1993).

Dois dos participantes do teste de usabilidade indagaram que os trechos de comentário "Defina aqui o método de escrita" e "Defina aqui o método de compilação" no código do *script* na *task* 4 confundiu o entendimento sobre o que era para ser feito, pois entendeu a palavra método pelo significado de "técnica", "procedimento" ou "mecanismo", e não de método de código, o que pode também ter acontecido com outros participantes até mesmo em outras situações em que demoraram de encontrar o método mas não verbalizaram o motivo. No geral, não houve nenhuma dificuldade em encontrar os métodos de escrita nem de compilação do código Solidity, por isso não precisou lidar com esta ameaça neste caso, porém, se o mesmo aconteceu em outras situações em que o condutor não teve retorno do participante que esse era o motivo, neste caso, não foi possível controlar.

O editor de códigos do GitPod, por mais que seja semelhante ao editor VSCode que muitos desenvolvedores utilizam, pode ter causado algum desconforto nos participantes por não estar utilizando seu editor de códigos usual. O VSCode, por exemplo, permite que o desenvolvedor instale *plugins* que oferecem algumas facilidades enquanto estiver programando. O GitPod também permite a instalação de *plugins*, porém nenhum foi instalado, pois instalar facilidades para os participantes do teste poderia influenciar negativamente nos resultados da avaliação, assim como permitir que os participantes realizassem o teste em seus editores de código que geralmente utilizam na máquina local. Mesmo assim, o editor do GitPod não foi um empecilho para nenhum dos participantes na resolução das *tasks*.

Somente uma participante do estudo não conseguiu implantar o contrato modelado na rede. Isto porque a rede de teste da BSC naquele momento estava instável, então a Solgen estava emitindo um erro que indicava *timeout* (tempo de espera esgotado), ou seja, o contrato estava demorando mais do que o normal para ser implantado na rede, a qual poderia estar com muitas requisições de transações pendentes para validar antes da transação solicitada. Porém, o comando de implantação era somente um *task* adicional do teste de usabilidade para permitir aos participantes terem uma experiência mais prática em utilizar o Gifflar. Por isso, é considerado que esta participante, ainda assim, conseguiu completar todas as *tasks* necessárias, uma vez que o contrato foi corretamente compilado e o método de "*deploy()*" foi corretamente invocado.

5.3.2 Validade Interna

Uma ameaça bem comum é a realização da avaliação com muitos participantes do ciclo de amizade do condutor ou que são até do mesmo grupo de pesquisa, o que pode influenciar

negativamente nos resultados da avaliação. Com o objetivo de diminuir os impactos desta ameaça, somente 7 pessoas foram do ciclo de amizade do condutor, os outros 15 foram pessoas desconhecidas. Nenhuma das pessoas tiveram contato direto com o Gifflar e somente 1 leu os artigos já publicados sobre o *framework*. Somente 1 das 7 pessoas faz parte do mesmo grupo de pesquisa do condutor da avaliação, que é o GAUDI (Grupo de Algoritmos e Computação Distribuída).

Grande parte dos participantes não estavam acostumados em utilizar a documentação no próprio código. Provavelmente se a documentação fosse disponibilizada em um *website*, os desenvolvedores teriam mais facilidade em encontrar os recursos da biblioteca. Porém, o motivo de disponibilizar a documentação somente no código foi para estimular a busca pelos nomes e descrições dos métodos a fim de verificar se os nomes dos recursos estavam intuitivos.

Os autores Piccioni, Furia e Meyer destacam a importância de ter exemplos de código na documentação, o que deixa a explicação ainda mais clara para o participante. Seguindo esta lição aprendida por eles, a documentação dos métodos da Solgen, além de apresentar a descrição do método e dos parâmetros, também incluiu trechos de código exemplificando como utilizar aquela funcionalidade. Porém, alguns participantes (23%) aproveitaram os exemplos para copiar e alterar de acordo com o que a *task* solicitava. Não houve impedimento para que eles fizessem isto, pois foi considerado que esta poderia ser uma forma de aprendizado destes desenvolvedores. Mesmo assim, esta ação pode ter influenciado os resultados da avaliação uma vez que alterar um código copiado evita mais erros do que escrever o código manualmente.

Existe também o fato de as pessoas sentirem algum tipo de estresse, nervosismo ou ansiedade em experimentar uma ferramenta que nunca utilizaram antes, principalmente quando este experimento está sendo observado por outra pessoa. A metodologia utilizada no teste de usabilidade é muito eficaz, entretanto traz consigo uma semelhança ao que gera muito estresse em desenvolvedores, que é a entrevista de emprego. De fato, o teste de usabilidade não visa nenhum julgamento ao participante, somente ao artefato sendo testado, mas o simples fato de ter um condutor do teste observando o participante completar as *tasks*, inconscientemente o programador assemelha a uma entrevista de emprego. Os autores Behroozi, Parnin e Barik (2019) explicam em seu estudo que o estresse pode causar efeitos disruptivos na memória, o que pode afetar negativamente a performance dos participantes, além disso, afirmam que uma das causas de ansiedade de um candidato em uma entrevista de emprego é o fato de estar sendo observado, até mesmo por alguém conhecido. Por este motivo, o condutor do teste informou a todos os participantes que eles não seriam julgados nem seriam cobrados para terminar as *tasks* em tempo hábil. Este foi um meio de tentar diminuir qualquer ansiedade ou estresse que podem ter surgido nos participantes no início do teste.

Piccioni, Furia e Meyer apontam sobre uma outra ameaça que é a possibilidade de ocorrer o que é chamado de "*interviewer effect*" (efeito entrevistador), onde o entrevistador, ao realizar as perguntas do questionário, acaba inconscientemente influenciando uma possível resposta que está esperando do participante. Uma outra ameaça é o condutor acabar respondendo dúvidas dos participantes sobre como utilizar uma certa funcionalidade da biblioteca. A primeira foi evitada seguindo estritamente as perguntas definidas

na Seção 4.1.2.1 do Capítulo 4. A segunda foi evitada com o condutor respondendo somente questões para clarificar o entendimento do que deve ser feito na *task*, também dúvidas sobre o editor do GitPod, sobre a blockchain utilizada e conceitos genéricos sobre o domínio onde o Gifflar está inserido. Além disso, todos os testes foram gravados, e após rever todas as gravações, foi certificado que somente essas dúvidas foram tiradas e que nas dúvidas que surgiram sobre como utilizar uma certa funcionalidade o condutor incentivou o participante a reexaminar a documentação.

5.3.3 Validade Externa

Como visto na Tabela 4.1 do Capítulo 4, a maioria dos participantes do estudo tinham conhecimento básico em Blockchain (13 dos 22 participantes) e muitos não tinham nenhum conhecimento em programação de SmCs (16 dos 22 participantes). Isso pode ser considerado uma ameaça aos resultados, uma vez que, caso houvessem desenvolvedores mais experientes nestas tecnologias, poderiam haver críticas mais técnicas com relação à representação e uso dos recursos destas tecnologias dentro do Gifflar.

Uma outra ameaça é que os participantes não utilizaram todas as funcionalidades da biblioteca, somente as que foram necessárias para resolver as *tasks*. Ou seja, existe a possibilidade de as *tasks* terem sido muito simples, não refletindo o código de um projeto real mais complexo. Ou seja, caso as *tasks* fossem voltadas para um projeto real, os participantes poderiam usufruir melhor das possibilidades que existem na API e produzir resultados diferentes pois as *tasks* envolveriam mais recursos. Por outro lado, isto levaria o teste a um nível mais complexo e que provavelmente demandaria muito mais conhecimento em blockchain e tempo livre do desenvolvedor para participar do teste, o que dificultaria ainda mais no recrutamento de participantes. O motivo da simplificação das *tasks* foi justamente para conseguir envolver desenvolvedores em geral que não necessariamente soubessem programar SmCs.

5.4 AVALIAÇÃO DE APLICABILIDADE

Esta seção apresenta os resultados da avaliação de aplicabilidade do Gifflar em outros projetos. Inicialmente são demonstrados os resultados do processo de busca dos artigos. Em seguida, são apresentados alguns projetos selecionados na pesquisa que utilizam a tecnologia blockchain e que poderiam ser aperfeiçoados caso utilizasse o Gifflar internamente.

5.4.1 Busca e Seleção dos Artigos

Na etapa de busca foram encontrados 100 artigos, 50 artigos em cada *string* de busca. Na etapa de seleção, foram selecionados 20 artigos candidatos à possibilidade de integração com o Gifflar. Destes artigos, 3 fizeram mais sentido em utilizar o Gifflar (com o IoTCocoa formam 4 projetos), portanto, foram selecionados para uma releitura e análise mais técnica e avaliação de possíveis cenários em que o Gifflar pode ser empregado.

A Tabela 5.3 mostra os artigos selecionados para análise técnica. Explicando de forma breve, o artigo **A** propõe um sistema de rastreabilidade do ciclo de vida de produ-

Tabela 5.3 Artigos selecionados na pesquisa de avaliação de aplicabilidade

Nº	Título do artigo	Ano	Referência
A	A decentralized application for the traceability process in the pharma industry	2020	(CHIACCHIO et al., 2020)
B	An Attribute-Based Access Control Model in RFID Systems Based on Blockchain Decentralized Applications for Healthcare Environments	2019	(FIGUEROA; AÑORGA; ARRIZABALAGA, 2019)
C	IoTCocoa-an IoT platform to assist gourmet cocoa production	2019	(ABIJAUDE et al., 2019)
D	Building an Ethereum-based Decentralized Smart Home System	2018	(XU et al., 2018)

tos farmacêuticos utilizando SmCs. O projeto **B** propõe um sistema de gerenciamento de produtos para a saúde e controle de acesso utilizando SmCs e IoT. Durante a releitura, percebeu-se que o Gifflar pode aplicado de forma semelhante nos dois projetos. Tanto o **A** quanto o **B** pretendem gerenciar dados de produtos em SmCs de forma que gera a demanda de estruturas variáveis, uma vez que os atributos utilizados nestes projetos variam de acordo com cada produto. Da mesma forma, o projeto **D** tem relação com o projeto **C**. O artigo **D** propõe um sistema para casas inteligentes utilizando dispositivos e sensores IoT, como sensores de temperatura ambiente, e salvam os dados capturados pelos sensores em SmCs. Ou seja, assim como o IoTCocoa, este projeto também pode lidar com diferentes dispositivos que apresentam diferentes comportamentos e atributos, e portanto, podem utilizar o Gifflar para o mesmo fim. Para não gerar repetições desnecessárias, foram escolhidos dois destes projetos para serem apresentados com mais detalhes, revelando também como o Gifflar poderia ser empregado.

5.4.2 Projetos

Esta seção trata com detalhes como o Gifflar pode ser aplicado em cada um dos dois projetos escolhidos. O primeiro deles é o mencionado IoTCocoa, um projeto do nosso grupo de pesquisa, que fez surgir a concepção inicial do Gifflar. O segundo é o projeto **A** da Tabela 5.3. Para cada projeto, inicialmente é realizada uma descrição inicial, depois uma motivação e, por fim, é apresentado como o Gifflar seria aplicado. Como mencionado na seção anterior, foi considerado que os outros dois projetos selecionados nos resultados da pesquisa resultariam em explicações bem semelhantes aos projetos a seguir.

5.4.2.1 IoTCocoa

Descrição

IoTCocoa é um sistema Web baseado em IoT que visa controlar e rastrear dados obtidos da produção de cacau *gourmet* por meio de sensores IoT. O *Hardware*, *Middleware* e Aplicação são os principais modelos deste sistema. Por meio de uma aplicação Web e sensores de IoT que capturam dados de fermentação das amêndoas, os agricultores podem

encontrar facilmente os níveis ideais de fenômenos físico-químicos para a produção de cacau fino. O *middleware* é a camada central que atua como intermediário; ele emprega as mensagens REST e o Protocolo Simples de Gerência de Rede (SNMP - do inglês *Simple Network Management Protocol*). O próximo estágio de seu desenvolvimento é integrar uma camada de blockchain e SmC para garantir melhor rastreabilidade, segurança e transparência. Gifflar é o *framework* escolhido para ajudar o projeto a alcançar este objetivo.

O cenário da IoTCocoa consiste em dispositivos IoT e um *middleware*. Um dispositivo IoT, neste caso, é composto por uma placa microcontroladora que pode ser conectada a vários sensores e/ou atuadores e principalmente a um módulo Wi-Fi. Esta placa é programada para conhecer algumas das características dos sensores e organizá-las em um código JSON. Quando a placa é ligada, ela aciona uma rota do *middleware* chamada `/init` pelo método HTTP POST para informar ao *middleware* sobre a identificação desta placa e as características de todos os sensores e/ou atuadores conectados a ela. Quando o *middleware* obtém o JSON, ele toma algumas decisões como persistir esses dados em um banco de dados e também escrever, compilar e implantar SmCs. Dessa forma, toda vez que a placa capturar dados de um sensor e enviá-los para o middleware, esses dados serão salvos em um SmC já implantado para que o *middleware* possa rastrear os dados dos sensores utilizando a blockchain como fonte.

Motivação

A primeira ideia para o desenvolvimento desda proposta, seria verificar quais são todos os dispositivos IoT a serem utilizados, assim como todos os sensores. Em seguida, criar um ou mais SmCs manualmente. Tem-se então duas possibilidades. Poderia ser criado somente um SmC o qual já abrange todas possibilidades de sensores, o desenvolvedor do sistema então faria a implantação do contrato e a integração com o *middleware*. A segunda opção seria criar um SmC para cada sensor, uma vez que cada sensor pode funcionar de forma diferente e conter dados de tipos de dado diferentes, assim, o desenvolvedor iria criar estes contratos, implantar e integrar cada um com o sistema. De fato, as duas formas funcionam, mas no caso de aparecer um outro dispositivo com sensores desconhecidos pelo sistema já sendo executado em ambiente de produção, surgiriam as seguintes questões:

- Na primeira opção o desenvolvedor não teria como modificar o SmC, uma vez que a blockchain não permite. Ele poderia, no entanto, criar um novo SmC que abrange dados dos novos sensores. O sistema então passaria a utilizar dois SmC para salvar dados de grupos de sensores diferentes.

- Ainda considerando a primeira opção, o desenvolvedor pode ter utilizado o padrão *Proxy* para permitir "modificar" o SmC. O padrão *Proxy* permite criar um SmC (*proxy*) que aponta para o SmC original que contém toda a implementação. O *proxy* então intermedia todas as requisições de aplicações externas com o contrato original e mantém o registro de todas as variáveis. Posteriormente, é possível adicionar mais funcionalidades ao código-fonte do contrato original, implantar novamente na rede, e atualizar o *proxy* com o novo endereço blockchain do contrato original. Ou seja,

ele teria que atualizar o código do antigo contrato adicionando informações dos novos sensores, implantar novamente na rede, e atualizar o contrato *Proxy* com o novo endereço blockchain do contrato atualizado.

- Na segunda opção, o desenvolvedor iria criar um novo SmC para cada novo sensor, implantar na rede e integrar novamente na aplicação.

Em todos os casos, seria necessário o desenvolvedor escrever um ou mais contratos, implantá-los na rede, integrar com a aplicação e atualizar a aplicação em produção novamente. Mais uma vez, todas as opções funcionam, porém, cada vez que o surgimento de diferentes sensores for sendo mais frequente, mais demorado o processo fica, consequentemente mais desenvolvedores a empresa irá precisar, aumentando assim a probabilidade de falhas e também os gastos. Portanto, é sempre melhor quando existe uma forma de aprimorar o processo.

Solução com Gifflar

Considerando agora que o desenvolvedor utilizou o Gifflar integrado com o *middleware*. Ele então cria algumas das possíveis configurações de sensores, e cria serviços que usam a Solgen para modelagem e definição das condições de criação dos SmCs para cada configuração de sensor.

O processo na prática então funcionaria assim: (a) O *middleware* recebe os dados iniciais dos sensores; (b) O *middleware* modela os SmCs para cada sensor baseado na configuração de cada um, de acordo com as condições definidas anteriormente pelo desenvolvedor; (c) O *middleware* realiza a escrita e compilação destes SmCs; (d) O *middleware* implanta os SmCs na rede e já obtém os endereços blockchain de cada um; (e) O *middleware* recebe dados de captura de sensores (como temperatura ou umidade) e repassa para os SmCs acionando as funções específicas de cada sensor; (f) surge um novo sensor com características diferentes, o *middleware* de acordo com sua configuração JSON, modela o SmC e repete o processo, agora para este novo sensor.

Uma demonstração prática deste processo foi apresentada no artigo "*A Framework to Generate Smart Contracts On the Fly*" (SANTIAGO; ABIJAUDE; GREVE, 2021a) citados no Capítulo 1. Neste experimento, foi utilizado um sensor de temperatura e umidade chamado DHT11 e uma placa microcontroladora com Wi-Fi integrado. A rede blockchain utilizada para implantar os SmCs foi uma rede de teste da Ethereum.

5.4.2.2 Cadeia de Suprimentos na Indústria Farmacêutica

Descrição

Este é um projeto bem relevante que propõe um sistema de rastreabilidade do ciclo de vida de produtos farmacêuticos utilizando SmCs desenvolvidos em Solidity para a rede Ethereum. Neste, os autores implementam um protótipo para a proposta criando uma DApp para uma indústria local chamada SIFI® SPA.

O SIFI® SPA é uma indústria que já possui um sistema de identificação dos produtos, de forma que torna possível rastreá-los enquanto estão percorrendo as etapas da cadeia

de suprimentos. Os autores do projeto propõem 5 níveis de sistema, onde o nível 5 inclui o uso de SmCs. No primeiro e segundo nível, os produtos são encaixotados e também recebem o identificador único através de maquinários da indústria. Todo esse maquinário exporta dados em tempo real da preparação dos produtos para o sistema no nível 3. No nível 4, o sistema armazena estes dados em um banco de dados para depois ser enviado para a blockchain e também para a autoridade central que regula este processo. No último nível, estes dados são encriptados e salvos no SmC, então os produtos são liberados para a rede de distribuição.

Uma vez que os produtos entram na rede de distribuição, o SmC permite que a propriedade do produto seja transferida para cada setor que torna-se responsável por ele. O proprietário do produto é o único que tem a permissão para realizar alterações nos dados do produto no SmC. No início do processo, a indústria é a proprietária do produto, depois, dependendo do tipo de produto, pode ser enviado para um armazém, depois um atacadista e então um hospital. Nesta sequência, o próximo proprietário então seria o armazém, em seguida o atacadista, e assim por diante. Quando o produto chega no cliente final, o token de propriedade é destruído, e ninguém mais pode alterar os dados daquele produto.

A fim de permitir esse processo, o SmC possui uma estrutura (*struct*, como no código C ou C++) para salvar dados sobre o produto (nome, classe, data de expiração, identificador único, etc), um atributo para definir o atual proprietário e funções para alterar o status do produto, o token de propriedade, dentre outros. No protótipo criado pelo projeto, os dados do produto são gerados logo depois que o produto é empacotado. Em seguida, os dados são salvos no SmC como um registro do tipo *struct* que é adicionado em uma lista e pode depois ser acessado por um identificador e atualizado.

Motivação

Dependendo dos objetivos do projeto, pode ser importante ou não salvar dados mais completos de um produto no SmC. Caso não seja necessário, esta solução já é o suficiente, uma vez que somente precisaria salvar dados mínimos que depois poderiam ser unidos com dados mais completos do banco de dados do sistema. Mas, de acordo com o próprio autor, a ideia do SmC é que este armazene dados dos produtos de forma que não seja necessário manter estes dados no sistema por tantos anos. Dessa forma, o SmC então deve salvar atributos mais completos dos produtos.

Sabe-se que uma farmácia pode vender diversos tipos de produtos diferentes além de remédios, como sabonete, escova de dentes, mamadeira, desodorante, barras de cereal, suplementos alimentares, alicate de unhas, dentre outros. Ou seja, são produtos que diferem bastante entre si e que, portanto, possuem atributos diferentes. Outra questão seria o caso de produtos que precisam de um controle maior no transporte. Por exemplo, determinados produtos, como alguns remédios e alimentos, precisam de um controle de temperatura durante o transporte e armazenamento. Logo, poderia ser interessante que para estes produtos possa haver uma funcionalidade no SmC para definir a temperatura em que estes produtos foram submetidos. Assim, ao fim do ciclo, uma farmácia ou até um cliente final poderia verificar com mais certeza se a qualidade daquele produto se

mantém.

Considerando estas colocações, percebe-se que um *struct* não seria o suficiente para salvar os dados dos produtos, uma vez que cada tipo de produto tem suas características individuais diferentes dos outros tipos, e um *struct* define um padrão de atributos genérico.

Solução com Gifflar

Uma solução utilizando o Gifflar seria modelar SmCs específicos para cada produto. Assim, ao invés de um só SmC que contém uma lista de *structs*, seriam vários SmCs, cada um para um produto diferente. Dessa forma, cada SmC teria os atributos e funcionalidades característicos daquele tipo de produto.

Na prática, funcionaria assim: Primeiro o produto é encaixotado pela empresa, depois que o sistema já possui os dados daquele produto, inicia-se a modelagem do SmC com o Gifflar com base nestes dados (lembrando que as condições de modelagem devem antes ser definidas pelo desenvolvedor ao criar o sistema utilizando os métodos da Solgen). O sistema então gera o código do SmC do produto, compila e aciona o método de implantação na blockchain. O sistema então salva o endereço blockchain do contrato para posterior atualizações no status do produto e transferência de propriedade. Posteriormente, segue o fluxo normal de transporte repassando o token de propriedade do produto e, no final, os dados podem ser obtidos do SmC com os atributos completos e histórico de status. Durante o transporte do produto, poderia também, se for o caso, capturar e salvar no SmC dados de temperatura ambiente, por exemplo.

Capítulo 6

CONCLUSÕES E TRABALHOS FUTUROS

6.1 CONCLUSÕES

A blockchain trouxe consigo um novo modelo de programação de aplicações descentralizadas, onde estas já não precisam mais depender de serviços centralizados, pois dispõem de uma estrutura descentralizada à prova de violação e que integra diversos conceitos de computação distribuída tolerante a falhas e de segurança computacional já consolidados na literatura, tais como protocolos de consenso distribuído e criptografia de chave assimétrica. O contrato inteligente (SmC), implementado pela blockchain da Ethereum, é atualmente o principal artefato que permite a criação das DApps.

Porém, a programação de SmCs apresenta um alto nível de complexidade, por isso, são poucos os desenvolvedores que possuem esse conhecimento. Neste sentido, muitos trabalhos vêm propondo diferentes maneiras de facilitar a criação de SmC através de um processo de geração automática de código-fonte, onde parte-se da definição de um modelo que representa o SmC e, em seguida, permite-se acionar uma funcionalidade que traduz este modelo em código de SmC. Este modelo é utilizado como uma representação abstrata do SmC. Um modelo onde o nível de abstração é alto, permite ser utilizado com maior facilidade pelo ser humano, como diagramas visuais. Por outro lado, modelos com um nível de abstração baixo, reduz a fácil compreensão, de modo que somente um grupo específico de pessoas sejam capazes de entender e utilizar tal modelo, como uma representação de SmC em nível de código de programação.

Foram identificados 28 trabalhos da literatura que propõem projetos com esta abordagem de geração automática de contratos inteligentes. Além disso, esta dissertação apresentou duas tabelas comparativas com base em 9 critérios diferentes. Ocorre que a abordagem utilizada pelos trabalhos relacionados gera uma dependência do desenvolvedor, pois, toda vez que uma aplicação necessita de um novo SmC, em determinada situação, precisará recorrer ao desenvolvedor para modelar este novo SmC, gerar o código correspondente, compilar, implantar na rede e integrar com a aplicação.

Esta dissertação contribui para diminuir a dependência do desenvolvedor, oferecendo a possibilidade de construir sistemas capazes de criar modelos, gerar códigos de SmCs,

compilá-los, implantá-los e interagir com estes por conta própria em tempo de execução e de forma dinâmica (sem o uso de fragmentos de códigos pré-definidos). Até onde sabemos, este trabalho de dissertação é um dos primeiros a propor uma ferramenta que, além de facilitar o desenvolvimento de SmCs, possa permitir que sistemas escrevam e implantem contratos na blockchain em tempo de execução de forma dinâmica.

A principal contribuição dessa dissertação é a proposta e desenvolvimento do Gifflar, um *framework* que permite desenvolvedores criarem sistemas com a capacidade de gerarem SmCs em tempo de execução. O *framework* surge a partir da necessidade do projeto IoTCocoa (um sistema de monitoramento de fermentação de Cacau), assim contribuindo para o seu propósito, porém não limita-se a este escopo, podendo ser aplicado em diversos outros contextos.

O *framework* proposto introduz um novo paradigma onde sistemas podem atuar como desenvolvedores de SmCs. Esta dissertação apresentou a estrutura e os principais componentes internos do Gifflar que, além de utilizar OOP, segue as boas práticas dos princípios SOLID. Além disso, apresenta um fluxograma que objetiva guiar outros projetos para definir quando é adequado utilizar o Gifflar. Esta é uma abordagem inovadora que pode contribuir muito na evolução das aplicações blockchain.

O Gifflar contém um núcleo central que é responsável por unir todas as ferramentas internas. Uma destas ferramentas é uma interface de linha de comandos que visa facilitar o processo de desenvolvimento com o *framework*. A principal ferramenta interna é a Solgen, uma biblioteca estruturada em componentes, onde cada um tem sua responsabilidade bem definida fazendo o uso de padrões de projeto. A Solgen oferece um conjunto de métodos que podem ser utilizados pelo desenvolvedor para modelar os SmCs e configurar todo o processo de automatização da geração dos seus códigos-fonte e *deploy* dos mesmos. A modelagem consiste em gerar a representação do SmC em JSON para posteriormente ser transformado em código de SmC.

A Solgen segue um fluxo que é organizado em quatro etapas diferentes. Na primeira etapa é realizado o processo de modelagem do SmC utilizando métodos da própria biblioteca. Posteriormente, o JSON gerado na etapa de modelagem é enviado para a etapa de escrita, que irá interpretar o JSON criado e então criar o código-fonte do SmC. A próxima etapa é a de compilação, que é responsável por gerar os *bytecodes* e a ABI (Interface Binária de Aplicação) a partir do código-fonte gerado na etapa de escrita. Os *bytecodes* são as instruções em baixo nível do SmC que são enviadas para a máquina virtual da rede blockchain. Já a ABI é a interface utilizada por aplicações para comunicar-se com o SmC. A última etapa é responsável pela implantação do contrato na rede blockchain utilizando os *bytecodes* e retornar a instância do contrato na rede utilizando a ABI.

É importante destacar que, o nível de abstração utilizado no modelo de representação do Gifflar (JSON) é menor do que o nível utilizado pelos trabalhos relacionados (grande parte utilizam diagramas e outras estruturas visuais). Ou seja, no Gifflar, somente um grupo específico consegue entender e modelar os SmCs utilizando a biblioteca, que são os desenvolvedores de software. O motivo desta escolha foi justamente para permitir que os desenvolvedores possam utilizar os métodos da Solgen para integrar no sistema que posteriormente irá modelar e gerar os SmCs em tempo de execução. A partir desta estratégia, as aplicações podem evoluir para um outro nível onde podem atuar como

desenvolvedoras de SmC.

Além disso, como o modelo do Gifflar é voltado para desenvolvedores permite que projetos como os próprios trabalhos relacionados possam beneficiar-se deste *framework*. Utilizando como exemplo um projeto planeje utilizar componentes visuais para permitir usuários modelarem e gerarem SmCs, o sistema deste projeto poderia utilizar internamente, em cada componente visual, um código de modelagem de SmC utilizando a Solgen. Então o usuário, ao acionar o mecanismo para transformar o modelo visual em SmC, o sistema utiliza a Solgen para gerar o JSON e então o código-fonte do SmC.

A fim de validar a aceitação dos desenvolvedores em utilizar o Gifflar e evidenciar a aplicação do *framework* em alguns sistemas, foram realizadas duas avaliações. A primeira avaliação consistiu em um teste de usabilidade, onde desenvolvedores puderam testar a API da Solgen utilizada para modelar os contratos. Esta avaliação seguiu metodologias já consolidadas na literatura que foram essenciais para a condução da mesma. Os resultados obtidos foram muito positivos sobre a usabilidade da interface proposta, assim como apontaram algumas falhas de usabilidade que foram analisadas e serão consideradas nas próximas alterações do *framework*.

A segunda avaliação demonstrou de forma conceitual como o Gifflar pode ser utilizado nos projetos que usam blockchain. Após uma revisão da literatura, foram encontrados quatro projetos com potencial de se beneficiar do Gifflar. Dois deles foram apresentados com mais detalhes ao fim da análise. Um destes foi o IoTCocoa, por ser um projeto do grupo de pesquisa que visamos contribuir, já o segundo é um projeto de rastreabilidade na indústria farmacêutica. Para cada um deles, apresenta-se a motivação em utilizar o Gifflar e detalha-se de que forma o Gifflar pode ser aplicado em cada contexto.

Os resultados obtidos nesta proposta e desenvolvimento trazem uma grande contribuição para o domínio inserido, porém, ainda existem algumas correções a serem realizadas, assim como alguns desafios a serem solucionados como a garantia da semântica e segurança dos contratos gerados pelos sistemas que utilizam o Gifflar. Estes e outros fatores serão retomados em trabalhos futuros.

6.2 TRABALHOS FUTUROS

Este trabalho de dissertação é promissor, pelo que se observa nos resultados apresentados, mas, ainda precisa evoluir, podendo ser aperfeiçoado em vários pontos. Esta seção apresenta algumas ideias que surgiram durante o desenvolvimento deste trabalho de dissertação e que podem ser definidas como projetos futuros. Estes são somente alguns dos caminhos que o Gifflar pode seguir para aperfeiçoar suas funcionalidades, conforme o projeto for evoluindo, outras ideias surgirão certamente, principalmente porque a blockchain é uma tecnologia que hoje não para de evoluir e trazer novos recursos.

Os itens abaixo foram divididos em dois grupos. O primeiro grupo refere-se à trabalhos futuros que requerem um maior aprofundamento de pesquisa científica. O segundo grupo refere-se à trabalhos futuros que requerem desenvolvimento tecnológico mais direto.

6.2.1 Grupo 1

- **Correção automática de contratos**: De fato, este é um dos principais projetos futuros. Pretende-se integrar alguma ferramenta de correção automática de *bugs* para corrigir possíveis vulnerabilidades e erros de semântica na geração de contratos do Gifflar. Quatro delas foram citadas nesta dissertação, porém, existem várias outras na literatura, por isso, é interessante uma revisão sistemática de ferramentas desta natureza para encontrar de forma direcionada a ferramenta ideal para integração no Gifflar. Não está descartada a ideia de desenvolver um módulo próprio do Gifflar para este fim, mas esta ideia somente será considerada caso a análise da revisão sistemática não convirja para uma ferramenta ideal.

- **Engenharia reversa**: Em algum momento pode ser necessário realizar o processo reverso, ou seja, caso um desenvolvedor tenha um código em Solidity e queira aproveitar para inserir em seu sistema que implementa o Gifflar, como fará para traduzir o código-fonte Solidity em modelo SmC do Gifflar? Até então teria que fazer manualmente, acionando os métodos da Solgen. Este projeto futuro visa facilitar este processo criando um módulo *parser* do Gifflar, que analisa o código-fonte e cria o modelo de contrato deste código fonte no Gifflar para continuar realizando modificações com a própria modelagem da biblioteca.

- **Modelo genérico**: Este projeto permitirá que um desenvolvedor crie um modelo de SmC que pode gerar diferentes tipos de códigos-fonte de SmC. Aplicando esta funcionalidade junto com a engenharia reversa, pode ser possível transformar o código-fonte do SmC de uma linguagem para outra, além de facilitar o processo de modelagem. A ideia é criar um modelo que represente os contratos de forma mais genérica, com um nível de abstração maior. Dessa forma, seria possível representar um SmC independente da linguagem. A modelagem com este modelo irá diminuir a liberdade de personalização de um SmC, mas, uma vez que representa um contrato de forma genérica, poderá servir como modelo na conversão de uma linguagem de SmC para outra. O usuário também poderá escolher utilizar diretamente o modelo genérico ou o modelo específico da linguagem de SmC para a modelagem do contrato. O modelo genérico terá uma abstração maior na modelagem, mas irá limitar a personalização. Já o modelo da própria linguagem de SmC, funciona como explicado no Capítulo 3, permite uma maior personalização, porém envolve alguns conceitos específicos da linguagem que o desenvolvedor precisa saber.

- **Aprendizagem de máquina e Inteligência Artificial**: Até então, para um sistema gerar os SmCs de forma dinâmica, um desenvolvedor precisa criar a modelagem do contrato com os métodos da Solgen considerando as condições do seu sistema. Este projeto visa realizar o processo de modelagem também de forma automática utilizando aprendizagem de máquina. Neste sentido, o Gifflar teria um modelo treinado para modelar SmCs. Assim, o desenvolvedor não precisaria mais se preocupar com a etapa de modelagem do Gifflar, teria somente que configurar algumas funcionalidades do *framework* para integração com aplicação e o próprio

modelo treinado do Gifflar iria criar os modelos e gerar os SmCs. Este projeto pode até evoluir para o uso de Inteligência Artificial (GUPTA et al., 2020), que além de gerar de forma inteligente os modelos utilizando a Solgen, os códigos já irão integrar as melhores práticas para evitar falhas de vulnerabilidades e erros semânticos.

6.2.2 Grupo 2

- **Propriedades ainda não suportadas**: Como apresentado na Seção 3.3 do Capítulo 3, existem algumas propriedades do Solidity ainda não suportadas pela biblioteca. A versão atual fornece um método em que o desenvolvedor pode incluir qualquer código Solidity que ainda não é representado por métodos específicos, mas, ainda assim, são instruções que no futuro vão ter seu próprio método específico.

- **Modificações em SmC implantado**: Sabe-se que um SmC, uma vez implantado na rede, não pode ser modificado. Porém, existe uma estratégia que faz uso do padrão *Proxy* para criar um SmC que intermedia as requisições do contrato criado pelo desenvolvedor, que é o contrato original. Este contrato *Proxy* armazena todos os dados de memória do contrato original, e, portanto, quando o desenvolvedor quer realizar uma alteração no contrato original, ele modifica o código fonte, realiza uma nova implantação e troca o apontamento do SmC *Proxy* para a nova implementação do contrato original. Ou seja, o contrato em si não é alterado, mas sim o apontamento do *Proxy* que intermedia as requisições. A ideia deste projeto futuro é permitir ao usuário utilizar este padrão com facilidade para que, quando quiser utilizar esta flexibilidade de "modificar" um SmC que já foi implantado, possa criar alguns métodos na biblioteca que permita realizar este processo.

- **Gifflar como serviço**: A Solgen atual foi desenvolvida para sistemas em TypeScript ou JavaScript. Entretanto, existe a possibilidade de expandir para outras linguagens utilizarem o *framework* como um serviço. Para isso, será criado um servidor em Node.js que implementa o Gifflar e exporta rotas para escrever, compilar e implantar os SmCs. Estes sistemas então devem modelar os SmCs direto no código JSON, como na Figura 3.2 do Capítulo 3, e então enviar a estrutura como corpo da requisição da rota de escrita do servidor do Gifflar. Uma outra opção seria criar uma biblioteca específica para cada linguagem de programação, para que a modelagem seja facilitada utilizando os métodos da biblioteca como é feito no TypeScript, o que já está também dentro dos projetos futuros no próximo item desta lista.

- **Bibliotecas para outras linguagens de programação**: A fim de simplificar a modelagem de SmCs em outras linguagens de programação, está nos planos a criação de bibliotecas para realizar a modelagem como é feita no TypeScript. É certo que este é um projeto extenso e contínuo, uma vez que existem hoje diversas linguagens de programação. Mas isto será feito conforme for observado uma demanda para uma certa linguagem. Além disso, como o projeto é de código aberto, outros desenvolvedores podem ajudar no desenvolvimento destas bibliotecas, de forma que cada time de desenvolvedores pode ser responsável por oferecer suporte

a uma linguagem de programação diferente.

- **Bibliotecas para outras linguagens de SmC**: Com o objetivo de permitir criar contratos para outros tipos de linguagens de SmC, serão criadas bibliotecas para cada tipo de linguagem de SmC. Como cada linguagem tem suas diferenças na sintaxe, o modelo JSON para cada uma será diferente. O usuário então no momento da criação de um projeto Gifflar, poderá escolher qual a linguagem de SmC que irá utilizar, ou, em caso de sistemas multi-chain (que utiliza duas ou mais redes blockchain), poderá também criar modelos de SmC utilizando bibliotecas para diferentes linguagens de SmC simultaneamente, de forma que o sistema pode administrar SmCs em redes blockchain diferentes.

- **Modelagem visual**: Uma vez que o modelo genérico esteja desenvolvido e testado, este trará uma abstração maior na modelagem de contratos e, portanto, pode abranger o público alvo do Gifflar, que até então eram somente de desenvolvedores. Este projeto futuro visa desenvolver uma página para modelagem visual com o Gifflar, como feito por alguns trabalhos relacionados apresentados no Capítulo 2. A ideia aqui é utilizar algum diagrama ou estrutura que permita uma pessoa sem conhecimento em computação possa desenvolver um SmC para sua necessidade, seria um sistema de desenvolvimento *low code* para SmC. Para isso, é interessante realizar um estudo na literatura de estruturas *low code* mais utilizadas e consolidadas para geração automática de código-fonte, seja para contratos ou código-fonte de linguagem de programação tradicional.

- **Geração automática de API**: Uma questão que pode surgir é a seguinte: Uma vez dispondo da gramática da linguagem de SmC, é possível gerar ou atualizar código da Solgen automaticamente? A resposta é sim. Para isso, seria criado um script ou um módulo do Gifflar que interprete a gramática e crie a API da biblioteca de forma automática. Este é um dos trabalhos futuros. Isto poderia também ser feito logo no início do projeto, porém, foi preferível, a princípio, definir toda a organização da primeira biblioteca, para então seguir com este outro projeto, onde um módulo do Gifflar é capaz de interpretar uma gramática e gerar a API da biblioteca já seguindo a organização definida nesta dissertação. Este projeto será muito útil para as atualizações de versão das linguagens de SmC nas bibliotecas.

6.3 DIFICULDADES E OBSTÁCULOS

Uma das dificuldades encontradas no desenvolvimento do projeto é que a gramática do Solidity, disponível no *website* da documentação[1], já está atualizada para a versão mais nova, e o Gifflar está utilizando a versão 0.6.0 (versão mais antiga). Portanto, existem algumas instruções na gramática que são diferentes na versão do Solidity usada pelo Gifflar. Ainda assim, foi possível encontrar a gramática da versão 0.6.0 no repositório do Solidity no GitHub[2], mas essa gramática não considera todas as condições e possibilidades

[1]https://docs.soliditylang.org/en/develop/grammar.html
[2]https://github.com/ethereum/solidity/blob/v0.6.0/docs/grammar.txt

de uso de cada *statement*. Por exemplo, uma criação de contrato permite a definição de variáveis de estado, mas uma criação de interface não permite. A criação de um contrato também permite a definição de construtores, mas a criação de interfaces e bibliotecas de contrato não permitem. Então, muitas destas condições foram testadas de forma manual nos próprios testes automáticos ou utilizando a plataforma de edição de código Solidity oficial da Ethereum, o Remix, que permite também a compilação e implantação na rede.

Uma outra dificuldade de menor ordem foi identificar uma quantidade razoável de pessoas para participar da avaliação de usabilidade. Isto ocorreu porque o público alvo era específico (desenvolvedores com no mínimo conhecimento básico em *JavaScript*). Além disso, apesar de várias pessoas serem contactadas para participar da avaliação, muitas abdicaram do teste alegando sobrecarga de outros compromissos. Em compensação, todas as pessoas que participaram contribuíram com a avaliação e gostaram muito, não só do *framework*, como também o contato com a tecnologia blockchain, pois muitos dos que participaram tinham conhecimento muito básico ou nenhum conhecimento na tecnologia.

REFERÊNCIAS BIBLIOGRÁFICAS

ABIJAUDE, J. W. et al. Iotcocoa-an iot platform to assist gourmet cocoa production. In: IEEE. *2019 IEEE Latin-American Conference on Communications (LATINCOM)*. [S.l.], 2019. p. 1–6.

ARGAÑARAZ, M. et al. Detection of vulnerabilities in smart contracts specifications in ethereum platforms. In: SCHLOSS DAGSTUHL–LEIBNIZ-ZENTRUM FUER INFORMATIK. *9th Symposium on Languages, Applications and Technologies (SLATE 2020)*. [S.l.], 2020. v. 83, p. 1–16.

ASTIGARRAGA, T. et al. Empowering business-level blockchain users with a rules framework for smart contracts. In: SPRINGER. *International Conference on Service-Oriented Computing*. [S.l.], 2018. p. 111–128.

ATZEI, N.; BARTOLETTI, M.; CIMOLI, T. A survey of attacks on ethereum smart contracts (sok). In: SPRINGER. *International conference on principles of security and trust*. [S.l.], 2017. p. 164–186.

BEHROOZI, M.; PARNIN, C.; BARIK, T. Hiring is broken: What do developers say about technical interviews? In: IEEE. *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. [S.l.], 2019. p. 1–9.

BLACKWELL, A. et al. Cognitive dimensions of notations: Design tools for cognitive technology. In: _____. [S.l.: s.n.], 2001. p. 325–341. ISBN 978-3-540-42406-2.

BOOCH, G. et al. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes*, ACM New York, NY, USA, v. 33, n. 5, p. 29–29, 2008.

BRENT, L. et al. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

BUTERIN, V. et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

CACHIN, C.; VUKOLIĆ, M. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

CHIACCHIO, F. et al. A decentralized application for the traceability process in the pharma industry. *Procedia Manufacturing*, Elsevier, v. 42, p. 362–369, 2020.

CHOUDHURY, O. et al. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. In: IEEE. *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. [S.l.], 2018. p. 963–970.

CHRISTIDIS, K.; DEVETSIKIOTIS, M. Blockchains and smart contracts for the internet of things. *Ieee Access*, Ieee, v. 4, p. 2292–2303, 2016.

CRAWFORD, S. E.; OSTROM, E. A grammar of institutions. *American political science review*, JSTOR, p. 582–600, 1995.

DIAZ, D.; HARMES, R. *Pro JavaScript design patterns*. [S.l.]: Apress, 2008.

DIKA, A. *Ethereum smart contracts: Security vulnerabilities and security tools*. Dissertação (Mestrado) — NTNU, 2017.

DWIVEDI, V.; NORTA, A. Auto-generation of smart contracts from a domain-specific xml-based language. In: *Intelligent Data Engineering and Analytics*. [S.l.]: Springer, 2022. p. 549–564.

EID, C. *Model-driven approach to smart contract development with automatic code generation*. Tese (Doutorado) — Notre Dame University-Louaize., 2020.

ERICSSON, K.; SIMON, H. *Protocol Analysis: Verbal Reports as Data*. [S.l.: s.n.], 1993. ISBN 9780262272391.

FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Communications of the ACM*, ACM New York, NY, USA, v. 40, n. 10, p. 32–38, 1997.

FENTON, S.; FENTON; SPEARING. *Pro TypeScript*. [S.l.]: Springer, 2014.

FERNÁNDEZ-CARAMÉS, T. M.; FRAGA-LAMAS, P. A review on the use of blockchain for the internet of things. *IEEE Access*, IEEE, v. 6, p. 32979–33001, 2018.

FIGUEROA, S.; AÑORGA, J.; ARRIZABALAGA, S. An attribute-based access control model in rfid systems based on blockchain decentralized applications for healthcare environments. *Computers*, MDPI, v. 8, n. 3, p. 57, 2019.

FILIPPI, P. D. F. D. *Blockchain and the law: The rule of code*. [S.l.]: Harvard University Press, 2018.

FLANAGAN, D. *JavaScript: o guia definitivo*. [S.l.]: Bookman Editora, 2004.

FRANCO, P. *Understanding Bitcoin: Cryptography, engineering and economics*. [S.l.]: John Wiley & Sons, 2014.

FRANTZ, C. K.; NOWOSTAWSKI, M. From institutions to code: Towards automated generation of smart contracts. In: IEEE. *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. [S.l.], 2016. p. 210–215.

FU, Y.; ZHU, J. Trusted data infrastructure for smart cities: a blockchain perspective. *Building Research & Information*, Taylor & Francis, p. 1–17, 2020.

GAMMA, E. et al. Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 1995.

GAO, Z. et al. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering*, IEEE, 2020.

GARAMVÖLGYI, P. et al. Towards model-driven engineering of smart contracts for cyber-physical systems. In: IEEE. *2018 48th annual IEEE/IFIP international conference on dependable systems and networks workshops (DSN-W)*. [S.l.], 2018. p. 134–139.

GARCÍA-BAÑUELOS, L. et al. Optimized execution of business processes on blockchain. In: SPRINGER. *International Conference on Business Process Management*. [S.l.], 2017. p. 130–146.

GARTNER. *Hype Cycle Shows Most Blockchain Technologies Are Still Five to 10 Years Away From Transformational Impact*. 2019. Https://www.gartner.com/en/newsroom/press-releases/2019-10-08-gartner-2019-hype-cycle-shows-most-blockchain-technologies-are-still-five-to-10-years-away-from-transformational-impact. Accessed in Nov. 01, 2020.

GREVE, F. et al. Blockchain e a revolução do consenso sob demanda. *Livro de Minicursos do SBRC*, v. 1, p. 1–52, 2018.

GUPTA, R. et al. Smart contract privacy protection using ai in cyber-physical systems: tools, techniques and challenges. *IEEE access*, IEEE, v. 8, p. 24746–24772, 2020.

HAMDAQA, M.; METZ, L. A. P.; QASSE, I. Icontractml: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. In: *Proceedings of the 12th System Analysis and Modelling Conference*. [S.l.: s.n.], 2020. p. 34–43.

JIANG, B.; LIU, Y.; CHAN, W. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: IEEE. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2018. p. 259–269.

JOHNSON, R. E.; FOOTE, B. Designing reusable classes. *Journal of object-oriented programming*, v. 1, n. 2, p. 22–35, 1988.

JOSHI, B. *Beginning SOLID Principles and Design Patterns for ASP. NET Developers*. [S.l.]: Springer, 2016.

JURGELAITIS, M.; BUTKIENĖ, R. et al. Solidity code generation from uml state machines in model-driven smart contract development. *IEEE Access*, IEEE, v. 10, p. 33465–33481, 2022.

JURGELAITIS, M. et al. Smart contract code generation from platform specific model for hyperledger go. In: SPRINGER. *World Conference on Information Systems and Technologies*. [S.l.], 2021. p. 63–73.

KARAME, G. O.; ANDROULAKI, E. *Bitcoin and blockchain security*. [S.l.]: Artech House, 2016.

LADLEIF, J.; FRIEDOW, C.; WESKE, M. An architecture for multi-chain business process choreographies. In: SPRINGER. *International Conference on Business Information Systems*. [S.l.], 2020. p. 184–196.

LI, Y. et al. Design and management of a distributed hybrid energy system through smart contract and blockchain. *Applied Energy*, Elsevier, v. 248, p. 390–405, 2019.

LIAO, C.-F. et al. Toward a service platform for developing smart contracts on blockchain in bdd and tdd styles. In: IEEE. *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*. [S.l.], 2017. p. 133–140.

LIU, C. et al. Reguard: finding reentrancy bugs in smart contracts. In: IEEE. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. [S.l.], 2018. p. 65–68.

LÓPEZ-PINTADO, O. et al. Caterpillar: a business process execution engine on the ethereum blockchain. *Software: Practice and Experience*, Wiley Online Library, v. 49, n. 7, p. 1162–1193, 2019.

LUO, H. et al. Construction payment automation through smart contract-based blockchain framework. In: IAARC PUBLICATIONS. *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*. [S.l.], 2019. v. 36, p. 1254–1260.

LUU, L. et al. Making smart contracts smarter. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. [S.l.: s.n.], 2016. p. 254–269.

MAO, D. et al. Visual and user-defined smart contract designing system based on automatic coding. *Ieee Access*, IEEE, v. 7, p. 73131–73143, 2019.

MARKIEWICZ, M. E.; LUCENA, C. J. de. Object oriented framework development. *XRDS: Crossroads, The ACM Magazine for Students*, ACM New York, NY, USA, v. 7, n. 4, p. 3–9, 2001.

MAVRIDOU, A.; LASZKA, A. Tool demonstration: Fsolidm for designing secure ethereum smart contracts. In: SPRINGER. *International Conference on Principles of Security and Trust*. [S.l.], 2018. p. 270–277.

MAVRIDOU, A. et al. Verisolid: Correct-by-design smart contracts for ethereum. In: SPRINGER. *International Conference on Financial Cryptography and Data Security*. [S.l.], 2019. p. 446–465.

MCFARLAND, D. S. *Javascript & jQuery: the missing manual*. [S.l.]: "O'Reilly Media, Inc.", 2011.

MERLEC, M. M.; LEE, Y. K.; IN, H. P. Smartbuilder: A block-based visual programming framework for smart contract development. In: IEEE. *2021 IEEE International Conference on Blockchain (Blockchain)*. [S.l.], 2021. p. 90–94.

MIN, H. *The essentials of supply chain management: New business concepts and applications*. [S.l.]: FT Press, 2015.

NAKAMOTO, S.; BITCOIN, A. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 2008.

NAKAMOTO, S. et al. Bitcoin: A peer-to-peer electronic cash system. Working Paper, 2008.

NANCE, C. *TypeScript Essentials*. [S.l.]: Packt Publishing Ltd, 2014.

NGUYEN, T. D.; PHAM, L. H.; SUN, J. Sguard: towards fixing vulnerable smart contracts automatically. In: IEEE. *2021 IEEE Symposium on Security and Privacy (SP)*. [S.l.], 2021. p. 1215–1229.

NIELSEN, J. *Usability engineering*. [S.l.]: Elsevier, 1994.

OSMANI, A. *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. [S.l.]: "O'Reilly Media, Inc.", 2012.

PACE, G. J.; SÁNCHEZ, C.; SCHNEIDER, G. Reliable smart contracts. In: SPRINGER. *International Symposium on Leveraging Applications of Formal Methods*. [S.l.], 2020. p. 3–8.

PETTERSSON, J.; EDSTRÖM, R. *Safer smart contracts through type-driven development*. Dissertação (Mestrado), 2016.

PICCIONI, M.; FURIA, C. A.; MEYER, B. An empirical study of api usability. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, p. 5–14, 2013.

PRAITHEESHAN, P. et al. Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*, 2019.

PRAUSE, G.; BOEVSKY, I. Smart contracts for smart rural supply chains. *Bulgarian Journal of Agricultural Science*, v. 25, n. 3, p. 454–463, 2019.

QASSE, I.; MISHRA, S.; HAMDAQA, M. icontractbot: a chatbot for smart contracts' specification and code generation. In: IEEE. *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*. [S.l.], 2021. p. 35–38.

QIN, P. et al. Towards self-automatable and unambiguous smart contracts: Machine natural language. In: SPRINGER. *International Conference on e-Business Engineering*. [S.l.], 2019. p. 479–491.

RASTI, A. *From Symboleo to Smart Contracts: A Code Generator*. Tese (Doutorado) — Université d'Ottawa/University of Ottawa, 2022.

REBELLO, G. et al. Correntes de blocos: Algoritmos de consenso e implementação na plataforma hyperledger fabric. In: _____. [S.l.: s.n.], 2019. p. 93–148. ISBN 9788576694717.

RODLER, M. et al. {EVMPatch}: Timely and automated patching of ethereum smart contracts. In: *30th USENIX Security Symposium (USENIX Security 21)*. [S.l.: s.n.], 2021. p. 1289–1306.

SANTIAGO, L.; ABIJAUDE, J.; GREVE, F. A framework to generate smart contracts on the fly. In: *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2021. p. 410–415.

SANTIAGO, L.; ABIJAUDE, J. W.; GREVE, F. Gifflar: a framework to generate smart contracts on the fly. In: *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*. [S.l.: s.n.], 2021. p. 214–219.

SANTIAGO, L. M. S. et al. Blockchain applied to academic environments as a way to ensure educational process quality control. In: IEEE. *2020 IEEE World Conference on Engineering Education (EDUNINE)*. [S.l.], 2020. p. 1–5.

SETHI, P.; SARANGI, S. R. Internet of things: architectures, protocols, and applications. *Journal of Electrical and Computer Engineering*, Hindawi, v. 2017, 2017.

SINGH, A. et al. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security*, Elsevier, v. 88, p. 101654, 2020.

SINGH, H.; HASSAN, S. I. Effect of solid design principles on quality of software: An empirical assessment. *International Journal of Scientific & Engineering Research*, v. 6, n. 4, 2015.

SINGHAL, B.; DHAMEJA, G.; PANDA, P. S. *Beginning Blockchain: A Beginner's Guide to Building Blockchain Solutions*. [S.l.]: Springer, 2018.

SINGHAL, B.; DHAMEJA, G.; PANDA, P. S. Building an ethereum dapp. In: *Beginning Blockchain*. [S.l.]: Springer, 2018. p. 319–375.

SKOTNICA, M.; PERGL, R. Das contract-a visual domain specific language for modeling blockchain smart contracts. In: SPRINGER. *Enterprise Engineering Working Conference*. [S.l.], 2020. p. 149–166.

SO, S. et al. Verismart: A highly precise safety verifier for ethereum smart contracts. In: IEEE. *2020 IEEE Symposium on Security and Privacy (SP)*. [S.l.], 2020. p. 1678–1694.

SRIPARASA, S. S. *JavaScript and JSON essentials*. [S.l.]: Packt Publishing Ltd, 2013.

STEFANOV, S. *Object-Oriented JavaScript*. [S.l.]: Packt Publishing Ltd, 2008.

SZABO, N. The idea of smart contracts, 1997. *URL http://szabo. best. vwh. net/smart_contracts_idea. html*, 1997.

TAN, S. et al. Latte: Visual construction of smart contracts. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. [S.l.: s.n.], 2020. p. 2713–2716.

TEIXEIRA, P. *Professional Node. js: Building Javascript based scalable software*. [S.l.]: John Wiley & Sons, 2012.

TORRES, C. F.; JONKER, H. et al. Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In: *International Symposium on Research in Attacks, Intrusions and Defenses, Limassol, Cyprus 26-28 October 2022*. [S.l.: s.n.], 2022.

TRAN, A. B.; LU, Q.; WEBER, I. Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In: *BPM (Dissertation/Demos/Industry)*. [S.l.: s.n.], 2018. p. 56–60.

TRAN, A. B. et al. Regerator: a registry generator for blockchain. In: *CAiSE-Forum-DC*. [S.l.: s.n.], 2017. p. 81–88.

VALENTE, M. T. Engenharia de software moderna (livro digital). 2020.

VUKOLIĆ, M. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In: SPRINGER. *International workshop on open problems in network security*. [S.l.], 2015. p. 112–125.

WEBER, I. et al. Untrusted business process monitoring and execution using blockchain. In: SPRINGER. *International Conference on Business Process Management*. [S.l.], 2016. p. 329–347.

WÖHRER, M.; ZDUN, U. Domain specific language for smart contract development. In: IEEE. *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. [S.l.], 2020. p. 1–9.

XU, Q. et al. Building an ethereum-based decentralized smart home system. In: IEEE. *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. [S.l.], 2018. p. 1004–1009.

ZHANG, P.; XIAO, F.; LUO, X. A framework and dataset for bugs in ethereum smart contracts. *arXiv preprint arXiv:2009.02066*, 2020.

ZHANG, Y. et al. Smartshield: Automatic smart contract protection made easy. In: IEEE. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2020. p. 23–34.

ZHU, Y. et al. Ta-spesc: Toward asset-driven smart contract language supporting ownership transaction and rule-based generation on blockchain. *IEEE Transactions on Reliability*, IEEE, v. 70, n. 3, p. 1255–1270, 2021.

Apêndice

A

TASKS DA AVALIAÇÃO DE USABILIDADE

Este Apêndice mostra de forma detalhada as *tasks* utilizadas no teste de usabilidade do Gifflar. Este é o mesmo conteúdo utilizado pelos participantes da avaliação para guiar o teste. A única diferença é que os participantes tiveram acesso a este conteúdo através de um repositório do GitHub.

A.1 TASK 1: CRIANDO GIFFLAR MODEL

Vamos aprender a criar um modelo do zero.

- Para isso, vá até a pasta `src/models` e crie um arquivo com o nome `MessageModel.ts`.
- Agora importe a fábrica para criação do GifflarContract:

```
import { createGifflarContract } from "@gifflar/solgen";
```

- Utilize a função importada para criar um novo Gifflar Contract e nomeie-o de `MessageContract`. Atribua o retorno para uma variável chamada `MessageContract`.

A.2 TASK 2: DESENVOLVENDO CONTRATO INTELIGENTE COM GIFFLAR CONTRACT MODEL

- Utilize um método do `MessageContract` para criar uma variável chamada `message`. Veja um exemplo de como seria a criação desta variável no Solidity:

```
// Escrito em solidity
string public message;
```

- Crie um evento chamado `MessageUpdated` que será emitido no momento em que a mensagem for alterada. Utilize o método do `MessageContract` responsável por criar eventos. Ele deve receber dois parâmetros: o valor antigo da mensagem e o seu novo valor. Exemplo no Solidity:

`// Escrito em solidity`

```
// Escrito em solidity
event MessageUpdated(string oldMessage, string newMessage);
```

- Crie um construtor para este contrato recebendo como parâmetro um valor inicial para a variável `message`. E como conteúdo, atribua o valor do atributo para a variável `message`. Exemplo no Solidity:

```
// Escrito em solidity
constructor(string memory _message) public{
  message = _message;
}
```

- Crie também uma função `setMessage` para a variável `message`, para permitir alterar o valor desta variável. Antes da alteração, crie uma variável local chamada `oldMessage` e atribua o valor de `message` para esta variável, lembre de definir a localização de dado `memory` na criação da variável `oldMessage`. Depois da alteração, emita o evento `MessageUpdated`. Exemplo no Solidity:

```
// Escrito em solidity
function setMessage(string memory _message) public{
  string memory oldMessage = message;
  message = _message;
  emit MessageUpdated(oldMessage, message);
}
```

- Agora que o contrato já está modelado, vamos exportar o modelo para que o ambiente Gifflar o encontre. Adicione a exportação no arquivo `MessageModel.ts`:

```
export default MessageContract;
```

A.3 TASK 3: CRIANDO GIFFLAR SCRIPT PARA IMPLANTAÇÃO DO CONTRATO NA BLOCKCHAIN

Vamos agora criar um script para realizar o deploy do contrato na rede blockchain.

- Crie um arquivo chamado `0_message.ts` na pasta `src/scripts`.
- Copie este conteúdo para dentro do arquivo:

```
import { IScriptFunctionInputs } from "types-gifflar/modules/commands/
    DeployContracts/dtos/IScriptFunctionInputs";

export default async ({ contracts }: IScriptFunctionInputs) => {
  const MessageContract = contracts["MessageContract"];

  // DEFINA AQUI O MÉTODO DE ESCRITA

  // DEFINA AQUI O MÉTODO DE COMPILAÇÃO

  // DEFINA AQUI O MÉTODO DE IMPLANTAÇÃO

  // DEFINA AQUI O MÉTODO QUE RETORNA A INSTÂNCIA DO CONTRATO NA REDE

  // COLE AQUI O RESTO DO CONTEÚDO DO SCRIPT
};
```

A.4 TASK 4: CONFIGURANDO ESCRITA E COMPILAÇÃO DO CONTRATO

Uma vez que estamos evitando a linha de comandos do Gifflar, teremos que escrever e compilar o código do contrato dentro deste script (normalmente seria utilizado um comando específico da linha de comandos do Gifflar).

- Utilize o Gifflar Contract `MessageContract` para chamar o método responsável por escrever o código do seu modelo de contrato abaixo de `DEFINA AQUI O MÉTODO DE ESCRITA`.
- Utilize o Gifflar Contract `MessageContract` para chamar o método responsável por compilar o seu modelo de contrato abaixo de `DEFINA AQUI O MÉTODO DE COMPILAÇÃO`.

A.5 TASK 5: CONSTRUINDO SCRIPT DE IMPLANTAÇÃO NA REDE BLOCKCHAIN TESTNET

Utilize o Gifflar Contract *MessageContract* para chamar o método responsável por implantar o contrato na rede. **Veja que este é um método assíncrono**.

- Utilize o endereço de carteira `"0xc49d80472ffa30a9a7b1c7b137dd05ff528f4e1d"` como o endereço blockchain que fará a implantação do contrato.
- Utilize como argumento do construtor do contrato a frase: `"Hello World!"`.
- Insira a chave privada da carteira no arquivo `gifflarconfig.json` em `mainAddressPrivateKey`, caso ainda não tenha.
- Agora vamos alterar a rede blockchain utilizada para a implantação do contrato. Veja que em `gifflarconfig.json`, a rede padrão (`defaultNetwork`) está como a rede local `local_network`. Perceba que a chave `networks` contém também a rede de teste da BSC (Binance Smart Chain) configurada, vamos selecioná-la alterando o valor de `defaultNetwork` para `bsc_testnet`.

Chame o método do `MessageContract` responsável por retornar a instância do contrato implantada na rede e salve dentro da variável chamada `contractInstance`.

Agora copie e adicione este código abaixo ao conteúdo do script em `0_message.ts` a fim de usar a instância de seu contrato da Web3 para alterar a mensagem.

```
if (contractInstance) {
  // Imprimindo endereço de contrato
  console.log(
    `Veja seu contrato em: https://testnet.bscscan.com/address/${
    contractInstance.options.address}`
  );

  // Capturando mensagem inicial
  const initialMessage = await contractInstance.methods.message().call
    ();
  console.log(`Mensagem inicial: ${initialMessage}`);

  // Alterando mensagem
  const receita = await contractInstance.methods
    .setMessage("Hello! I'm using Gifflar!")
    .send({
      from: "0xc49d80472ffa30a9a7b1c7b137dd05ff528f4e1d",
      gas: 3000000,
    });
  console.log(
    `Veja sua transação em: https://testnet.bscscan.com/tx/${receita.
    transactionHash}`
  );

  // Capturando mensagem alterada
  const updatedMessage = await contractInstance.methods.message().call
    ();
  console.log(`Mensagem alterada: ${updatedMessage}`);
}
```

Agora execute o comando abaixo no terminal para executar seu script:

```
$ gifflar deploy
```

Apêndice

B

EXEMPLO BASE DE CÓDIGO SOLIDITY

O Código B.1 foi utilizado na avaliação de usabilidade do Gifflar como código de exemplo para explicar alguns conceitos da linguagem Solidity para os participantes do experimento.

Código B.1 Código Solidity usado na avaliação de usabilidade do Gifflar como exemplo

```
pragma solidity 0.6.0;

// Exemplo de inicialização de contrato
contract Storage {
    // Exemplo de criação de variável de estado
    uint256 public number;

    // Exemplo de criação de evento
    event NumberStored(uint256 newNumber);

    // Exemplo de criação de construtor
    constructor() public {
    }

    // Exemplo de criação de função
    function store(uint256 num) public {
        // Exeplo de criação de variável local
        string memory testing;
        // Exemplo de atribuição
        testing = "1";
        number = num;
        // Exemplo de emissão de evento
        emit NumberStored(num);
    }

    // Exemplo de uma função que retorna um valor
    function retrieve() public view returns (uint256){
        return number;
    }
}
```

Apêndice

C

EXEMPLO DE CÓDIGO FINAL APÓS TASKS

Este Apêndice mostra um exemplo de como o código de cada participante do teste de usabilidade ficou ao finalizar as *tasks*. O Código C.1 mostra o código final esperado do modelo de contrato feito pelos participantes após finalizar a *task* 1 e 2 (ver Apêndice A). O Código C.2 demonstra como deve ser o código final do *script* de implantação criado pelos participantes na *task* 3, 4 e 5. E o Código C.3 apresenta como fica o arquivo de configuração final do Gifflar ao ser alterado na *task* 5 (somente a chave "defaultNetwork" é alterada).

Código C.1 Código exemplo de modelagem do contrato após *tasks*

```
import { createGifflarContract } from "@gifflar/solgen";

const MessageContract = createGifflarContract("MessageContract");

MessageContract.createVariable(
{ regularType: "string" },
"message",
"public"
);

MessageContract.createEvent("MessageUpdated", [
{ type: { regularType: "string" }, name: "oldMessage" },
{ type: { regularType: "string" }, name: "newMessage" },
]);

MessageContract.createConstructor(
[{ type: { regularType: "string" }, name: "_message" }]
)
.setAssignment("message", { customExpression: "_message" });
```

```
MessageContract.createFunction(
    "setMessage",
    "public",
    [{ type: { regularType: "string" }, name: "_message" }],
  )
  .setVariable({ regularType: "string" }, "oldMessage", {
    dataLocation: "memory",
    expressionValue: { customExpression: "message" },
  })
  .setAssignment("message", { customExpression: "_message" })
  .setEventCall("MessageUpdated", ["oldMessage", "message"]);

export default MessageContract;
```

Código C.2 Código exemplo do script de implantação após *tasks*

```
import { IScriptFunctionInputs } from "types-gifflar/modules/commands/
    DeployContracts/dtos/IScriptFunctionInputs";

export default async ({ contracts }: IScriptFunctionInputs) => {
  const MessageContract = contracts["MessageContract"];

  // DEFINA AQUI O MÉTODO DE ESCRITA
  MessageContract.write()

  // DEFINA AQUI O MÉTODO DE COMPILAÇÃO
  MessageContract.compile()

  // DEFINA AQUI O MÉTODO DE IMPLANTAÇÃO
  await MessageContract.deploy(
    {
      args: ["Hello World!"],
      from: "0xc49d80472ffa30a9a7b1c7b137dd05ff528f4e1d",
    }
   );

  // DEFINA AQUI O MÉTODO QUE RETORNA A INSTÂNCIA DO CONTRATO NA REDE
  const contractInstance = MessageContract.deployed()

  // COLE AQUI O RESTO DO CONTEÚDO DO SCRIPT
  if (contractInstance) {
    // Imprimindo endereço de contrato
    console.log(
      "Veja seu contrato em: https://testnet.bscscan.com/address/${
    contractInstance.options.address}"
    );

    // Capturando mensagem inicial
    const initialMessage = await contractInstance.methods.message().
    call();
    console.log("Mensagem inicial: ${initialMessage}");

```

```
34      // Alterando mensagem
35      const receita = await contractInstance.methods
36        .setMessage("Hello! I'm using Gifflar!")
37        .send({
38          from: "0xc49d80472ffa30a9a7b1c7b137dd05ff528f4e1d",
39          gas: 3000000,
40        });
41      console.log(
42        "Veja sua transação em: https://testnet.bscscan.com/tx/${receita.
      transactionHash}"
43      );
44
45      // Capturando mensagem alterada
46      const updatedMessage = await contractInstance.methods.message().
      call();
47      console.log("Mensagem alterada: ${updatedMessage}");
48    }
49  };
```

Código C.3 Código exemplo de arquivo de configuração do Gifflar após finalizar as *tasks*

```
1  {
2   "projectName": "Gifflar Project",
3   "root": "./src",
4   "modelsFolder": "./src/models",
5   "contractsFolder": "./src/contracts",
6   "servicesFolder": "./src/services",
7   "compileFolder": "./src/arctifacts",
8   "scriptsFolder": "./src/scripts",
9   "appKey": "77f91915f2ce0bc190e66...",
10  "defaultNetwork": "bsc_testnet",
11  "mainAddressPrivateKey": "5f2e4a18feb1c62da59e8c7f2ad05...",
12  "networks": [
13    {
14      "key": "local_network",
15      "networkId": 0,
16      "gas": 3000000,
17      "gasPrice": "10000000000",
18      "nodeLink": "http://localhost:8545"
19    },
20    {
21      "key": "bsc_testnet",
22      "networkId": 97,
23      "gas": 3000000,
24      "nodeLink": "https://data-seed-prebsc-1-s1.binance.org:8545/"
25    }
26  ]
27 }
```

Apêndice

D

TABELAS COMPLETAS DE RESULTADOS DA AVALIAÇÃO DE USABILIDADE

Este apêndice apresenta de forma completa os dados obtidos da análise de usabilidade do Gifflar. A Tabela D.1 apresenta os resultados das respostas das entrevistas com cada participante aplicando uma porcentagem mais precisa com duas casas decimais. A Tabela D.2 apresenta todos os tokens de usabilidade detectados durante o teste realizado por cada participante. Esta última tabela referencia as questões de pesquisa (QPs) apresentadas a Seção 4.1.1 do Capítulo 4.

Tabela D.1 Respostas das perguntas da entrevista (versão detalhada).

Q	**Sim**		**Não**		**Algumas vezes**	
	#	%	#	%	#	%
1	15	68,18%	0	0,00%	7	31,82%
2	10	45,46%	4	18,18%	8	36,36%
3	21	95,46%	1	4,56%	0	0,00%
4	22	100%	0	0,00%	0	0,00%
5	0	0,00%	22	100%	0	0,00%
6	0	0,00%	21	95,46%	1	4,56%
7	0	0,00%	19	86,36%	3	13,64%
8	22	100%	0	0,00%	0	0,00%
9	1	4,56%	16	72,73%	5	22,73%
10	0	0,00%	22	100%	0	0,00%
11	21	95,46%	0	0,00%	1	4,56%
12	0	0,00%	22	100%	0	0,00%

Tabela D.2 Tokens de usabilidade mais frequentes (versão detalhada).

Q	Token	Descrição	QPs	#	%
T1	S	não acostumado a esse tipo de doc.	1	17	77%
T2	S	tipo de dado incomum.	1, 2	13	59%
T3	N	não encontrado sobre 'memory'.	1, 2	12	55%
T4	R	'dataLocation' ao criar variável local não entendido.	1, 2	8	36%
T5	S	esperado 'setEmit' não 'setEventCall'.	1	8	36%
T6	R	exemplo do padrão builder não está claro.	1, 2	7	32%
T7	C	definir '*inputs*' por argumento ou método 'setInput'.	1, 3	5	23%
T8	S	fábrica é um objeto?	1, 2	4	18%
T9	N	método 'deploy' não encontrado.	1	4	18%
T10	S	criar a variável pelo TypeScript.	1, 2	4	18%
T11	I	criar variável local nos *inputs* de 'createFunction'.	1, 2, 4	4	18%
T12	I	tentou criar variável definindo somente o nome.	1	4	18%
T13	U	atribuição depois de criação de variável.	1, 4	3	14%
T14	S	achou que 'createVariable' cria variável local.	1	3	14%
T15	R	difícil encontrar método 'deployed'.	1	3	14%
T16	I	tentou criar o construtor pelo TypeScript.	1, 2	3	14%
T17	N	método 'deployed' não encontrado.	1	3	14%
T18	I	criar uma variável dentro dos *outputs* de uma função.	1, 2, 4	2	9,09%
T19	M	esqueceu como definir uma atribuição.	4	2	9,09%
T20	S	dúvida se 'setVariable' cria uma variável local.	1	2	9,09%
T21	N	'createFunction' não encontrado, neste caso a task não foi clara.	1	2	9,09%
T22	I	tentou usar o 'setAssignment' para criar uma variável local.	1, 4	2	9,09%
T23	S	devo instanciar o MessageContact?	1, 2	1	4,56%
T24	C	opção 'setInput' não foi entendido.	1	1	4,56%
T25	S	a propriedade 'customExpression' é realmente necessária?	1, 2	1	4,56%
T26	I	tentou atribuir nos *inputs* do construtor.	1, 2, 4	1	4,56%
T27	I	usou 'setInput' para definir uma atribuição.	1, 4	1	4,56%