MATHEUS HENRIQUE CARVALHO DE PAIVA RESENDE

TSARCH: ANÁLISE DE CONFORMIDADE ARQUITETURAL PARA TYPESCRIPT

LAVRAS – MG

2023

MATHEUS HENRIQUE CARVALHO DE PAIVA RESENDE

TSARCH: ANÁLISE DE CONFORMIDADE ARQUITETURAL PARA TYPESCRIPT

Monografia apresentada à Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel.

Prof. DSc. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2023

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos
da Biblioteca Universitária da UFLA

Resende, Matheus Henrique Carvalho de Paiva
TSArch: Análise de conformidade arquitetural para TypeScript / Matheus Henrique Carvalho de Paiva Resende. 2ª ed. rev., atual. e ampl. – Lavras : UFLA, 2023.
41 p. : il.

Monografia (graduação)–Universidade Federal de Lavras, 2023.
Orientador: Prof. DSc. Ricardo Terra Nunes Bueno Villela.
Bibliografia.

1. Arquitetura de Software, 2. Análise estática, 3. Linguagem de Programação
1. TCC. 2. Monografia. 3. Dissertação. 4. Tese. 5. Trabalho Científico – Normas. I. Universidade Federal de Lavras. II. TSArch: Análise de conformidade arquitetural para TypeScript.

MATHEUS HENRIQUE CARVALHO DE PAIVA RESENDE

TSARCH: ANÁLISE DE CONFORMIDADE ARQUITETURAL PARA TYPESCRIPT

Monografia apresentada à Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel.

APROVADA em 27 de Outubro de 2023.

Prof. DSc. Maurício Ronny de Almeda Souza	UFLA
BSc. Eduardo Fernando de Lima	UFLA

Prof. DSc. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2023

AGRADECIMENTOS

Gostaria de agradecer minha família e amigos que estiveram comigo durante toda a graduação, que me ajudaram a manter o foco e não desistir. Em especial, gostaria de agradecer a minha mãe, Gláucia, que possibilitou minha dedicação exclusiva à graduação e me inspirou a sonhar e correr atrás dos meus sonhos.

RESUMO

Com a alta demanda e prazos curtos de entrega, é difícil manter boas práticas e respeitar as restrições da arquitetura planejada, o que favorece o processo de erosão arquitetural. TypeScript oferece ao desenvolvedor um ecossistema de desenvolvimento rápido com suporte aos mais modernos recursos de linguagem. Entretanto, tantos recursos tornam a análise arquitetural da linguagem um desafio. Diante desse cenário, este trabalho discorre sobre os desafios da análise arquitetural sobre a linguagem e propõe o *TSArch*, uma ferramenta que realiza verificação da arquitetura a partir de um conjunto de regras especificadas pelo arquiteto. A ferramenta, além de prover um relatório com as violações arquiteturais detectadas, provê visualizações arquiteturais em grafo e em DSM.

Palavras-chave: Arquitetura de Software, Análise estática, Linguagem de Programação.

LISTA DE FIGURAS

Figura 2.1 – Ciclo de vida do código Java 11
Figura 2.2 – Ciclo de vida do código TypeScript 11
Figura 3.1 – Visão geral do TSArch 14
Figura 3.2 – Arquitetura planejada da aplicação *Corrector.ts* 16
Figura 3.3 – Exemplo de estrutura de diretórios 17
Figura 3.4 – Grafo resultante da análise de conformidade produzido pela ferramenta TSArch . . . 22
Figura 3.5 – DSM resultante da análise de conformidade produzida pela ferramenta TSArch . . . 23
Figura 4.1 – Representação do fluxo de comunicação entre os módulos da ferramenta 24
Figura 4.2 – DSM resultante da execução da ferramenta sobre o próprio código fonte 32
Figura 4.3 – DSM resultante da execução da ferramenta sobre o próprio código fonte 32

LISTA DE CÓDIGOS

Código 3.1 – Exemplo de arquivo de especificação 17
Código 3.2 – Especificação do *Corrector.ts* 18
Código 3.3 – Exemplo de retorno da análise de conformidade 21
Código 4.1 – Exemplo de utilização da linha de comando da ferramenta TSArch 25
Código 4.2 – Exemplo de inferência de tipos 25
Código 4.3 – Conjunto de regras antes da transformação 26
Código 4.4 – Conjunto de regras depois da transformação 26
Código 4.5 – Exemplo de inferência de tipos 27
Código 4.6 – Exemplo de dependência sem importação direta 28
Código 4.7 – Regras para o exemplo de dependência sem importação direta 29
Código 4.8 – Regras arquiteturais da ferramenta 31

SUMÁRIO

1 INTRODUÇÃO 8

2 CARACTERÍSTICAS DA LINGUAGEM TYPESCRIPT 10

3 SOLUÇÃO PROPOSTA 14

3.1 Projeto Exemplo 15

3.2 *Parsing* das regras 16

3.3 *Parsing* do código fonte 19

3.4 Análise da conformidade 19

3.5 Resultados e visualização 21

4 PROJETO E IMPLEMENTAÇÃO 24

4.1 Entrada de dados 24

4.2 Processamento das regras 25

4.3 Processamento do código-fonte 26

4.4 Análise de conformidade 29

4.5 Criação de artefatos 30

5 Ferramentas relacionadas 33

6 Considerações finais 36

REFERÊNCIAS 38

1 INTRODUÇÃO

Definição de arquitetura é uma etapa comum do ciclo de vida do software, estabelecendo regras e padrões cuja equipe de desenvolvimento deverá seguir durante os demais passos (SHYLESH, 2017). Essa etapa é fundamental para garantir o crescimento saudável e apropriado do software, evitando surgimento de dívidas técnicas, simplificando a criação de testes e simplificando o processo de depuração, já que a estrutura de comunicação das interfaces já estaria previamente definida (PERRY; WOLF, 1992). O aquecimento do mercado de tecnologia, aumentando a rotatividade das equipes de desenvolvimento, a corrida pelo *time to market* e a diferença de conhecimento técnico e de negócio entre os membros da equipe aceleram o processo de erosão arquitetural, caracterizado pela diferença entre a arquitetura implementada e planejada (LI et al., 2022).

Com a popularização da cultura *DevOps*, desenvolvedores de software passaram a ter mais contato e preocupação com processos de qualidade, testes e distribuição do software produzido (AMAZON, 2022). Esse movimento cria demanda de ferramentas capazes de validar e formatar código sobre padrões pré-definidos. Com o mercado receptivo para novas ferramentas que contribuam com a qualidade do código, tem-se a ideia de desenvolver ferramentas de análise de conformidade arquitetural.

A linguagem JavaScript, atualmente, é a segunda linguagem mais utilizada (ZAPONI, 2022). Sua fama advém da possibilidade de ser utilizada tanto no lado dos clientes quanto no lado dos servidores e da praticidade de desenvolver sobre a mesma criando *supersets* e variações que são transpiladas para o JavaScript puro (MICROSOFT, 2022c). TypeScript e *React* são dois casos onde isto acontece. Por um lado, *React* utiliza uma variação do JavaScript que permite inserir código HTML ao lado do código JavaScript, conhecido como JSX (*JavaScript Syntax Extension*) para JavaScript e TSX (*TypeScript Syntax Extension*) para TypeScript. Por outro lado, TypeScript é um *superset* que adiciona a possibilidade de utilizar anotação de tipos convertendo para JavaScript nativo utilizando processo de compilação (MICROSOFT, 2022c; FACEBOOK, 2022a).

Portanto, este trabalho sugere a criação de uma ferramenta de análise estática de código cujo objetivo é identificar desvios arquiteturais a partir do código fonte e uma descrição de alto nível da arquitetura proposta para o sistema de software, definindo quais são os módulos e como se relacionam quanto à permissão (*allowed* e *forbidden*) e quanto à obrigatoriedade (*required*). A ferramenta provê um relatório textual apontando cada desvio arquitetural e gera um grafo direcionado (ESSAM; FISHER, 1970) e

uma DSM (*Dependency Structure Matrix*) (SULLIVAN et al., 2001) que ilustram as dependências entre os módulos com o diferencial de destacar os desvios arquiteturais encontrados.

A ferramenta busca contribuir com a manutenção da arquitetura dos sistemas de software permitindo encontrar desvios antes que sejam integrados. Isso permite corrigir os desvios antes que causem maiores impactos no sistema e, portanto, reduzindo custos para correção (BRYKCZYNSKI; MEESON; WHEELER, 1994), além de proporcionar oportunidades para evoluir o projeto arquitetural corrente.

Este trabalho está organizado como a seguir. A Seção 2 expõe as dificuldades de se analisar estaticamente um projeto escrito em TypeScript do ponto de vista arquitetural. A Seção 3 traz uma visão geral da solução proposta incluindo detalhamento de como é feita a especificação da arquitetura, de como é feita a inferência de tipos e de como as violações são detectadas e visualizadas. A Seção 4 detalha o projeto e implementação com viés mais técnico. A Seção 5 reporta ferramentas relacionadas. Por fim, Seção 6 traz as considerações finais e trabalhos futuros.

2 CARACTERÍSTICAS DA LINGUAGEM TYPESCRIPT

Segundo Microsoft (2022c), TypeScript trata de um complemento (*superset*) da linguagem JavaScript que busca adicionar anotação e verificação de tipos em tempo de compilação além de funcionalidades do paradigma orientado a objetos. O intuito por trás da criação do *superset* é trazer para o JavaScript um conjunto de técnicas e estratégias de desenvolvimento de software existentes em outras linguagens, simplificando a manutenção e crescimento do software.

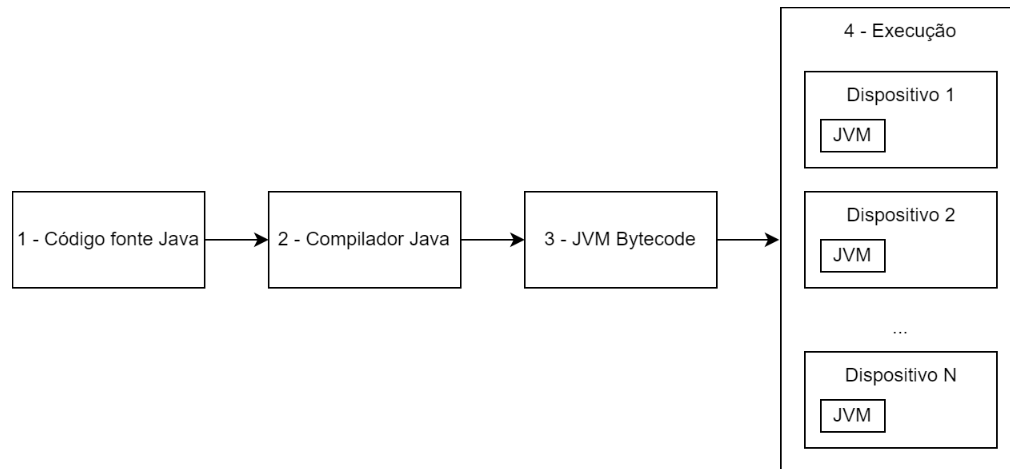
No contexto da análise arquitetural, a linguagem fornece uma série de características que trazem complexidades. Por exemplo, paradigma, *design*, *runtimes*, sintaxe e recursos, dificultando a identificação de componentes que poderiam indicar dependências.

Black (2020) pontua a linguagem como multi-paradigma, podendo ser escrita no funcional, no imperativo e no orientado a objetos. No paradigma imperativo, a atenção é direcionada a variáveis e funções. O paradigma funcional traz uma abordagem diferente, orientada a funções adicionando conceitos de funções de alta ordem, *callbacks* e funções parciais. O paradigma orientado a objetos, por sua vez, insere herança e polimorfismo na equação (SEBESTA, 2005).

Uma das características da linguagem herdada do JavaScript é a indiferenciação entre funções e variáveis. Ou seja, uma função nada mais é do que um objeto “chamável” (*callable*) (MOZILLA CORPORATION, 2022a). Tal característica está presente em todos os paradigmas utilizados pela linguagem incluindo orientado a objetos onde métodos também podem ser observados como “atributos chamáveis”. Essa característica permite a nomeação de funções anônimas.

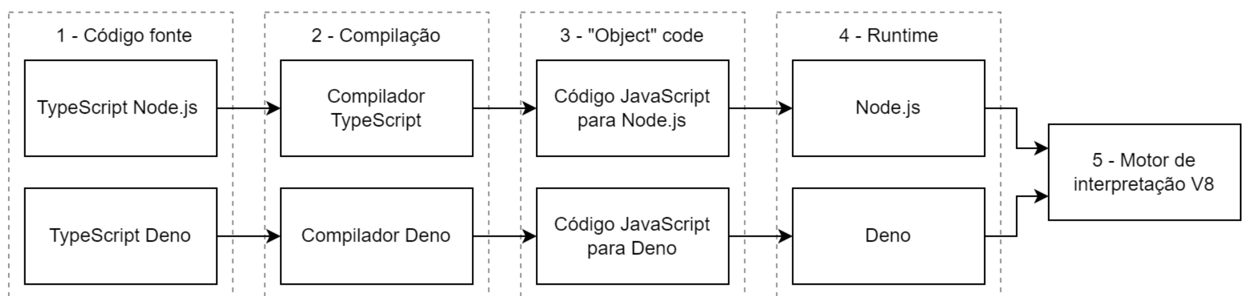
Semelhante à linguagem Java, que é compilada para um *bytecode* executável por uma máquina virtual, TypeScript é compilado para JavaScript e executado em um interpretador conhecido como *runtime* (ORACLE, 2022; MICROSOFT, 2022c) (Figura 2.1). Outra característica comum é a existência de diferentes *runtimes* para execução do código. No entanto, ao contrário do que é visto em Java onde o mesmo *bytecode* pode ser utilizado em diferentes máquinas virtuais sem alterar o funcionamento, o *runtime* utilizado altera a perspectiva a qual o código é escrito e utilizado (Figura 2.2). Os *runtimes* mais conhecidos são Node.js e Deno.

Figura 2.1 – Ciclo de vida do código Java



Fonte: Do autor (2023)

Figura 2.2 – Ciclo de vida do código TypeScript



Fonte: Do autor (2023)

No caso do Node.js, as dependências externas são armazenadas de forma local no projeto, no diretório *node_modules*, obtidos durante o desenvolvimento e referenciados no código pelo caminho absoluto (OPENJS FOUNDATION, 2022). Enquanto isso, no *runtime* Deno, não existe um diretório local para dependências externas, mas sim, um *cache* global que é preenchido em tempo de execução (DENO, 2022a). A referência para o módulo é feita diretamente para o repositório da biblioteca a partir de uma URL, assim como é feito em *GoLang* (GUIDES, 2020). No que tange o ecossistema entre os *runtimes*, não é possível garantir interoperabilidade entre as bibliotecas desenvolvidas em *runtimes* diferentes devido a características específicas. Assim, a implementação do módulo *Express* tradicional do Node.js não funcionaria adequadamente sem devidas camadas de compatibilidade (DENO, 2022b).

TypeScript possui uma comunidade ativa que está sempre contribuindo para evolução do ecossistema. Uma das formas de contribuição está na criação de complementos para a linguagem que modificam

a escrita de código adicionando novas funcionalidades. Entretanto, tais modificações utilizam *transpiladores* para converter o código escrito com o complemento para o código TypeScript compilável. *React* é um exemplo de tecnologia que utiliza outros compiladores e transpiladores auxiliares para alterar a sintaxe do código, permitindo escrever código HTML ao lado de código TypeScript (FACEBOOK, 2022b).

Ao contrário de outras linguagens de programação que permitem a organização e separação de arquivos em pacotes como Java (STRNIŠA; SEWELL; PARKINSON, 2007) e Python (PYTHON, 2023), JavaScript não possui um sistema de módulos em seu *design* inicial. Por consequência, TypeScript herda essa limitação (KANG; RYU, 2012). Essa necessidade veio ao longo do tempo com o aumento da complexidade das aplicações e foi potencializada com a popularização do interpretador V8 e *runtime Node.js* fomentando o interesse de se utilizar JavaScript do lado servidor. Com isto, a comunidade implementou os sistemas de módulos *CommonJS* (DANGOOR, 2010) e, futuramente, *ECMAScript Modules (ESModules)* (CLARK, 2018) como solução para esse problema.

CommonJS veio como uma solução funcional simples que trata os módulos importados como variáveis (NODEJS, 2023a) enquanto *ESModules*, que foi lançado futuramente, trouxe uma sintaxe própria mais clara por deixar explícito a importação e exportação (NODEJS, 2023b). Após aceitação, a sintaxe do *ESModules* veio a se tornar o padrão desejado da Indústria, sendo, inclusive, adotada pelo TypeScript nativo e no dialeto utilizado pelo Deno (MICROSOFT, 2022a; MOZILLA CORPORATION, 2022b).

Black (2020) e Microsoft (2022c) apresentam a linguagem como gradualmente tipada, o que garante uma maior adesão e permite uma transição gradual do código previamente escrito em JavaScript para *TypeScript*. Para isto, o compilador da linguagem traz opções de configuração que ditam como o processo de compilação deve ser feito e o quão permissível deve ser quanto à anotação de tipos (MICROSOFT, 2022d).

Um dos efeitos colaterais da falta de um sistema de módulos nativo está na possibilidade de utilizar um recurso sem sua devida importação. Suponha um módulo **`A`** chamando uma função de um módulo **`B`** que retorna um objeto **`x`** de um módulo **`C`**. O módulo **`A`** irá manipular – normalmente – o objeto **`x`** mesmo sem nenhuma referência explícita ao módulo **`C`**.

Considerando os elementos supracitados, pode-se observar a dificuldade para identificar os componentes dentro da flexibilidade da linguagem e, portanto, dificultando o estabelecimento das relações entre as partes do código, o que é essencial para a análise de conformidade arquitetural. Portanto, de modo a ter uma primeira versão utilizável, a ferramenta se propõe a analisar projetos escritos utilizando

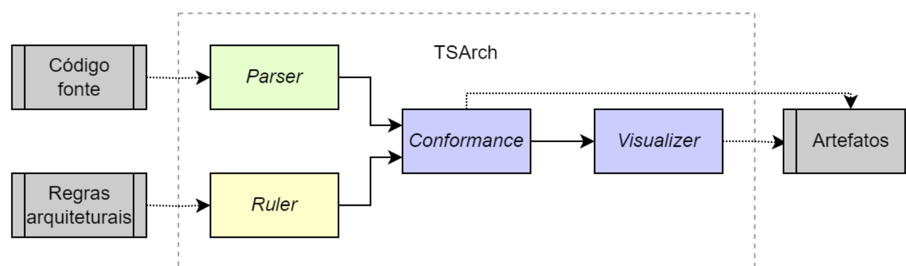
paradigmas imperativo e orientado a objetos, sem utilização de outros complementos que alteram a sintaxe do código. Isso inclui a identificação de relações, mesmo que implícitas, entre as partes do sistema de modo a estabelecer sua arquitetura.

3 SOLUÇÃO PROPOSTA

A solução proposta se baseia em um analisador estático implementado em TypeScript capaz de identificar desvios arquiteturais a partir do *parsing* do código fonte. Seu objetivo consiste em fornecer a equipes de desenvolvimento uma forma de monitorar a erosão da arquitetura a partir de relatórios e representações gráficas.

A Figura 3.1 representa a relação de sequência entre as etapas do processamento:

Figura 3.1 – Visão geral do TSArch



Fonte: Do autor (2023)

1. *Parsing* do código fonte, populando a tabela de símbolos (*Parser*);
2. *Parsing* do conjunto de regras (*Rules*);
3. Análise de conformidade (*Conformance*); e
4. Visualização da arquitetura e seus desvios (*Visualizer*).

Os artefatos produzidos podem ser utilizados no contexto de integração e entrega contínua do software (*Continuous Integration and Continuous Delivery* CI/CD), onde pode agir como gatilho de qualidade bloqueando integrações que não estão adequadas à arquitetura (FOWLER; FOEMMEL, 2006). Com isso, é possível identificar defeitos relacionados à arquitetura ainda em tempo de desenvolvimento, o que reduz custos relacionados a correção (BRYKCZYNSKI; MEESON; WHEELER, 1994). Outro ponto forte da ferramenta está na visualização da arquitetura destacando pontos de erosão da arquitetura.

3.1 Projeto Exemplo

O projeto *Corrector.ts* – desenvolvido pelo autor deste trabalho de conclusão de curso como exemplo para este estudo – consiste de uma simples aplicação responsável por corrigir provas e produzir estatísticas sobre os resultados obtidos. São recebidos por linha de comando o caminho para o arquivo que contém uma relação entre os alunos e suas respectivas respostas e outro caminho para um arquivo que contenha as respostas para a prova em questão. Após isso, a partir dos resultados, são calculadas as estatísticas. E, por fim, os dados são persistidos em um "S3 Bucket". A arquitetura planejada para esse projeto consiste em oito módulos seguindo uma estratégia não estritamente em camadas:

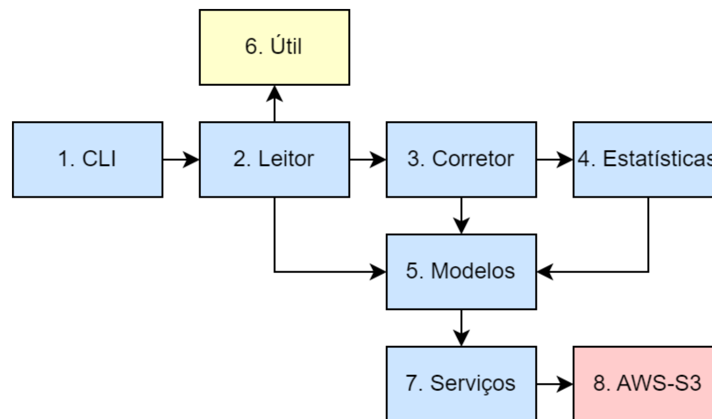
1. **`Modelos`**: Estruturas utilizadas para persistência e manipulação dos dados.
2. **`Serviços`**: Adaptadores que atuam como intermediário entre serviços externos e a regra de negócio da aplicação.
3. **`Estatísticas`**: Responsável por manipular os dados fornecidos e produzir novas informações.
4. **`Corretor`**: Conjunto de funções responsáveis por comparar respostas de uma prova com um dado gabarito.
5. **`Leitor`**: Responsável por interpretar os dados contidos no arquivo convertendo os dados para a estrutura de dados do programa.
6. **`CLI`**: Responsável por identificar o formato do arquivo recebido e seus caminhos. Possui um subdiretório chamado "Comandos" onde são especificados os subcomandos e parâmetros.
7. **`AWS-S3`**: Módulo abstrato que representa as bibliotecas externas utilizadas para comunicar com o serviço de armazenamento da AWS.
8. **`Útil`**: Funções auxiliares para processamento de datas.

A Figura 3.2 ilustra a arquitetura planejada da aplicação. O módulo **`CLI`** consiste em arquivos que definem os comandos utilizados utilizando a biblioteca *Yargs*[1]. Esse módulo, por sua vez, aponta

[1] Yargs é uma biblioteca que auxilia a criação de ferramentas de linhas de comando simplificando o *parsing* dos argumentos recebidos (YARGS, 2022).

para o leitor de arquivos adequado para converter para os modelos definidos no módulo **`Modelos`**. O **`Corretor`** é responsável por corrigir cada questão das provas a partir do gabarito. Em seguida à correção, o módulo **`Estatísticas`** realiza o cálculo de métricas. O módulo **`Modelos`**, além de definir as interfaces utilizadas, é utilizado para persistência dos dados após todas as análises. O módulo **`Serviços`** trazem implementações das funções de apoio que atuam como adaptadores para serviços externos. No módulo **`Útil`**, estão as funções não relacionadas necessariamente ao projeto, nesse contexto, relacionadas a processamento de datas.

Figura 3.2 – Arquitetura planejada da aplicação *Corrector.ts*



Fonte: Do autor (2023)

3.2 *Parsing* das regras

A especificação da arquitetura é feita por meio de um arquivo no formato JSON (*JavaScript Object Notation*). Essa escolha foi feita justamente por ser um formato amplamente adotado na comunidade JavaScript por se basear na notação de objetos da própria linguagem (JACKSON, 2016). Outro motivo que fomentou a escolha foi a praticidade da construção da especificação por se tratar de um arquivo texto estruturado de fácil leitura dispensando a utilização de outros recursos computacionais e ser mais simples de se escrever comparado com linguagens de marcação como XML (*Extensible Markup Language*). Como pode ser visto no Código 3.1, o arquivo consiste de um objeto global anônimo que acolhe a definição dos módulos que consistem em um objeto nominado, o qual dá nome o módulo em questão ("*ModuleID*"), onde seus atributos declaram os arquivos ou bibliotecas que integram o módulo (*files* e *packages*), linhas 3 e 4 do exemplo, os arquivos ou módulos permitidos ou não (*allowed* e *forbidden*), linha 5, e os arquivos ou módulos cuja dependência é obrigatória (*required*), linha 6.

Código 3.1 – Exemplo de arquivo de especificação

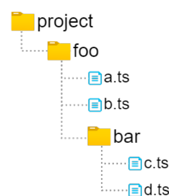
```
1 {
2   "ModuleID": {
3     "files": [Arquivos],
4     "packages": [Bibliotecas externas]
5     "forbidden" ou "allowed": [Arquivos ou módulos], // mutuamente exclusivos
6     "required": [Arquivos ou módulos], // opcional
7   }
8 }
```

Fonte: Do autor (2023)

Diante da falta de um sistema de módulos, a sintaxe das regras leva em questão diretamente a disposição dos arquivos na estrutura de diretórios para simplificar a especificação. É fortemente recomendado que os módulos definidos não utilizem apenas parte dos arquivos contidos no diretório, mas sim, sua totalidade aproveitando os *wildcards* demarcados pelos símbolos * e **. Semelhante a sua utilização dentro do contexto das expressões regulares, os * especificam se são considerados os diretórios aninhados ou apenas os arquivos contidos no diretório especificado.

Por um lado, tendo como base a estrutura de diretórios da Figura 3.3, os arquivos referentes a regra "./foo/*" são estritos àquele diretório, não considerando seus subdiretórios. Por outro lado, a regra "./foo/**" considera todos os seus subdiretórios e demais arquivos daquele ramo. Ou seja, considerando a regra "./foo/*", apenas os arquivos "./foo/a.ts" e "./foo/b.ts" fariam parte do módulo declarado enquanto a regra "./foo/**" contempla também "./foo/bar/c.ts" e "./foo/bar/d.ts".

Figura 3.3 – Exemplo de estrutura de diretórios



Fonte: Do autor (2023)

O campo *package*, por sua vez, não considera subdivisões do mesmo. Se uma biblioteca é subdividida em outras partes, apenas o que está declarado explicitamente é considerado. Por exemplo, na linha 32 do Código 3.2, apenas "@aws-sdk/client-s3" será considerado. Portanto, "@aws-sdk/client-s3/foo", por exemplo, não pertenceria ao módulo. Para fins de simplificação, todas as bibliotecas externas não declaradas em um módulo – por padrão – podem ser utilizadas em qualquer parte do código.

Os campos *allowed* e *forbidden* são mutuamente exclusivos, portanto, não podem ser utilizados de forma simultânea. Por padrão, a não declaração de nenhuma dessas propriedades considera que o módulo em questão não pode depender de nenhum outro. Quando um módulo, arquivo ou biblioteca ocorre no campo *required* automaticamente implica em ser permitido. É importante mencionar que o *parser* das regras não valida a coerência da especificação, logo uma especificação mal feita pode levar a resultados inconclusivos.

Código 3.2 – Especificação do *Corrector.ts*

```
1  {
2    "CLI": {
3      "files": ["./src/cli/**"],
4      "allowed": [],
5      "required": ["Leitor"]
6    },
7    "Corretor": {
8      "files": ["./src/corretor/*"],
9      "forbidden": ["CLI", "AWS-S3"]
10   },
11   "Estatisticas": {
12     "files": ["./src/estatisticas/*"],
13     "allowed": ["Modelos"]
14   },
15   "Leitor": {
16     "files": ["./src/leitor/*"],
17     "allowed": ["Modelos", "Util"],
18     "required": ["Corretor"]
19   },
20   "Modelos": {
21     "files": ["./src/modelos/*"]
22   },
23   "Util": {
24     "allowed": [],
25     "files": ["./src/util/**"]
26   },
27   "Servicos": {
28     "files": ["./src/servicos/**"],
29     "Allowed": ["AWS-S3"]
30   },
31   "AWS-S3": {
32     "packages": ["@aws-sdk/client-s3"]
33   }
34 }
```

Fonte: Do autor (2023)

O Código 3.2 trata da especificação da arquitetura do exemplo descrito na Seção 3.1 utilizando a estrutura descrita anteriormente. Como pode ser observado, o módulo **`Modelos`** definido nas linhas 20 a 22, não possui regras do tipo *allowed* e *forbidden*, logo a ferramenta interpreta como tudo proibido. O módulo **`CLI`** definido nas linhas 2 a 6 possui subdiretórios e é razoável para o contexto da aplicação considerar todos os demais arquivos nos subdiretórios. Os demais módulos não possuem subdiretórios,

logo, nesse cenário específico, é indiferente a utilização de * ou **. Embora indiferente, utiliza-se * por ser mais restritivo caso novos diretórios sejam criados e requeiram alguma evolução arquitetural.

3.3 *Parsing* do código fonte

Como elucidado na Seção 2, a complexidade intrínseca da linguagem TypeScript levou a reduzir o escopo inicial da ferramenta. Por isso, ainda não são contempladas as características relacionadas a paradigmas que não seja imperativo ou orientado a objetos. Ou seja, a análise foi reduzida a declaração de tipos (classes, interfaces, enumerativos e *type alias*), objetos (atributos, métodos, funções e variáveis) e importações.

Essa parte da ferramenta é responsável por encontrar os marcadores de dependências destacando todas as informações necessárias para a análise de conformidade. Para isso, percorre-se a árvore sintática, obtida pela compilação do código, buscando por estruturas de "importação", declaração de tipos e variáveis.

Como TypeScript não obriga a declaração de tipos em todos os contextos, neste trabalho também é feita a inferência dos tipos utilizando estruturas de verificação de tipos obtidas pela compilação do código utilizando a opção *strict*, ou seja, deve respeitar as regras mais severas quanto a tipagem. Por exemplo, ao declarar uma função, seus parâmetros devem declarar explicitamente o tipo *any* caso permita receber qualquer tipo como aquele parâmetro. Outro exemplo está relacionado ao uso de *undefined* e *null* que devem respeitar seus próprios tipos.

3.4 Análise da conformidade

Com os resultados do *parsing* das regras e do código fonte, pode então ser feita uma comparação entre as dependências permitidas no arquivo de regras e as reais utilizadas no projeto identificando os seguintes casos:

- *Convergência*: Quando um arquivo de um módulo **`A`** depende de um arquivo do módulo **`B`** de forma direta ou indireta e (i) existe uma regra que permite essa dependência ou (ii) não existe regra explícita que proíba a dependência.

- *Divergência*: Quando um arquivo de um módulo **A** depende de um arquivo de um módulo **B** de forma direta ou indireta, (i) mas não existe regra permitindo essa dependência ou (ii) está explícita a proibição dessa dependência.
- *Ausência*: Existe um arquivo de um módulo **A** que não possui dependência com um arquivo do módulo **B** sendo que existe uma regra que obriga que **A** dependa de **B**.
- *Alerta*: Existe uma regra que permite (mas não obriga) que arquivos do módulo **A** dependam de arquivos do módulo **B** mas nenhum arquivo de **A** depende de algum arquivo de **B**. Não se trata de um desvio mas pode apontar falhas da especificação.

Para exemplificar, foram inseridas quatro violações no exemplo definido na Seção 3.1 de forma arbitrária de modo a simular situações reais possíveis.

- **Ausência**: É criado um novo arquivo no módulo CLI com objetivo de definir um comando para leitura de arquivos JSON. Entretanto, tal arquivo não utiliza um leitor definido no módulo **Leitor** para processar o arquivo JSON fornecido, o que quebra a regra onde **CLI** deve depender de **Leitor**. A linha 5 do Código 3.2 especifica essa regra.
- **Divergência**: Durante a criação do novo comando para leitura de arquivos JSON, é importado o modelo “Prova”, o que quebra a regra que proíbe um arquivo do módulo **CLI** depender de um arquivo do módulo **Modelos**. A linha 4 do Código 3.2 especifica todos os módulos cuja dependência é permitida, porém **Modelos** não está na lista.
- **Divergência**: Durante a correção é produzido um resultado intermediário que é persistido em um “S3 Bucket”. A correção é realizada no módulo **Corretor** e a persistência é feita utilizando uma biblioteca descrita no módulo **AWS-S3** o que é explicitamente proibido na linha 9 do Código 3.2
- **Alerta**: Algumas funções são comumente utilizadas em vários módulos e não representam regras de negócio da aplicação, por isso, foram extraídas para o módulo **Útil**. O módulo **Leitor** permite a utilização de **Útil**, como descrito na linha 17 do Código 3.2, mas a implementação não o utiliza, o que configura um alerta.

Como pode ser observado, é obtido um objeto contendo todas as ocorrências de divergência, ausência e alerta. O Código 3.3 representa o retorno da análise de conformidade.

Código 3.3 – Exemplo de retorno da análise de conformidade

```
{
  "ABSENCES": [
    {
      "targetFile": "./corrector-ts/src/cli/commands/json.ts",
      "originModule": "CLI",
      "targetModule": "Leitor"
    }
  ],
  "DIVERGENCES": [
    {
      "line": 2,
      "kind": "importation",
      "originFile": "./corrector-ts/src/cli/commands/json.ts",
      "targetFile": "./corrector-ts/src/models/prova.ts",
      "originModule": "CLI",
      "targetModule": "Modelos"
    },
    {
      "line": 3,
      "kind": "importation",
      "originFile": "./corrector-ts/src/corretor/corretor.ts",
      "targetFile": "@aws-sdk/client-s3"
      "originModule": "Corretor",
      "targetModule": "AWS-S3"
    }
  ],
  "ALERTS": [
    {
      "originModule": "Leitor",
      "targetModule": "Util"
    },
  ]
}
```

Fonte: Do autor (2023)

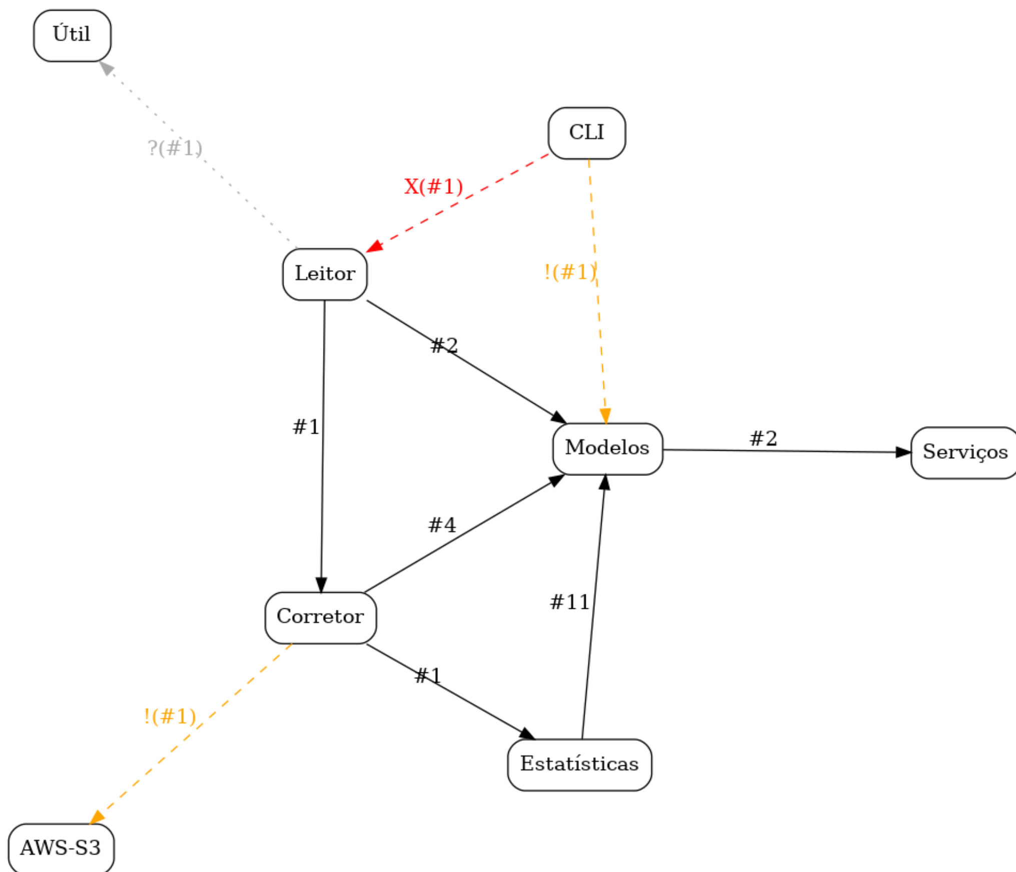
3.5 Resultados e visualização

Como resultados das análises, são obtidos um relatório textual, apresentado no Código 3.3, e representações visuais na forma de matriz e grafo. O relatório no formato JSON permite a integração da ferramenta com outros sistemas de software, além de permitir a fácil interpretação humana e apresentar detalhes como o arquivo e a linha do código que produziram o desvio. Entretanto, em sistemas de grande porte que apresentam muitos desvios, a interpretação do resultado textual pode ser lenta e um tanto confusa. Diante disso, resultados visuais também são produzidos. As representações visuais, além de exibir os pontos de erosão da arquitetura, também apresentam a comunicação de todos os módulos

da aplicação. A maior vantagem das representações visuais está na rápida identificação dos pontos de erosão através dos símbolos e cores das arestas e células da matriz.

No grafo, os nós representam os módulos e as arestas direcionadas representam as dependências (ESSAM; FISHER, 1970). O sentido da seta representa o sentido da dependência, o número seguido do simbolo "#" representa a quantidade de ocorrências daquela dependência e arestas tracejadas representam desvios arquiteturais. Como pode ser visto na Figura 3.4, o módulo **`Estatísticas`** depende do módulo **`Modelos`**. As cores laranja e vermelho e os símbolos "!" e "X" destacam, respectivamente divergências e ausências na imagem. Alertas, por sua vez, são representados pela cor cinza e o símbolo "?".

Figura 3.4 – Grafo resultante da análise de conformidade produzido pela ferramenta TSArch



Fonte: Do autor (2023)

Outra forma de visualização é a DSM (Matriz de Dependência Estrutural) utilizada para representar de forma escalável as dependências de um projeto (SULLIVAN et al., 2001). Nessa implementa-

ção, as colunas representam os módulos de onde partem as dependências enquanto as linhas representam os módulos com os quais as dependências são estabelecidas, assim, a leitura deve ser "módulo da coluna depende de módulo da linha". Por exemplo, pode ser visto na Figura 3.5, **Estatísticas** depende 11 vezes de **Modelos**. DSM segue a mesma lógica de cores e símbolos do grafo onde a célula vermelha iniciada por um “X” representa ausência, célula laranja iniciada por um “!” representa divergência e célula cinza iniciada por um “?” representa alerta.

Figura 3.5 – DSM resultante da análise de conformidade produzida pela ferramenta TSArch

Módulos	1	2	3	4	5	6	7	8
1 - Modelos			11	4	2	!1		
2 - Serviços	2							
3 - Estatísticas				1				
4 - Corretor					1			
5 - Leitor						X1		
6 - CLI								
7 - AWS-S3				!1				
8 - Útil					?1			

Fonte: Do autor (2023)

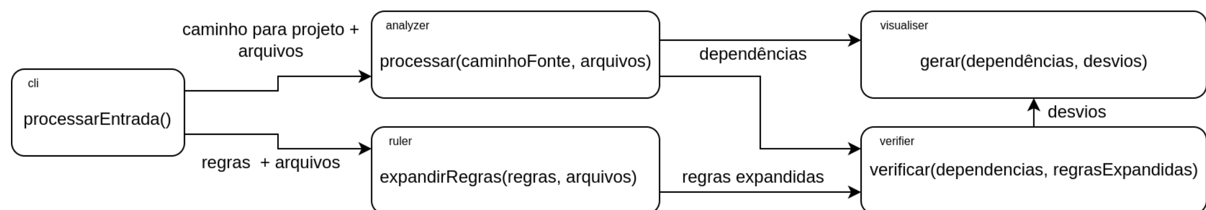
Como pode ser visto, em ambas representações é possível observar pelo código de cores a ausência de dependência de **CLI** para **Leitor**, as divergências de **CLI** para **Modelos** e de **Corretor** para **AWS-S3** e o alerta de **Leitor** para **Útil**.

4 PROJETO E IMPLEMENTAÇÃO

Esta seção apresenta a estrutura do projeto, tomadas de decisão e ferramentas utilizadas para o desenvolvimento da ferramenta TSArch.

Conforme pode ser observado na Figura 4.1, a ferramenta foi organizada nos seguintes cinco módulos em que cada um representa uma etapa de sua execução: Entrada de dados (CLI), Processamento de regras (*ruler*), Processamento do código-fonte (*analyzer*), Análise de Conformidade (*verifyer*) e Criação de Artefatos (*visualizer*).

Figura 4.1 – Representação do fluxo de comunicação entre os módulos da ferramenta



Fonte: Do autor (2023)

O módulo CLI coleta os caminhos para o código-fonte e as regras, assim como a lista de arquivos pertencentes ao projeto. Essas informações são fornecidas para os módulos seguintes. O *parser* identifica as dependências de cada arquivo que serão utilizadas na análise de conformidade e na produção das imagens. O módulo *ruler* processa as regras simplificando seu formato para ser utilizado na análise de conformidade a partir do conjunto de regras. A partir das regras simplificadas e do conjunto de dependências, o *verifier* realiza a análise de conformidade identificando os desvios arquiteturais e fornece ao *visualizer* os desvios arquiteturais. Por fim, *visualizer* utiliza as informações sobre dependências e desvios para produzir os artefatos visuais.

4.1 Entrada de dados

TSArch propõe a ser um executável de linha de comando (CLI) que realiza análises arquiteturais a partir das dependências entre os módulos definidos no conjunto de regras escrito no formato JSON. A execução da ferramenta consiste em invocar o binário da aplicação passando os parâmetros necessários, conforme exemplo ilustrado no Código 4.1. Os argumentos permitidos definem os caminhos para o projeto, artefatos, regras, DSM, grafo e relatório textual. Dentre esses argumentos, o único obrigatório é o caminho para o projeto que deve incluir o código-fonte e as configurações do compilador.

Código 4.1 – Exemplo de utilização da linha de comando da ferramenta TSArch

```
1 $ npx tsarch conformance ./meu_projeto --rules ./meu_projeto/regras.json --artifacts
    ./artefatos --dsm matriz.png --graph grafo.png --report relatorio.json
```

Fonte: Do autor (2023)

Outro arquivo importante para a execução da ferramenta é o arquivo "tsconfig.json", entretanto, infere-se que esse arquivo está presente na raiz do projeto informado.

A CLI foi implementada utilizando a biblioteca *Yargs* que oferece todos os recursos necessários para captura de informação, argumentos e ajuda.

4.2 Processamento das regras

O processamento das regras busca simplificar a análise arquitetural, transformando as regras descritas em outras equivalentes. Para isso, é necessário expandir os recursos sintáticos para obter a lista com todos os arquivos. O Código 4.2 exemplifica a expansão de um conjunto arbitrário de regras que se inspira na estrutura de diretórios da Figura 3.3.

Código 4.2 – Exemplo de inferência de tipos

```
1 {
2   "Foo": {
3     "files": ["./foo/**"] // -> ["./foo/a.ts", "./foo/b.ts", "./foo/bar/c.ts",
        "./foo/bar/d.ts"]
4   },
5   "Bar": {
6     "files": [./bar/bar.ts],
7     "forbidden": ["Foo"] // -> ["./foo/a.ts", "./foo/b.ts", "./foo/bar/c.ts",
        "./foo/bar/d.ts"]
8   }
9 }
```

Fonte: Do autor (2023)

Primeiramente, transformam-se as definições genéricas, compostas por um caminho terminado com asterisco, a fim de se obter uma lista de arquivos que pertencem a essa definição. A linha 3 do Código 4.2 representa, como comentário, o que é de fato retornado nesse passo. Em seguida, referências a módulos utilizadas nas regras são substituídas pelos respectivos arquivos e pacotes que pertencem ao módulo em questão. O comentário da linha 7 do Código 4.2 representa o resultado final da regra após a expansão.

Com os módulos definidos de forma extensa, sem utilizar referências a outros módulos ou generalizações, é possível realizar operações de conjuntos para transformar regras do tipo "*allowed*" em "*forbidden*" ou vice-versa. Já que, por definição, um conjunto é necessariamente disjunto ao outro. Neste trabalho, optou-se por utilizar apenas regras do tipo "*allowed*", assim, as regras do tipo "*forbidden*" são convertidas para "*allowed*". Os Códigos 4.3 e 4.4 trazem um exemplo dessa transformação. Considerando os arquivos descritos na primeira linha e os arquivos proibidos descritos na linha 4 do Código 4.3, o conjunto de arquivos permitidos deve ser todos os arquivos exceto os proibidos, como pode ser visto na linha 4 do Código 4.4. Isso é feito de forma automática pela ferramenta, portanto se arquivos forem adicionados ou removidos a ferramenta se ajustará.

Código 4.3 – Conjunto de regras antes da transformação

```
1 // arquivos do projeto:  a.ts, b.ts, c.ts, d.ts
2
3 "Foo": {
4   "forbidden": ["a.ts"]
5 }
```

Fonte: Do autor (2023)

Código 4.4 – Conjunto de regras depois da transformação

```
1 // arquivos do projeto:  a.ts, b.ts, c.ts, d.ts
2
3 "Foo": {
4   "allowed": ["b.ts", "c.ts", "d.ts"]
5 }
```

Fonte: Do autor (2023)

4.3 Processamento do código-fonte

O objetivo desse módulo é relacionar elementos da linguagem – e.g., variáveis, classes, interfaces, funções e parâmetros – aos seus tipos e, consequentemente, ao arquivo que implementa esses elementos, definindo uma dependência. O resultado dessa etapa é a lista de todos os elementos existentes no arquivo processado, associando a cada um (i) seu tipo, (ii) o arquivo que implementa o tipo e (iii) sua linha correspondente. As importações, por sua vez, estão relacionadas apenas à linha e ao arquivo importado.

Para buscar essas informações, é utilizada a API do compilador da linguagem TypeScript, chamada de TypeScript *Compiler* API[1]. Essa API oferece os recursos necessários para navegar na estrutura do código e extrair as informações que apontam dependências como árvore sintática e o *TypeChecker*.

O compilador utiliza atribuições para realizar a inferência de tipos. Isso significa que, se um valor numérico está sendo atribuído a uma variável sem anotação explícita, essa variável também terá o tipo numérico. Entre variáveis, a inferência pode ser feita por transitividade, onde o tipo de uma variável é o mesmo da variável atribuída, de forma semelhante ao que acontece com tipos primitivos. As linhas 1 e 2 do Código 4.5 reportam um exemplo de inferência por atribuição onde a variável "a" recebe o tipo *Map* após ser atribuída a um novo objeto *Map* e, por transitividade, a variável "b" recebe o mesmo tipo de "a".

Código 4.5 – Exemplo de inferência de tipos

```
1  let a = new Map<boolean, number[]>()     // type: Map<boolean, number[]>
2  let b = a                                // type: Map<boolean, number[]>
3
4  let arr = [a, b, "string"]               // type: (Map<boolean, number[]> | string)[]
5  let brr = [0, 3.2, {}]                   // type: (number | Object)[]
6
7  class A {}
8  class B extends A {}
9  class C extends A {}
10
11 let arr1 = [new B(), new C()]            // type (B | C)[]
12 let arr2 = [new B(), new C(), new A()]   // type: A[]
```

Fonte: Do autor (2023)

Outro caso ocorre durante a atribuição de arranjos, onde a especificação do arranjo é feita pelos tipos dos seus elementos, buscando o menor e melhor conjunto que defina o arranjo, o que pode ser visto nas linhas 4 e 5 do Código 4.5. No caso do arranjo da linha 4, em que as variáveis "a" e "b" possuem o mesmo tipo, o mecanismo de inferência o contabiliza apenas uma vez ao construir um arranjo heterogêneo (MICROSOFT, 2022b). Assim, o arranjo "arr" contém elementos do tipo das variáveis "a" e "b" e também *string* devido a literal declarada nele. O arranjo "brr" segue a mesma lógica de "arr", entretanto, é composto apenas por literais tendo os tipos *number* e *Object*. Ou seja, o processo de inferência de tipos para arranjos irá tentar reduzir a redundância não repetindo os tipos caso o arranjo possua mais de um elemento de um determinado tipo. Considerando heranças, os elementos do arranjo

[1] Embora não possua documentação oficial, exemplos de uso da API podem ser vistos no repositório do GitHub do compilador (https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API).

podem ou não ser agrupados pelo tipo herdado comum. Só serão agrupados pelo tipo comum caso um elemento do tipo comum esteja no arranjo como representado nas linhas 11 e 12 do Código 4.5.

Como o TypeScript não obriga o desenvolvedor a declarar o tipo das variáveis, um módulo pode depender de outro módulo a partir de um terceiro. Portanto, não é possível identificar as dependências apenas pelas importações. Por exemplo, no projeto implementado no Código 4.6 submetido as regras definidas no Código 4.7, o módulo **Recursos Humanos** não pode depender do módulo **Relatorio** (Código 4.7, linha 8), entretanto a classe "Trabalhador" (Código 4.6, linha 3), do módulo **Trabalhador** (Código 4.7, linha 2), implementa o método "reportar" que retorna um objeto que pertence ao módulo **Relatorio** (Código 4.6, linha 4), o que permite que **Recursos Humanos** tenha acesso a relatorio formando uma dependência implícita sem a menção do tipo **Relatorio** no arquivo "recursos_humanos.ts" (Código 4.6, linha 14).

De modo a resolver esse problema, utilizam-se os nomes qualificados dos tipos encontrados para relacionar o tipo à sua origem e, portanto, identificando apropriadamente a dependência. O nome qualificado é utilizado para remover a ambiguidade de objetos com nomes iguais em escopos diferentes. Em TypeScript um nome qualificado de um objeto contém o caminho para o arquivo onde ele foi declarado, o que torna possível identificar o módulo de origem do tipo utilizado.

Código 4.6 – Exemplo de dependência sem importação direta

```
1  /trabalhador.ts:
2  import { Relatorio } from "./relatorio"
3  export class Trabalhador {
4    public reportar(): B{ return new Relatorio() }
5  }
6
7  /relatorio.ts:
8  export class Relatorio {}
9
10 /recursos_humanos.ts
11 import { Trabalhador } from "./trabalhador"
12
13 let trabalhador = new Trabalhador() // trabalhador:Trabalhador
14 let relatorio = trabalhador.reportar()   // relatorio:Relatorio nome qualificado
       "./relatorio".Relatorio
```

Fonte: Do autor (2023)

Código 4.7 – Regras para o exemplo de dependência sem importação direta

```
1  {
2      "Trabalhador": {
3          "files": ["./trabalhador.ts"],
4          "allowed": ["Relatorio"]
5      }
6      "Recursos Humanos": {
7          "files": ["./recursos_humanos.ts"]
8          "allowed": ["Trabalhador"]
9      }
10     "Relatorio": {
11         "files": ["./relatorio.ts"]
12     }
13 }
```

Fonte: Do autor (2023)

Em suma, esse módulo é responsável por identificar dependências através de importações explícitas e do nome qualificado dos tipos dos objetos declarados, o que permite identificar as dependências implícitas sem ser necessário mapear as declarações de tipos e utilização de *aliases* e tratar ambiguidades. Como ponto negativo, essa implementação dificulta a utilização de *cache* visto que tal recurso necessita da compilação completa do projeto.

4.4 Análise de conformidade

Para realizar o passo da análise de conformidade, é necessário saber qual módulo cada arquivo pertence. Para isso, é mapeado para cada arquivo qual módulo ele pertence.

Em seguida, com as dependências encontradas na análise de código-fonte, é feita a busca por divergências. Nessa etapa, para cada dependência do módulo analisado, é avaliado se o arquivo que o módulo depende é permitido nas regras, ou seja, o arquivo está contido na lista de *allowed*.

Para encontrar as ausências e alertas, é feita uma lógica invertida onde se parte das regras e verifica se todas as dependências mandatórias de fato existem. Assim, itera-se sobre o conjunto de arquivos definidos nas regras, *required* ou *allowed* verificando se o arquivo ou módulo especificado foi utilizado. Se algum arquivo ou módulo presente das regras *required* não for utilizado configura-se uma ausência e caso algum arquivo ou módulo presente nas regras *allowed* não for utilizado configura-se um alerta.

Após todos esses passos, os quatro conjuntos de relações necessários para construir os artefatos foram construídos: (i) lista de dependências (convergência), (ii) lista de divergências, (iii) lista de ausências e (iv) lista de alertas.

4.5 Criação de artefatos

A partir das dependências encontradas na etapa de conformidade, são produzidos três tipos de artefatos: relatório textual, sendo composto por todas as dependências exceto convergências, e os artefatos visuais – grafo e DSM – que representam todas as dependências identificadas no processo.

A geração do relatório é feita através da escrita de um arquivo texto utilizando a função "writeFileSync" fornecida pela biblioteca que interage com sistemas de arquivos da linguagem.

Com as dependências, é possível aglutinar os dados contabilizando a quantidade de relações existente entre cada módulo formando uma lista onde cada par de módulos é associado ao total e tipo de dependência. Com isso, é possível utilizar a biblioteca da ferramenta GraphViz[2] que fornece funções que auxiliam a escrita do *script* DOT que é utilizado para gerar as imagens PNG do grafo e da DSM.

Para desenhar o grafo, é adicionado um nó para cada módulo e, em seguida, itera-se sobre a lista de dependências e, para cada elemento, é adicionado uma nova aresta no grafo onde o par ordenado de módulos representa a direção da aresta, a quantidade de dependências representa o peso da aresta e o tipo de dependência define o estilo que a aresta possuirá, como cor e continuidade, assim como o símbolo que acompanhará o peso. Para realizar a distribuição dos nós do grafo na imagem, foi utilizado o motor SFDP[3] que, após testes ad hoc, apresentou o resultado mais agradável por apresentar os nós de forma mais espaçada cruzando menos arestas dentre DOT[4], NEATO[5] e FDP[6].

A DSM também é desenhada utilizando GraphViz, entretanto, a representação da matriz é feita utilizando a estrutura de legenda que permite desenhar tabelas. A sintaxe é semelhante à sintaxe de tabela do HTML o que permite construir um *template* utilizando interpolação de *strings*. Apesar de

[2] GraphViz é uma ferramenta que gera imagens de grafos utilizando a linguagem DOT (https://graphviz.org/).

[3] SFDP (*Scalable Force-Directed Placement*) é um motor especializado para grandes grafos (https://graphviz.org/docs/layouts/sfdp/).

[4] DOT é um motor padrão de desenho de grafos que tende a linearizar o grafo de modo que ele cresça para uma única direção (https://graphviz.org/docs/layouts/dot/).

[5] NEATO é um motor para grafos pequenos e não direcionados que busca evitar cruzamento de arestas e normalizar a distância entre os nós (https://graphviz.org/docs/layouts/neato/).

[6] FDP (*Force-Directed Placement*) é um motor semelhante ao NEATO mas tende a procurar arestas mais curtas (https://graphviz.org/docs/layouts/fdp/).

não ser usual, utilizar GraphViz para gerar a visualização permitiu simplificar o processo de geração da imagem da DSM retirando a necessidade de um navegador para visualizar a matriz representada em HTML, por exemplo. Embora não seja elegante, muito menos apropriada, a solução é eficiente em termos de usabilidade e atende a proposta de produzir uma imagem PNG.

Como um outro exemplo, ao aplicar a ferramenta sobre o próprio código fonte utilizando as regras definidas no Código 4.8, foram obtidas os artefatos mostrados nas Figuras 4.2 e 4.3. Em resumo, foram detectadas onze divergências das quais não eram de conhecimento do autor desta monografia.

Código 4.8 – Regras arquiteturais da ferramenta

```
{
    "analyzer": {
        "files": ["./src/analyzer/*"],
        "allowed": ["util"]
    },
    "cli": {
        "files": ["./src/cli/**"],
        "allowed": ["tsarch"," util"]
    },
    "conformance": {
        "files": ["./src/conformancer/*"],
        "allowed": ["analyzer","ruler"],
        "required": ["ruler"]
    },
    "visualizer": {
        "files": ["./src/visualizer/*"],
        "allowed": ["util"]
    },
    "tsarch": {
        "files": ["./src/tsarch/*"],
        "forbidden": ["cli"]
    },
    "util": {
        "files": ["./src/utils/*"]
    },
    "ruler": {
        "files": ["./src/ruler/*"]
    }
}
```

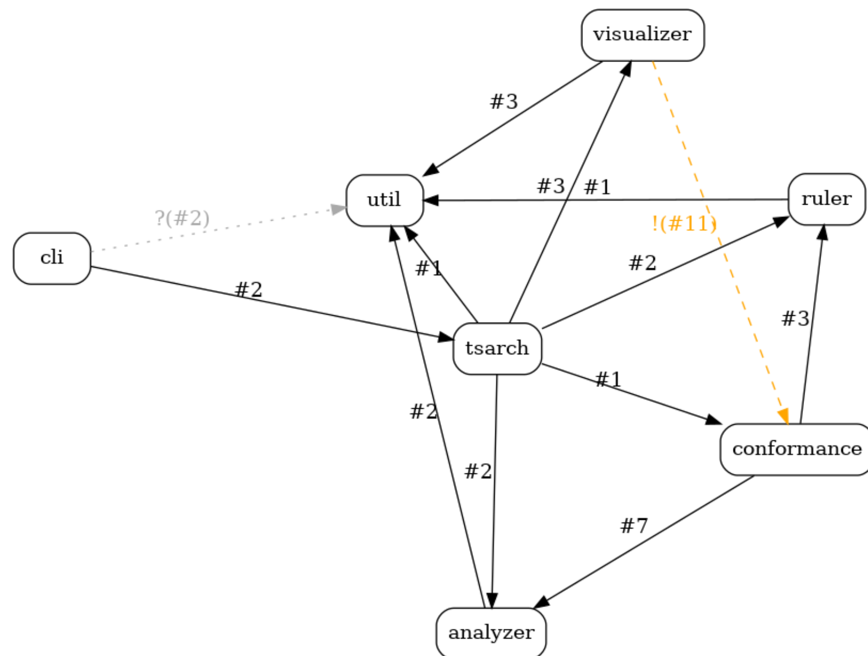
Fonte: Do autor (2023)

Figura 4.2 – DSM resultante da execução da ferramenta sobre o próprio código fonte

Módulos	1	2	3	4	5	6	7
1 - analyzer				7		2	
2 - util	2		1		3	1	?2
3 - ruler				3		2	
4 - conformance					!11	1	
5 - visualizer						3	
6 - tsarch							2
7 - cli							

Fonte: Do autor (2023)

Figura 4.3 – DSM resultante da execução da ferramenta sobre o próprio código fonte



Fonte: Do autor (2023)

5 FERRAMENTAS RELACIONADAS

TypeScript, como uma das linguagens de programação mais utilizadas segundo GitHub (2022), possui uma comunidade que desenvolve diversas ferramentas de apoio buscando resolver os mais diversos problemas existentes. Dentre essas ferramentas, existem analisadores estáticos responsáveis por qualidade de código e segurança, geradores de visualizações, etc. TSArch, como uma ferramenta que realiza análise estática de código com foco em qualidade por meio da identificação de desvios arquiteturais, se assemelha a outras disponíveis no mercado compondo o ecossistema da linguagem.

Dentre os analisadores estáticos, podem ser citados os servidores de linguagem, mais conhecidos como *Language Servers*, utilizados largamente nos editores de texto, e ferramentas relacionadas a estilo e qualidade de código (BINKLEY, 2007). Os servidores de linguagem são responsáveis por fornecer ao editor de texto recursos como sugestão de código, atalhos para documentação e implementação, além de identificar erros de sintaxe e de lógica em determinados casos (MICROSOFT, 2021; SOURCEGRAPH, 2022). Geralmente, os servidores de linguagem estão em execução durante o processo de codificação identificando, em tempo de escrita de código, erros relacionados a identificadores e, em tempo de salvamento, problemas mais complexos. Exemplos de *language servers* são TypeScript *Language Server*[1] para TypeScript e JavaScript, e Quick Lint[2] e Flow[3] para JavaScript.

Outras ferramentas de análise estática são relacionadas à qualidade e segurança, responsáveis por encontrar *bugs*, *code smells* e vulnerabilidades (BINKLEY, 2007). Nesse caso, essas ferramentas podem ser configuradas para executar após ou durante o salvamento do código fonte, durante um *commit* para o gerenciador de versões e também em fluxos de CI/CD (FOWLER; FOEMMEL, 2006). Ferramentas de qualidade voltadas ao estilo, por exemplo, podem fazer alterações no código fonte para adequar as regras de estilo como remover ou adicionar ponto e vírgula nos finais de linha, homogeneizar caracteres na indentação, etc. Prettier[4] é um exemplo de ferramenta do ecossistema que realiza verificações estéticas e que também é capaz de aplicar respectivas correções (PRETTIER, 2022). De forma complementar, ESLint[5] é uma ferramenta leve de análise que aponta possíveis *bugs*, erros de sintaxe e outras funcionalidades adicionadas via *plug-in* como análises de segurança. SonarQube[6], por sua vez, é uma ferramenta

[1] TypeScript Language Server (2023): https://www.npmjs.com/package/typescript-language-server
[2] Quick Lint (2023): https://github.com/quick-lint/quick-lint-js
[3] Flow (2023): https://github.com/flow/flow-for-vscode
[4] Prettier (2023): https://prettier.io/
[5] ESLint (2023): https://eslint.org/
[6] SonarQube (2023): https://www.sonarsource.com/products/sonarqube/

mais robusta e multilinguagem capaz de realizar uma análise mais completa e profunda identificando problemas que ESLint não é capaz de identificar.

Ferramentas de visualização permitem um planejamento e entendimento do software naquele momento. Grafos, diagramas UML e DSM são algumas formas de visualização possíveis de se extrair. Um exemplo de ferramenta de visualização é o Arkit[7] que, a partir do código fonte, produz grafos de dependência com base no arquivo de *schema* que declara os componentes arquiteturais. Para diagramas UML, a ferramenta Tplant[8] adapta o código TypeScript para o formato PlantUML utilizado para renderização dos diagramas. As IDEs da JetBrains, dentre os recursos e ferramentas para o desenvolvimento como *code completion* e geração de código, também oferecem recursos de visualização do código, exibindo estruturas hierárquicas do projeto trazendo informações sobre hierarquia de classes e interfaces (JETBRAINS, 2022).

Este trabalho é baseado no ArchRuby (MIRANDA; VALENTE; TERRA, 2015) e no ArchPython (LIMA; TERRA, 2020) que propuseram, respectivamente, ferramentas semelhantes para projetos Ruby e Python. Essas ferramentas realizam análise estática do código identificando dependências entre os componentes de código e relacionando com a arquitetura ideal especificada. Essas ferramentas também geram visualizações de grafo e DSM representando o relacionamento entre os módulos definidos e destacando os que apresentam desvios arquiteturais. Outra ferramenta que inspirou este trabalho foi a DCLsuite a qual se baseia na *Dependency Constraint Language* (DCL) que é uma linguagem especializada em definições de regras. DCL oferece recursos de especificação mais amplos do que a simples definição de módulos e arquivos proposta neste trabalho (TERRA; VALENTE, 2008). Baseando na DCL, o DCLcheck também realiza a análise de conformidade arquitetural de projetos Java que possuem sua arquitetura descrita utilizando a DCL. DCLCheck é um *plug-in* da IDE Eclipse, o que permite apontar os desvios arquiteturais no próprio editor (TERRA; VALENTE; MIRANDA, 2012). DCLfix, por sua vez, complementa as ferramentas anteriores orientando refatorações que corrijam os desvios arquiteturais encontrados (TERRA et al., 2012).

TSArch se difere do ArchRuby, ArchPython e DCLSuite na característica da linguagem tratada. Por um lado, Java é uma linguagem fortemente tipada com verificação de tipos em tempo de compilação. Por outro lado, Python e Ruby são linguagens fracamente tipadas cuja anotação de tipos é totalmente opcional e dispensável para execução do código em questão o que dificulta muito a análise de confor-

[7] Arkit (2020): https://github.com/dyatko/arkit
[8] Tplant (2023): https://github.com/bafolts/tplant

midade por não apresentar os tipos de forma explícita. TypeScript, por se tratar de um complemento da linguagem JavaScript que apresenta comportamento semelhante a Ruby e Python, também se trata de uma linguagem fracamente tipada sem anotações de tipos, entretanto, utilizando o modo estrito de compilação obriga o desenvolvedor a utilizar de forma explícita as anotações de tipos. Mas, mesmo utilizando o modo estrito, ainda são permitidas certas construções que demandam inferência como na inicialização de variáveis onde o tipo da variável recebe o mesmo tipo do valor atribuído.

6 CONSIDERAÇÕES FINAIS

Sob a demanda de melhorias e novas funcionalidades, sistemas de software estão sujeitos a se afastar de sua arquitetura original. Esse processo, conhecido como erosão arquitetural, pode dificultar correção de problemas e adição de novas funcionalidades. Portanto, estudos sobre a conformidade arquitetural vêm auxiliar a identificar e prevenir a erosão arquitetural.

A linguagem TypeScript, *superset* da linguagem JavaScript, é atualmente uma das linguagens mais utilizadas e oferece ao desenvolvedor grande liberdade de codificação. Mais importante, possui um ecossistema vasto de bibliotecas, *frameworks* e ferramentas, além de uma comunidade engajada no desenvolvimento e manutenção. No âmbito da análise de conformidade, a liberdade que a linguagem oferece é uma dificuldade para o desenvolvimento de ferramentas de análise que abranjam todo o escopo de soluções possíveis como ecossistemas Node e Deno e complementos de sintaxe como TSX implementado pela biblioteca React.

O projeto exemplo proposto neste trabalho buscou implementar as estruturas mais utilizadas dentro do paradigma orientado a objetos e funcional. Dentre as estruturas implementadas estão classes, interfaces, heranças, funções, funções anônimas, funções de primeira ordem, variáveis e objetos mas também recursos da linguagem como tipagem dinâmica e *Type Aliases*.

A ferramenta proposta implementa uma solução capaz de identificar desvios arquiteturais de um projeto a partir de um conjunto de regras e gerar artefatos – JSON, grafo e DSM – que representam o relacionamento entre os módulos e destacam os desvios encontrados. Os desvios arquiteturais são identificados a partir das dependências entre os módulos declarados nas regras e são classificados entre alertas, ausências e divergências, os quais são destacados de formas distintas nas visualizações, arestas estilizadas de formas, cores e símbolos diferentes no grafo, células com cores e símbolos distintos na DSM e listas distintas no relatório textual. Uma forma de aplicação dessa ferramenta está associada a fluxos de integração e entrega contínua (CI/CD), por exemplo, *quality gate* em uma etapa anterior ao *deploy* da aplicação de forma a impedir o *deploy* caso algum desvio seja encontrado.

Trabalhos futuros incluem: (i) a implementação de uma linguagem de definição de regras mais robusta que considere características da linguagem como herança, regras de nomeação, formas de declaração de tipos semelhante ao que DCL se propõe para a linguagem Java; (ii) aumentar o escopo de análise da ferramenta de modo a dar cobertura a mais paradigmas como programação genérica, cobrir complementos de sintaxe como TSX e suportar projetos do ecossistema Deno; e (iii) aprimorar as vi-

sualizações produzidas de modo a apresentar, além da relação entre os módulos, a semelhança entre os módulos. Nesse caso, a sugestão é utilizar técnicas de agrupamento para gerar imagens onde os módulos semelhantes ou que tenham maior interação fiquem representados mais próximos tanto no grafo quanto na DSM.

REFERÊNCIAS

AMAZON. **What is DevOps?** 2022. Disponível em: <https://aws.amazon.com/pt/devops/what-is-devops/>. Acesso em: 28 fev. 2023.

ARKIT. **Visualises JavaScript, TypeScript and Flow codebases as meaningful and committable architecture diagrams**. 2020. Disponível em: <https://github.com/dyatko/arkit>. Acesso em: 28 fev. 2023.

BINKLEY, D. Source code analysis: A road map. In: **2007 Future of Software Engineering (FOSE)**. [S.l.: s.n.], 2007. p. 104–119.

BLACK, N. Boris cherny on typescript. **IEEE Software**, v. 37, n. 2, p. 98–100, 2020.

BRYKCZYNSKI, B.; MEESON, R.; WHEELER, D. A. **Software inspection: Eliminating software defects**. Alexandria: Institute for Defense Analyses, 1994.

CLARK, L. **ES modules: A cartoon deep-dive**. 2018. Disponível em: <https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/>. Acesso em: 11 jun. 2023.

DANGOOR, K. **CommonJS: the First Year**. 2010. Disponível em: <https://www.blueskyonmars.com/2010/01/29/commonjs-the-first-year/>. Acesso em: 11 jun. 2023.

DENO. **Deno**. 2022. Disponível em: <https://deno.land/>. Acesso em: 28 fev. 2023.

DENO. **Interoperating with Node and NPM**. 2022. Disponível em: <https://deno.land/manual/node>. Acesso em: 28 fev. 2023.

ESLINT. **Find and fix problems in your JavaScript code**. 2023. Disponível em: <https://eslint.org/>. Acesso em: 28 fev. 2023.

ESSAM, J. W.; FISHER, M. E. Some basic definitions in graph theory. **Reviews of Modern Physics**, v. 42, n. 2, p. 271, 1970.

FACEBOOK. **Introduzindo JSX**. 2022. Disponível em: <https://pt-br.reactjs.org/docs/introducing-jsx.html>. Acesso em: 28 fev. 2023.

FACEBOOK. **Verificando Tipos Estáticos**. 2022. Disponível em: <https://pt-br.reactjs.org/docs/static-type-checking.html>. Acesso em: 28 fev. 2023.

FLOW. **Flow for VSCode**. 2023. Disponível em: <https://github.com/flow/flow-for-vscode>. Acesso em: 28 fev. 2023.

FOWLER, M.; FOEMMEL, M. **Continuous integration**. 2006. Disponível em: <https://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 5 maio 2023.

GITHUB. **The top programming languages**. 2022. Disponível em: <https://octoverse.github.com/2022/top-programming-languages>. Acesso em: 28 fev. 2023.

GUIDES, G. **Importando pacotes em Go**. 2020. Digital Ocean. Disponível em: <https://www.digitalocean.com/community/tutorials/importing-packages-in-go-pt>. Acesso em: 28 fev. 2023.

JACKSON, W. **JSON quick syntax reference**. [S.l.]: Apress, 2016.

JETBRAINS. **Source code hierarchy**. 2022. Disponível em: <https://www.jetbrains.com/help/webstorm/viewing-structure-and-hierarchy-of-the-source-code.html>. Acesso em: 28 fev. 2023.

KANG, S.; RYU, S. Formal specification of a javascript module system. In: **27th International Conference on Object oriented programming systems languages and applications (OOPSLA)**. [S.l.: s.n.], 2012. p. 621–638.

LI, R. et al. Understanding software architecture erosion: A systematic mapping study. **Journal of Software: Evolution and Process**, v. 34, p. e2423, 2022.

LIMA, E. F. de; TERRA, R. ArchPython: architecture conformance checking for Python systems. In: **34th Brazilian Symposium on Software Engineering (SBES)**. [S.l.: s.n.], 2020. p. 772–777.

MICROSOFT. **What is the Language Server Protocol?** 2021. Disponível em: <https://microsoft.github.io/language-server-protocol>. Acesso em: 28 fev. 2023.

MICROSOFT. **Modules**. 2022. Disponível em: <https://www.typescriptlang.org/docs/handbook/2/modules.html>. Acesso em: 28 fev. 2023.

MICROSOFT. **Type Inference**. 2022. Disponível em: <https://www.typescriptlang.org/docs/handbook/type-inference.html>. Acesso em: 28 fev. 2023.

MICROSOFT. **TypeScript Handbook**. 2022. Disponível em: <https://www.typescriptlang.org/docs/handbook/intro.html>. Acesso em: 28 fev. 2023.

MICROSOFT. **What is a tsconfig.json**. 2022. Disponível em: <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>. Acesso em: 28 fev. 2023.

MIRANDA, S.; VALENTE, M. T.; TERRA, R. ArchRuby: Conformidade e visualização arquitetural em linguagens dinâmicas. In: **VI Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session**. [S.l.: s.n.], 2015. p. 17–24.

MOZILLA CORPORATION. **JavaScript**. 2022. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Acesso em: 28 fev. 2023.

MOZILLA CORPORATION. **Módulos JavaScript**. 2022. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Modules>. Acesso em: 28 fev. 2023.

NODEJS. **Node.js v20.3.1 documentation Modules: CommonJS modules**. 2023. Disponível em: <https://nodejs.org/docs/latest/api/modules.html>. Acesso em: 11 jun. 2023.

NODEJS. **Node.js v20.3.1 documentation Modules: ECMAScript modules**. 2023. Disponível em: <https://nodejs.org/docs/latest/api/esm.html>. Acesso em: 11 jun. 2023.

OPENJS FOUNDATION. **Node.js**. 2022. Disponível em: <https://nodejs.org/en/about/>. Acesso em: 28 fev. 2023.

ORACLE. **Compilation Overview**. 2022. Disponível em: <https://openjdk.org/groups/compiler/doc/compilation-overview/index.html>. Acesso em: 28 fev. 2023.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **Software Engineering Notes**, v. 17, n. 4, p. 40–52, 1992.

PRETTIER. **Prettier: Code style options**. 2022. Disponível em: <https://prettier.io/docs/en/options.html>. Acesso em: 28 fev. 2023.

PRETTIER. **Prettier code formatting**. 2023. Disponível em: <https://prettier.io/>. Acesso em: 28 fev. 2023.

PYTHON. **Python Documentation Chapter 6. Modules**. 2023. Disponível em: <https://docs.python.org/3/tutorial/modules.html>. Acesso em: 11 jun. 2023.

QUICK LINT. **Quick Lint JS**. 2023. Disponível em: <https://github.com/quick-lint/quick-lint-js>. Acesso em: 28 fev. 2023.

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. [S.l.]: Bookman, 2005.

SHYLESH, S. A study of software development life cycle process models. **14th National Conference on Reinventing Opportunities in Management, IT, and Social Sciences (MANEGMA)**, p. 534–541, 2017.

SONARQUBE. **Clean code for teams and enterprises with SonarQube**. 2023. Disponível em: <https://www.sonarsource.com/products/sonarqube/>. Acesso em: 28 fev. 2023.

SOURCEGRAPH. **Langserver.org: A community-driven source of knowledge for Language Server Protocol implementations**. 2022. Disponível em: <https://microsoft.github.io/language-server-protocol/>. Acesso em: 28 fev. 2023.

STRNIŠA, R.; SEWELL, P.; PARKINSON, M. The java module system: core design and semantic definition. **ACM SIGPLAN Notices**, v. 42, n. 10, p. 499–514, 2007.

SULLIVAN, K. J. et al. The structure and value of modularity in software design. **Software Engineering Notes**, v. 26, n. 5, p. 99–108, 2001.

TERRA, R.; VALENTE, M. T. Towards a dependency constraint language to manage software architectures. In: **2nd European Conference on Software Architecture (ECSA)**. [S.l.: s.n.], 2008. p. 256–263.

TERRA, R. et al. DCLfix: A recommendation system for repairing architectural violations. In: **III Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Session**. [S.l.: s.n.], 2012. p. 1–6.

TERRA, R.; VALENTE, M. T.; MIRANDA, L. F. Conformação arquitetural com DCLcheck. **MundoJ**, X, n. 55, p. 44–49, 2012.

TPLANT. **Tplant**. 2023. Disponível em: <https://github.com/bafolts/tplant>. Acesso em: 28 fev. 2023.

TYPESCRIPT LANGUAGE SERVER. **TypeScript Language Server**. 2023. Disponível em: <https://github.com/typescript-language-server/typescript-language-server>. Acesso em: 28 fev. 2023.

YARGS. **Yargs**. 2022. Disponível em: <https://github.com/yargs/yargs>. Acesso em: 28 fev. 2023.

ZAPONI, C. **GitHut 2.0**. 2022. GitHub. Disponível em: <https://madnight.github.io/githut/#/pull_requests/2022/1>. Acesso em: 28 fev. 2023.