PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
ENGENHARIA DE COMPUTAÇÃO

CAIO GONÇALVES FEIERTAG

Programação Orientada a Aspectos em Typescript

Rio de Janeiro
Julho de 2022

CAIO GONÇALVES FEIERTAG

Programação Orientada a Aspectos em Typescript

Relatório de Projeto Final, apresentado ao programa de Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do titulo de Engenheiro de Computação.

Orientador: Marcos Vianna Villas

Rio de Janeiro
Julho de 2022

RESUMO

Feiertag, Caio Gonçalves; Villas, Marcos Vianna. **Programação Orientada a Aspectos em TypesScript**. Rio de Janeiro, 2022. 64p. Relatório de Projeto Final – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

A Programação Orientada a Aspectos ainda não é um paradigma largamente adotado por desenvolvedores, principalmente, pela curva de aprendizado que apresenta quando comparada a outros paradigmas de programação. Neste trabalho foi feito um estudo sobre as ferramentas de JavaScript existentes que possibilitam a implementação do paradigma, com base nisso, foi proposta uma nova ferramenta focada na linguagem TypeScript, visando, principalmente, diminuir esta curva através de interfaces claras e auto-descritivas para criação de aspectos, além de prover uma ampla variedade de funcionalidades comuns de programação orientada a aspectos e implementar suporte a reutilização de aspectos.

Palavras-chave: Programação Orientada a Aspectos; TypeScript; JavaScript; Ferramenta para Programação Orientada a Aspectos;

ABSTRACT

Feiertag, Caio Gonçalves; Villas, Marcos Vianna. **Aspect Oriented Programming in TypesScript**. Rio de Janeiro, 2022. 64p. Final Project Report – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Aspect Oriented Programming is not yet a paradigm widely adopted by developers, mainly because of the learning curve it presents when compared to other programming paradigms. In this work, a study was carried out on the existing JavaScript tools that allow the implementation of the paradigm, based on that, a new tool focused on the TypeScript language was proposed, aiming, mainly, to reduce this curve through clear and self-descriptive interfaces for the creation of Aspects, as well as providing a wide variety of common Aspect Oriented Programming functionality and enable aspect reuse support.

Keywords: Aspect Oriented Programming; TypeScript; JavaScript; Aspect Oriented Programming Tool;

LISTA DE ILUSTRAÇÕES

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

OA	Orientação a aspectos
OO	Orientação a objetos
POA	Programação orientada a aspectos
POO	Programação orientada a objetos

SUMÁRIO

1 **INTRODUÇÃO**

1.1 MOTIVAÇÃO

Ao longo do tempo, muitas técnicas de engenharia de software foram desenvolvidas para auxiliar os desenvolvedores a entregar aplicações com maior qualidade, porém com o avanço da tecnologia, há uma demanda crescente por produção de software que possuem o escopo cada vez mais amplo e na medida que ganham complexidade e as equipes envolvidas crescem, as escolhas de como o software é desenvolvido se tornam essenciais na produtividade e manutenção do sistema. Com isso, o paradigma da programação orientada a objetos, que é um dos mais utilizados na atualidade, está sendo questionado principalmente devido ao espalhamento e entrelaçamento do código.

Para isso novas alternativas estão sendo desenvolvidas, dentre elas o paradigma da programação orientada a aspecto, que visa principalmente separar as preocupações do sistema Hürsch e Lopes (1995).

De acordo com Ullman (2011) JavaScript já era uma das linguagens mais utilizadas na época da publicação, criada em 1995, se tornou bastante popular em páginas *web* principalmente as mais recentes. Apesar disso, na publicação de Bierman, Abadi e Torgensen (2014), a linguagem é introduzida como uma linguagem pobre para desenvolvimento e manutenibilidade de grandes aplicações. Dentro desse contexto, o TypeScript (MICROSOFT, 2012) surgiu para solucionar essas deficiências apresentadas, de forma a permitir uma transição suave entre as linguagens, já que o TypeScript é um super conjunto sintático de JavaScript que adiciona tipagem estática opcional, então todo programa JavaScript é um programa TypeScript.

1.2 OBJETIVO

Como mencionado por Kurdi (2013), a programação orientada a aspectos continua não sendo largamente adotada pela comunidade e precisa de mais estudos sobre, portanto o objetivo será renovar os estudos sobre o paradigma na linguagem mais utilizada atualmente que é o JavaScript, analisando as implementações atuais de POA para JavaScript, verificando se possuem suporte para TypeScript, a sua dificuldade de aprendizado, se a implementação se mistura ao código fonte e a maturidade da implementação. Além disso, será analisado brevemente o suporte a desenvolvimento de aplicações *web* e outros frameworks utilizados atualmente.

Por fim será implementada uma nova ferramenta open-source e com o mínimo de curva de aprendizado possível, para incentivar a adoção do paradigma por desenvolvedores que utilizam TypeScript, além de manter suporte a utilização para desenvolvimento *web* e que sejam compatíveis com outras tecnologias, como por exemplo, o React (META PLATFORMS, INC., 2013).

2 **PROGRAMAÇÃO ORIENTADA A ASPECTOS**

Neste capítulo será apresentada a motivação para a criação de um novo paradigma, sua origem e conceitos importantes para entendimento da programação orientada a aspectos.

2.1 PROBLEMAS DOS MODELOS DE PROGRAMAÇÃO PREDECESSORES

Há um universo de linguagens de programação e cada uma delas se beneficia de um ou mais modelos de programação, cada um deles propõe uma forma diferente no qual o programador construirá o algoritmo a ser desenvolvido. Dadas essas linguagens desenvolvidas, um ponto em comum é a programação imperativa definida por John von Neumann, este princípio defende a escrita de código de forma formalizada e sequencial e possibilita a criação de códigos de alto nível.

O principal modelo de programação adotado ao longo da história foi o modelo estruturado, ele é muita das vezes o primeiro contato que os programadores têm para aprenderem a desenvolver algoritmos, isso porque possuem uma transição pequena da descrição para o código desenvolvido. Porém com o aumento do poder computacional e a demanda crescente pelos mais diversos tipos de software e cada vez mais complexos, apesar da escrita de códigos de maneira estruturada possuir uma curva de aprendizado menor, não havia uma forma bem definida de componentização do algoritmo. Dessa forma, surgiu o modelo de POO que introduziu um novo conceito em que o fluxo do programa passou a ser feito pela interação entre objetos, por meio da mudança de visão de procedimentos e módulos para objetos e classes. “Ela deu uma importante contribuição para facilitar, por exemplo, a manutenabilidade, a componentização e a reusabilidade” (RESENDE; SILVA, 2005).

Apesar das grandes conquistas alcançadas pelos modelos de programação existentes, principalmente pela Programação Orientada a Objetos, nem todos os problemas no desenvolvimento de um software foram resolvidos, dos quais os mais marcantes são: Entrelaçamento de código e espalhamento de código.

> Utilizando apenas a OO como ferramental, ao programar-se interesses entrecortantes, pode surgir dois problemas de acoplamento e coesão no código: o espalhamento e o entrelaçamento do código. Um interesse é dito espalhado quando afeta vários componentes do sistema e entrelaçado quando se mistura com outros interesses dentro de um módulo (ALMEIDA, 2007, p. 6 apud FIGUEIREDO, 2006)

2.1.1 **Entrelaçamento do Código (*Tangled Code*)**

Como mencionado anteriormente, no desenvolvimento de software orientado a objetos o fluxo do programa passou a ser representado através de objetos e classes. Para isso, as classes de um sistema necessitam de atributos e métodos para compor as classes que irão representar uma entidade do mundo real. Porém esse processo nem sempre é

simples e ao não atribuir as responsabilidades de maneira correta, pode-se ao realizar interações entre os objetos criados, enfrentar problemas com código entrelaçado entre as classes, principalmente pela dificuldade de atribuir corretamente a responsabilidade em um primeiro momento.

> Quando o trabalho de preenchimento dos métodos se inicia, faz-se necessário inserir chamadas de responsabilidade de uma classe em outra... Estas linhas de código inseridas em outro componente para integração são denominadas código intrusivo ou invasivo. (RESENDE; SILVA, 2005).

Para demonstrar o problema, tomaremos de exemplo uma aplicação bastante comum que é uma lista de tarefas (*TODO List*), onde será possível registrar e remover tarefas que serão salvas em um banco de dados.

O Código 1 contém uma implementação da aplicação exemplo utilizando *Java*, primeiro foi criado uma classe Tarefa, que representa a entidade principal da aplicação definida, onde ela possui um método responsável por "registrar" e outro por "remover". Porém, nesses dois requisitos primários da aplicação já é possível visualizar o primeiro código intrusivo, para salvar um dado no banco de dados, faz-se necessário conectar ao banco e dar início a uma transação, logo após deve-se realizar a operação desejada para registrar a tarefa, por fim, as alterações devem ser cometidas e, com isso, a conexão encerrada. A implementação das responsabilidades que acontecem ao redor das operações de registro não devem ser de responsabilidade das Tarefas, logo, é necessário criar uma classe Banco com pelo menos dois métodos "iniciar" e "fechar" que serão chamados pela classe Tarefa.

Código 1 — Exemplo de entrelaçamento de código

```
class Banco {
    static void iniciar() {
        // Conecta e inicia transação com banco de dados
        return;
    }

    static void fechar() {
        // Comete transação e encerra conexão com banco de dados
    }
}

class Tarefa {
    static void registrar(String nome, String descricao) {
        Banco.iniciar();

        // Início código para registrar tarefa
        // Fim do código para registrar tarefa

        Banco.fechar();
    }

    static void remover(int idTarefa) {
        Banco.iniciar();

        // Início código para remover tarefa
        // Fim do código para remover tarefa

        Banco.fechar();
    }
}
```

Fonte: O autor (2022)

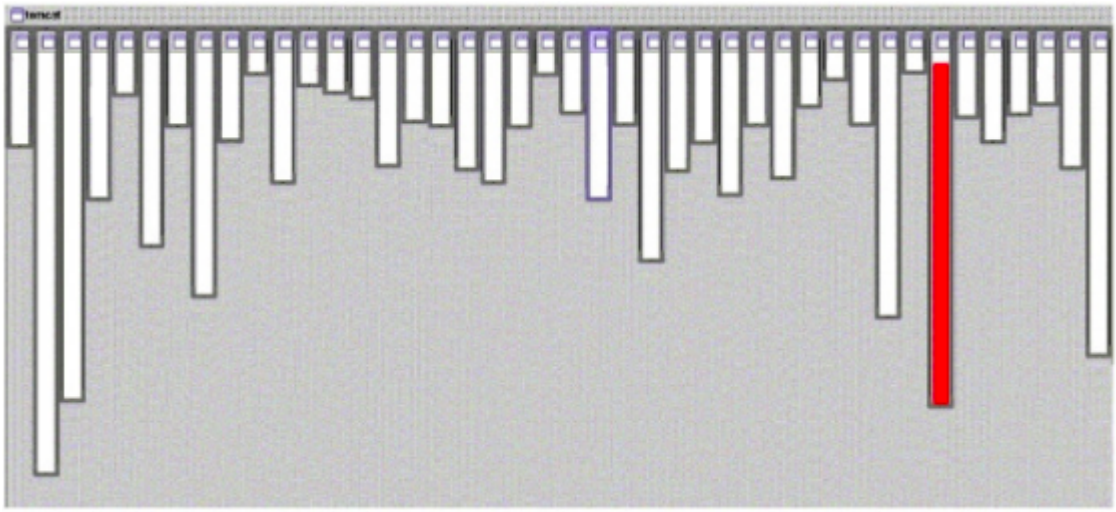
No Código 1, o interesse primário da classe Tarefa está delimitado pelos comentários de início e fim em ambos os métodos, pode-se perceber a inserção de código intrusivo nessa classe, que são as chamadas a métodos da classe Banco, responsáveis por iniciar e encerrar as comunicações com o banco de dados, estes códigos intrusivos podem potencialmente dificultar a leitura e adiciona responsabilidades de outras classes nos métodos. Além disso, só foram adicionados dois interesses entrecortantes, a tendência é o software ganhar cada vez mais complexidade e portanto um maior número deles estarem se misturando com as regras de negócio, por exemplo, poderia ser adicionado uma verificação de autorização, um *log* para registrar as ações realizadas, tratamento de erro, verificação de dados, entre outros. No exemplo apenas um componente do sistema foi criado, no próximo tópico será tratado como a aplicação como um todo será impactada por interesses transversais.

2.1.2 **Espalhamento do Código (Scattering Code)**

Além do problema citado no tópico anterior de entrelaçamento de códigos, visando agora um aspecto mais amplo, os requisitos entrecortantes com responsabilidades mal definidas acabam se repetindo ao longo de toda a aplicação, passando por diversos módulos da aplicação.

Primeiramente, pode-se observar em vermelho na Figura 1 o módulo de validação de *XML* do Apache Tomcat[1], cada barra vertical mostrada na figura é um módulo dessa aplicação, e cada linha da barra representa uma porção de código escrito do módulo. Apesar do código ser extenso, a responsabilidade foi bem modularizada e o código não se encontra espalhado ao longo de outros módulos aplicação.

Figura 1 — Implementação do processo de validação (parsing) de XML no Apache Tomcat

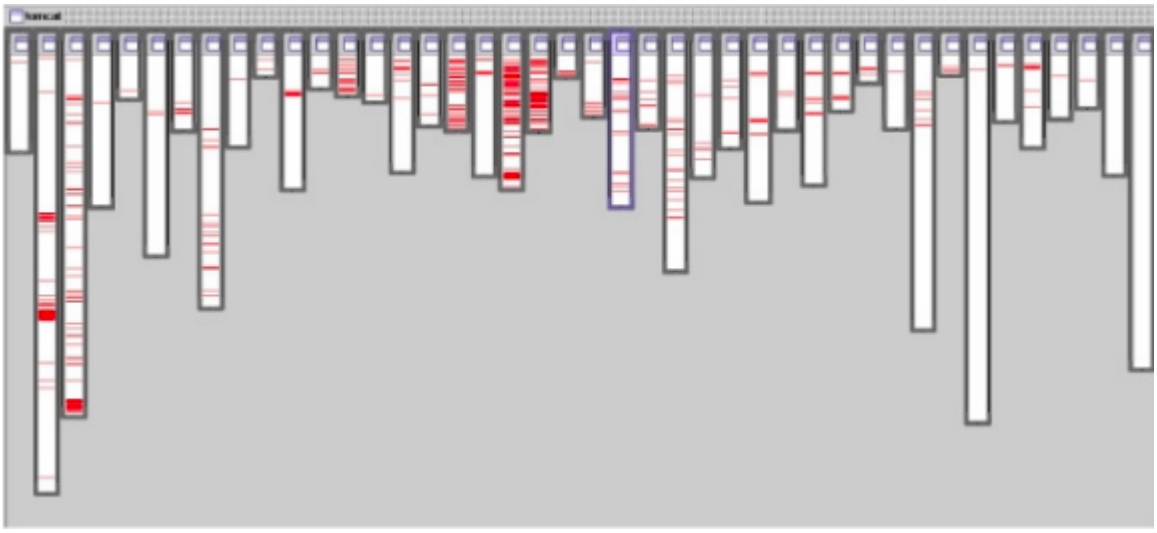


Fonte: SILVA, Kelli (2005, p. 25)

Porém, na Figura 2 é possível observar outro requisito transversal que não foi bem definido e modularizado, o registro de *log do Apache Tomcat* aparece em quase todos os módulos da aplicação, ao contrário do que foi retratado na Figura 1, portanto as linhas em vermelho aparecem em diversas barras, demonstrando o espalhamento dos código do interesse entrecortante ao longo de toda a aplicação. Por não centralizar e delimitar a implementação desse interesse, isto pode causar sérios problemas de manutenção, reusabilidade e comprometer o entendimento da aplicação.

1 Apache Tomcat é uma implementação de código aberto de tecnologias servlets para Java

Figura 2 — Implementação do registro de log do Apache Tomcat.



Fonte: SILVA, Kelli (2005, p. 25)

2.2 ORIGEM

A frente dos problemas relatados no tópico anterior, as pesquisas sobre o tema se iniciaram na década de 1990, a princípio surgiram pesquisas que propuseram novos paradigmas chamados de *Subject Oriented Programming* (Programação orientada a assuntos) e *Adaptive Programming* (Programação adaptável).

Segundo Silva (2004) as primeiras pesquisas mais focadas no assunto começaram a surgir em 1995, Cristina Lopes abordou o tema de separação limpa de interesses por meio de um relatório que identificam algumas técnicas que já estavam sendo desenvolvidas. Porém o termo programação orientada a aspectos só surgiu em 1997 quando Cristina se juntou com Gregor Kiczales no Xerox Palo Alto Research Center, onde definiram a nova terminologia e demonstraram os primeiros exemplos de código, mas foi apenas em 2001 que foi publicada a primeira linguagem de programação de aspectos que estendia *Java*, a *AspectJ* (KICZALES et al., 2001), desenvolvida também por Kiczales e sua equipe e se mantém até hoje como a ferramenta mais madura e utilizada de POA.

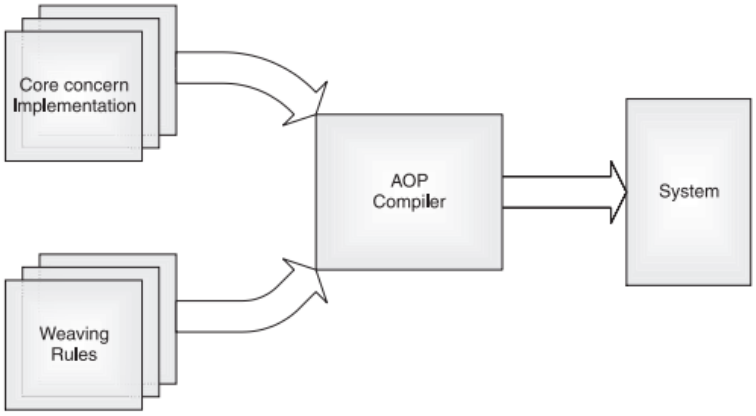
2.3 CONCEITO

A programação orientada a aspectos como mencionado anteriormente surgiu para complementar a POO, principalmente resolvendo o problema da separação de interesses, definindo de maneira mais assertiva os interesses transversais da aplicação. Para isso POA, introduziu o conceito de aspecto que possibilita a abstração e composição desses interesses.

A POA estende outros paradigmas, como a POO ou programação

> estruturada, propondo não apenas uma decomposição funcional, mas também sistêmica do problema. Isso permite que a implementação de um sistema seja separada em requisitos funcionais e não funcionais, disponibilizando a abstração de aspectos para a decomposição de interesses sistêmicos, além dos recursos já oferecidos pelas linguagens de componentes como Java, por exemplo. (GOETTEN; WINCK, 2006).

Figura 3 — Funcionamento base do processo de tecelagem de um software POA.



Fonte: Laddad (2003, p. 25)

Na Figura 3 pode-se observar a ideia principal do funcionamento de um sistema que foi implementado com POA, os *Core concern Implementation* são as implementações das preocupações primárias de seu sistema, já os *Weaving Rules* são as regras que definem como preocupações que possuem interesses entrecortantes devem interagir, o *AOP Compiler* será responsável por combinar as implementações centrais às preocupações entrecortantes do sistema, seguindo as regras de costuras que foram definidas, ao final da compilação o código do sistema e gerado. Para entender um pouco melhor esse processo, será apresentado nos próximos tópicos conceitos importantes que compõem o desenvolvimento de softwares orientados a aspectos, baseando-se principalmente em definições adotadas pelo *AspectJ*, por ser a ferramenta mais madura e não existir uma ferramenta referência para *JavaScript*.

2.3.1 **Componente**

Na programação Orientada a Aspectos, são chamados de componentes os interesses primários ou principais do sistema. Os componentes contém as funcionalidades básicas da aplicação e os problemas de negócio. A linguagem utilizada para a implementação dos mesmos não precisa prover recursos para utilizar o paradigma de Orientação a Aspectos. Portanto, todas as linguagens podem ser utilizadas para criar os interesses primários da aplicação, como por exemplo: *JAVA*, *JavaScript*, *C*, *PHP*, *C++*.

2.3.2 **Aspecto (*Aspect*)**

Os aspectos são os interesses do sistema, portanto são os interesses entrecortantes que serão utilizados ao longo de todos os módulos da aplicação, alvos dos problemas relatados em tópicos anteriores. A sua linguagem deve prover o ferramental necessário para a criação dos aspectos, permitindo a criação de estruturas que descrevem o comportamento dele e seja possível definir quando o comportamento ocorre. A abordagem mais comum é o código dos aspectos ser uma extensão do código dos *components*, pois como explicado por Silva (2019) garante uma integração direta entre os códigos e mantém acesso às funcionalidades da linguagem nos códigos dos aspectos, além disso, apresenta uma curva de aprendizado menor por não ser necessário aprender uma nova sintaxe. Exemplos dessa primeira abordagem seriam: *AspectJ(JAVA), AspectC++(C++), PHPAspect(PHP).*

A outra abordagem para criação de aspectos é feita de forma agnóstica[2] ao código dos componentes, portanto utiliza uma linguagem diferente para criação dos aspectos e exige uma ferramenta mais complexa para integrar ambos os códigos. Essa abordagem visa solucionar o problema de invasividade do código dos aspectos, este problema será um dos critérios da comparação das ferramentas que será visto no capítulo 3.

No Código 2 é apresentado um aspecto implementado com *AspectJ* para a classe Banco que foi utilizada na aplicação, exemplo de lista de tarefas mostrada no Código 1.

Código 2 — Exemplo de aspecto para Banco de dados

```
aspect AspectoBanco {
    pointcut transacoesBanco():
        call(static void Tarefa.registrar(String, String))
        || call(static void Tarefa.remover(int));

    before(): transacoesBanco() {
        // Conecta e inicia transação com banco de dados
    }
    after(): transacoesBanco() {
        // Comete transação e encerra conexão com banco de dados
    }
}
```

1 Ponto de corte

2 Adendos

Fonte: O autor (2022)

2.3.3 **Pontos de junção (*Join Points*)**

Um ponto de junção é um local identificável durante a execução do programa, onde um ou mais aspectos poderão atuar. O *AspectJ* dispõe de vários tipos de pontos de junção, sendo eles chamadas de métodos, chamadas a construtores, captura e atribuição de

2 Uma linguagem, implementação ou solução agnóstica na área de programação consiste em uma implementação que independe de uma específica, permitindo assim uma flexibilidade maior. Por exemplo, é muito comum a implementação de bibliotecas para Gateway de pagamento que possuem uma única Api de uso, mas que implemente suporte a vários gateways, dessa forma, o cliente poderá facilmente mudar entre serviços sem que sejam feitas grandes refatorações de código.

campos, tratamento de exceções e até mesmo execuções de adendos, que será introduzido no tópico 2.3.5.

> Um ponto de junção é um local bem definido no código primário em que a preocupação irá entrecortar a aplicação. Pontos de junção podem ser chamadas de métodos, invocações de construtores, tratadores de exceção, ou outros pontos na execução de um programa.[3] (GRADECKI; LESIECKI, 2003, p. 15).

Tomando como exemplo o Código 1, que contém o componente principal da aplicação, a classe Tarefa possui dois pontos de junções principais definidos por suas assinaturas, sendo eles:

1. static void Tarefa.registrar(String, String)
2. static void Tarefa.remover(int)

2.3.4 **Ponto de corte (*Pointcut*)**

Os pontos de corte, também chamados de pontos de atuação, são capazes de definir regras genéricas capazes de selecionar pontos de junção da aplicação. Dado um conjunto com todos os pontos de junção candidatos da aplicação, o ponto de corte será responsável por descrever regras para obter um subconjunto dos pontos.

> Pointcuts está para atributos assim como aspects está para class. Um pointcut pode ser entendido como uma variável. Esta variável armazena uma lista contendo as assinaturas de métodos que se tem interesse em interceptar. Esta interceptação é feita com o objetivo de alterar o fluxo original do programa e inserir novas chamadas, advices. (RESENDE; SILVA, 2005).

A sintaxe simplificada para definir um ponto de corte envolve definir o tipo do ponto de junção e uma assinatura. No Código 2 observamos a definição do ponto de corte para o aspecto de banco de dados delimitado pelo bloco 1, nele é possível ver a criação do ponto de corte chamado "transacoesBanco", que utiliza as duas assinaturas definidas no tópico anterior aplicadas ao tipo de junção *call,* nomeclatura do *AspectJ* para chamadas de métodos. Para definir os dois pontos foi necessário o uso do operador || (ou) para agrupar os pontos de junção que satisfazem a regra definida no aspecto. Para facilitar o entendimento, a regra definida utilizando a nossa linguagem seria: "os pontos de corte de transações de banco são compostos pela chamada estática do método Tarefa.registrar com dois argumentos String ou pela chamada estática a Tarefa.remover com um argumento inteiro".

O ponto de corte também pode ser definido sem agrupar as duas regras com o operador ou, isso pode ser feito utilizando o *Wildcard* que os *AspectJ* oferece, isto é, podemos utilizar o * para denotar quaisquer caracteres fora do período que queremos

---

3 A join point is a well-defined location within the primary code where a concernwill crosscut the application. Join points can be method calls, constructor invocations, exception handlers, or other points in the execution of a program.

definir, portanto a reescrita do ponto de corte ficaria: "*call* (*static void* Tarefa.*(*))". Novamente, em nossa linguagem ficaria: "os pontos de corte de transações de banco de dados são compostos pela chamada estática a quaisquer métodos que iniciem com 'Tarefa.' com quaisquer argumentos."

2.3.5 **Adendo (*Advice*)**

Os adendos são os responsáveis por conter a implementação do código de um aspecto, o mecanismo de implementação é similar a de criação de métodos para uma classe, estes levam o nome das posições pré-definidas nas quais devem ser chamados quando um ponto de junção é alcançado. Os momentos de execução são:

- Antes (*Before*)
- Ao redor ou durante (*Around*)
- Após (*After*)
- Após disparo de erro (*After Throwing*)

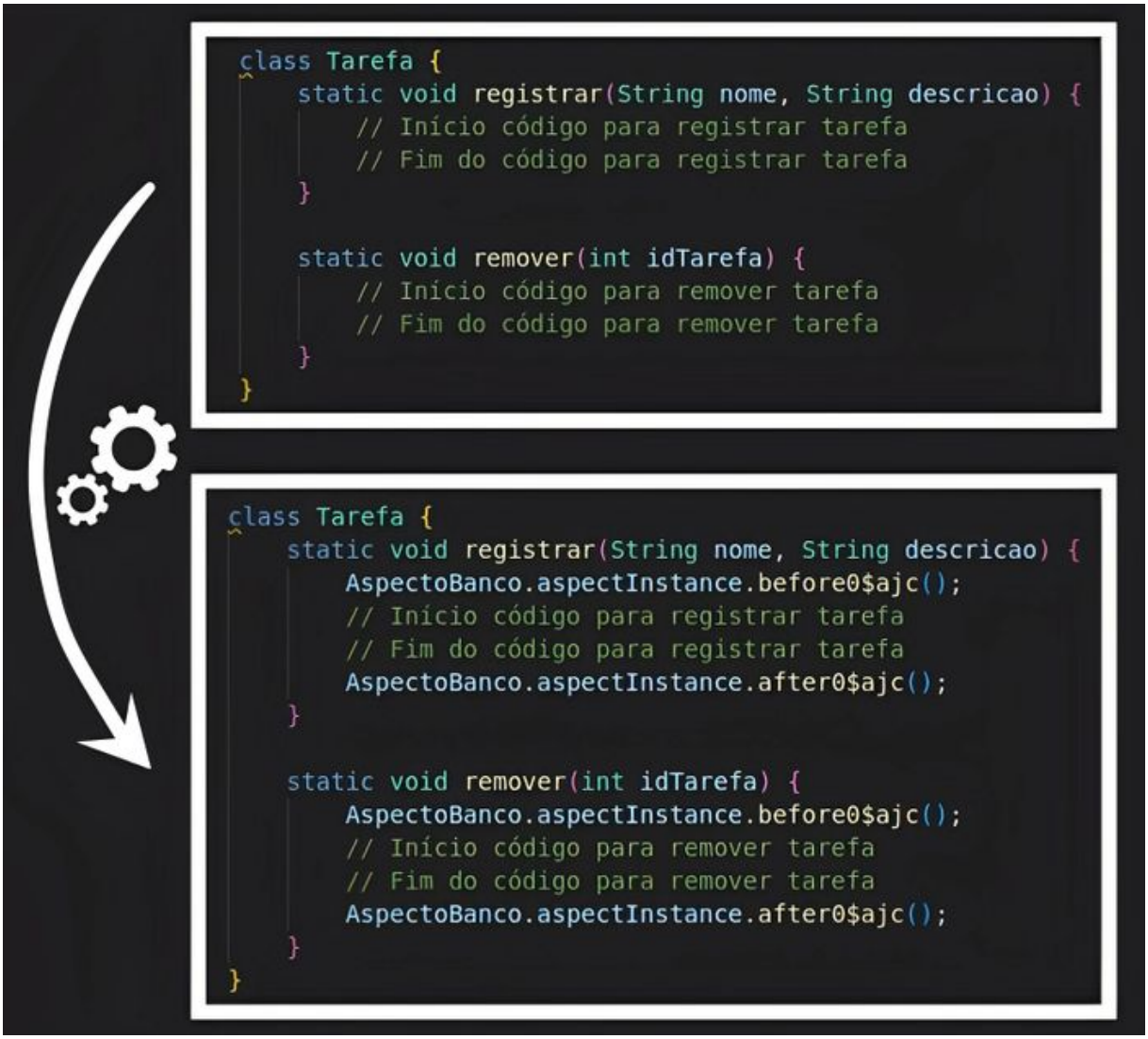
Para permitir iniciar a comunicação com o banco de dados na aplicação exemplo, foi criado no Código 2 um adendo *before* para substituir o que era feito na inicialização da comunicação com o banco e um adendo *after* para substituir o que era feito na finalização da comunicação, em ambos os adendos são aplicadas as regras definidas pelo ponto de corte "transacoesBanco".

2.3.6 **Tecelão de aspectos (*Aspect Weaver*)**

O Tecelão de aspectos é o responsável por costurar os códigos escritos na linguagem de componente, com os escritos na linguagem de aspectos seguindo as regras definidas nos mesmos. Ele é representado na Figura 3 pelo *AOP Compiler*, isto é, neste caso os códigos foram costurados em tempo de compilação utilizado um compilador oferecido pela ferramenta que implementa POA. É possível também fazer a costura dos códigos em tempo de execução ou carregamento, o tecelão de aspectos neste caso tende a ser menos complexo, e, por isso, foi o mais adotado nas implementações de *JavaScript*. Como referência, ambas as estratégias são implementadas pelo *AspectJ*.

Para exemplificar o que acontece neste processo, podemos utilizar como base o exemplo do Laddad (2003, p. 41,42) que mostra a representação do aspecto aplicado ao código do componente, mas com a ressalva desse processo acontecer apenas na versão *bytecode* (HILSDALE; HUGUNIN, 2004). Dessa forma na Figura 4 podemos ver o código do componente antes e depois da aplicação do aspecto, após a transformação foi inserido a chamada ao adendo *before* do AspectoBanco antes do código original dos métodos da classe Tarefa e o adendo *after* do AspectoBanco após o código dos métodos da classe.

Figura 4 — Representação do código após o processo de costura feito pelo AspectJ



Fonte: O autor (2022)

3 **ANÁLISE DAS IMPLEMENTAÇÕES EXISTENTES**

Neste capítulo será feita uma análise das ferramentas existentes para Programação orientada a aspectos em *JavaScript*, e assim efetuada uma comparação entre elas a fim de compreender os benefícios de cada uma e como isso pode agregar para a ferramenta que será proposta no próximo capítulo.

3.1 CRITÉRIOS

Os critérios que serão utilizados para análise seguirão os estudos de Huang, He e Li (2015) e Silva (2019) que comparam implementações de *JavaScript* com base na invasividade, brevidade e maturidade. Além disso, será avaliado se a implementação possui suporte para *TypeScript* e se é possível aplicá-la a tecnologias atuais de desenvolvimento *web*.

3.1.1 **Invasividade (*Invasiveness*)**

A invasividade avalia o quanto o código referente aos aspectos está presente no código original do programa e não pode ser separado do mesmo, segundo Silva (2019) existem diversas ferramentas que implementam com sucesso POA misturando os códigos, mas POA foi arquitetado para ser um suplemento e extensão da programação convencional. Por isso, o código fonte não deve ser modificado para fazer o uso do paradigma, principalmente no quesito de atualizações e manutenções do sistema. Como mencionado anteriormente, esse critério é alcançado quando o código dos aspectos é agnóstico ao código fonte.

3.1.2 **Brevidade (*Briefness*)**

Segundo Huang, He e Li (2015), o paradigma de POA é mais difícil de compreender do que POO para novos desenvolvedores que estão estudando novos paradigmas. Dessa forma é possível utilizar o *JavaScript* para definir sintaxes simples e claras para declarar os pontos de corte e adendos, facilitando seu uso e aprendizado.

3.1.3 **Maturidade (*Maturity*)**

A maturidade avalia a quantidade de funcionalidades que a ferramenta implementa de POA, assim como, inclui as funcionalidades básicas da linguagem JavaScript e sua natureza dinâmica.

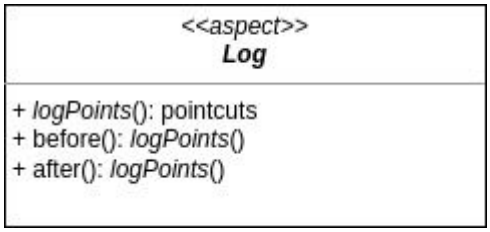
3.2 TESTE BASE

Para verificar e avaliar cada ferramenta que implementa POA em JavaScript, 2 aspectos serão desenvolvidos, e, por sua vez, aplicados em uma aplicação com escopo simples utilizando TypeScript e executadas com *node.js*[4]. Além disso, os mesmos aspectos foram aplicados em uma aplicações *web* utilizando o *framework React*.

3.2.1 **Aspecto para Log**

O aspecto de Log deverá mostrar as funcionalidades mais comuns de POA, portanto a Figura 5 mostra como o aspecto deve ser declarado, isto é, deve criar regras para os pontos de corte nomeado *logPoints*, estas regras serão aplicadas nos adendos *before e after*, para o teste será definido como pontos de corte quaisquer chamadas de métodos de classes. O adendo que é executado antes, deverá imprimir no *console* o nome da classe, o nome do método e seus argumentos. Já o adendo depois, deverá imprimir o método, a classe e o retorno.

Figura 5 — Definição de Aspecto Log



Fonte: O autor (2022)

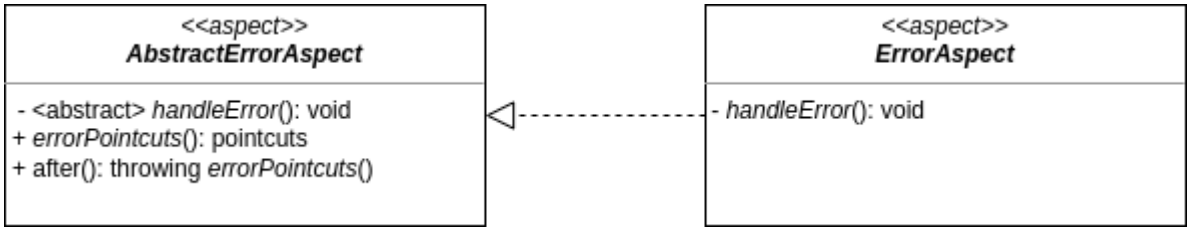
3.2.2 **Aspecto de Erro com abstração**

A publicação de Soares, Laureano e Borba (2002) apresenta aspectos reutilizáveis e para implementar tratamento de erro é utilizado um aspecto abstrato. Portanto o aspecto de erro desenvolvido para o teste base, terá como principal objetivo verificar se a ferramenta suporta aspectos reutilizáveis, assim como, testar se o adendo, que é executado após disparo de erro, é implementado. Para isso, será criado o *AbstractErrorAspect*, que é um aspecto abstrato que já define os pontos de corte e o adendo *afterThrowing* aplicado nos pontos de corte definidos pelo aspecto abstrato, que e faz uma chamada a um método *handleError* abstrato. Além disso, será criado o aspecto *ErrorAspect* que estende o aspecto de erro abstrato e implementa o método *handleError* com uma impressão do erro no *console*. A Figura 6 mostra o diagrama do aspecto de Erro descrito.

---

4 node.js - ferramenta capaz de executar JavaScript fora do navegador, disponível em: https://nodejs.org/en/

Figura 6 — Definição do aspecto de Erro

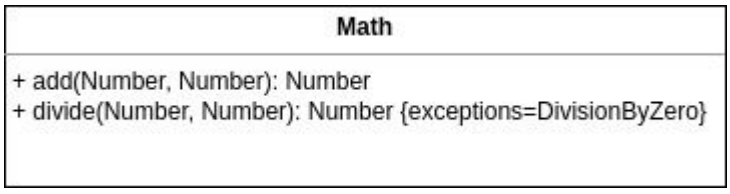


Fonte: O autor (2022)

3.2.3 **Aplicando os aspectos em um escopo simples**

Para testar os aspectos mencionados, foi criado uma classe *Math* que possui os métodos *add* para adição e *divide* para divisão, como pode ser visualizado no diagrama da Figura 7. A aplicação consiste em um método principal que será executado pelo terminal utilizando o *node.js*, a execução consiste na realização de 3 operações utilizando a classe *Math* desenvolvida, primeiro uma adição, em seguida uma divisão válida, e por último, uma divisão por zero. O aspecto de *Log* como definido deve atuar nas 3 operações, ou também, como introduzido antes, nos 3 pontos de junção, imprimindo no *console* antes e depois das chamadas, enquanto isso o de *Error* irá também atuar nas 3, porém apenas na última operação que irá disparar um erro, e neste instante o código do adendo deve ser executado, fazendo com que seja possível visualizar a impressão no *console* referente ao erro.

Figura 7 — Diagrama da classe Math



Fonte: O autor (2022)

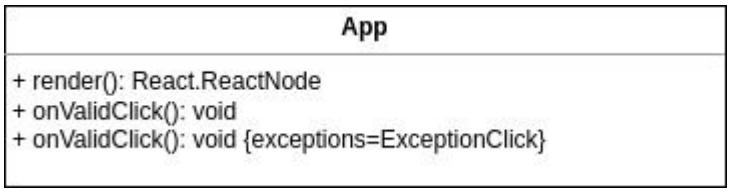
3.2.4 **Aplicando os aspectos em *React***

Para checar se a implementação de POA possui suporte a *frameworks* atuais de criação de interfaces *web*, foi escolhido o *React.* Para o teste base foi inicializada uma aplicação utilizando a ferramenta auxiliar *create-react-app*[5], após a aplicação inicial ser gerada, a classe principal *App* foi modificada para seguir o diagrama da Figura 8, dessa forma, além do método *render* que renderiza os elementos html na página, foi adicionado um método de *onValidClick* à classe e um método *onExceptionClick*, esses métodos são chamados por dois botões diferentes inseridos na página. Ao clicar no primeiro botão,

5 create-react-app - Ferramenta para criar aplicações em React executando apenas um comando, disponível em https://create-react-app.dev/

apenas o aspecto de *Log* deve ter seus adendos acionados, enquanto no botão de exceção deve, além de fazer o log, acionar o adendo do tratamento de erro. Além disso, como foi modificada a classe gerada pelo *framework*, há também o método *render* que no *React* é chamado a cada vez que é necessário atualizar a interface atual, portanto, pelo menos uma vez o adendo *before* e *after* do Log serão acionados por este método.

Figura 8 — Classe App do teste com React



Fonte: O autor (2022)

3.3 IMPLEMENTAÇÕES DE POA PARA *JAVASCRIPT*

Neste tópico serão analisadas as funcionalidades de cada ferramenta de POA, a forma como foi implementada e, se foi possível, realizar o experimento descrito e o resultado.

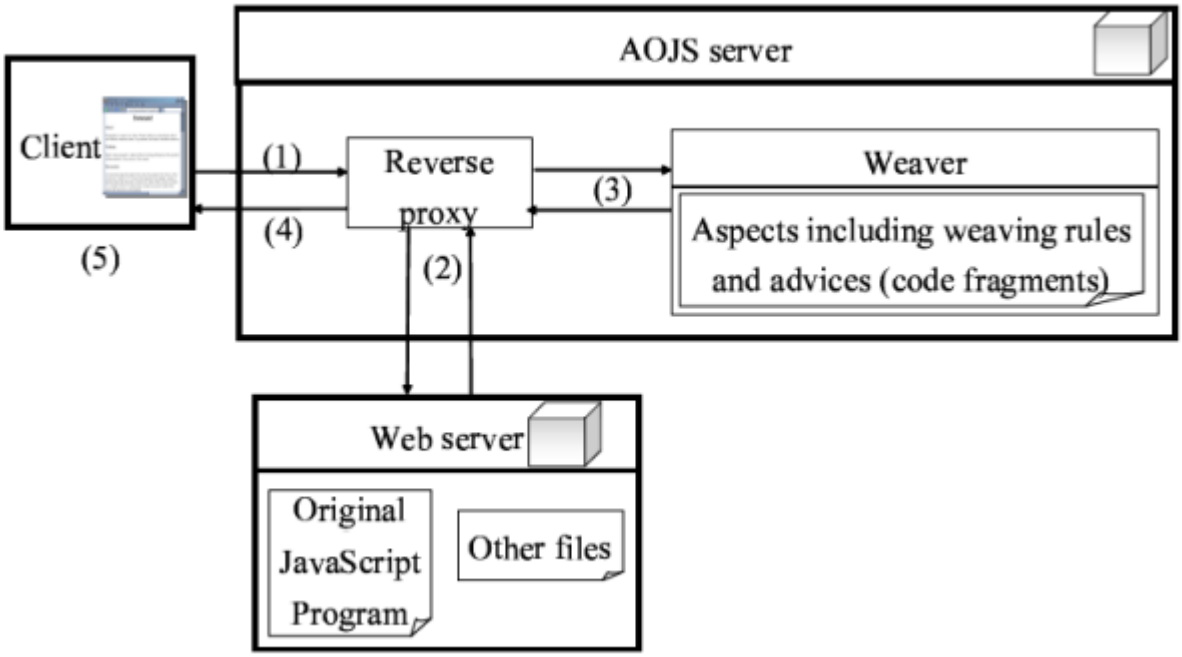
3.3.1 ***AOJS***

A ferramenta *AOJS* (WASHIZAKI et al., 2009) surgiu através de uma publicação em 2009 para solucionar problemas de implementações anteriores em *JavaScript*, principalmente para permitir a separação total do código fonte e dos aspectos. Foi descrito na publicação, que é possível os aspectos atuarem nos pontos de junção de variáveis globais, funções e inicializações, além de utilizar os adendos de *before* e *after*. A separação do código foi alcançada pela criação de uma camada adicional, que fará o intermédio de comunicação, entre cliente e o servidor *web* origem. Os servidores intermediários são chamados também de servidores de proxy reverso[6], e apenas sua *URL* ficará disponível para o cliente.

A Figura 9 mostra como é feita a comunicação orquestrada pelo servidor intermediário, primeiro o cliente faz uma requisição ao novo servidor, o servidor por sua vez, carrega a página do servidor *web*, mas antes de retornar ao cliente, o servidor intermediário executa o *Weaver* do *AOJS* nos códigos *JavaScript* da página carregada, aplicando os aspectos, que foram definidos em um arquivo no formato *XML* (e*Xtensible Markup Language*). Após o processo de costura dos aspectos terminar, a página é retornada ao cliente, finalmente podendo visualizar a página web.

6 Um servidor de proxy reverso é um servidor intermediário que atua em função do servidor destino ou de origem, é utilizado normalmente por sites para balanceamento de carga, proteção contra ataques e cacheamento.Um servidor de proxy direto ou de encaminhamento é um servidor intermediário que atua em função do cliente, utilizado para restringir o acesso a internet por instituições ou governos, controle de conteúdo e, também, para proteger a identidade online do cliente.

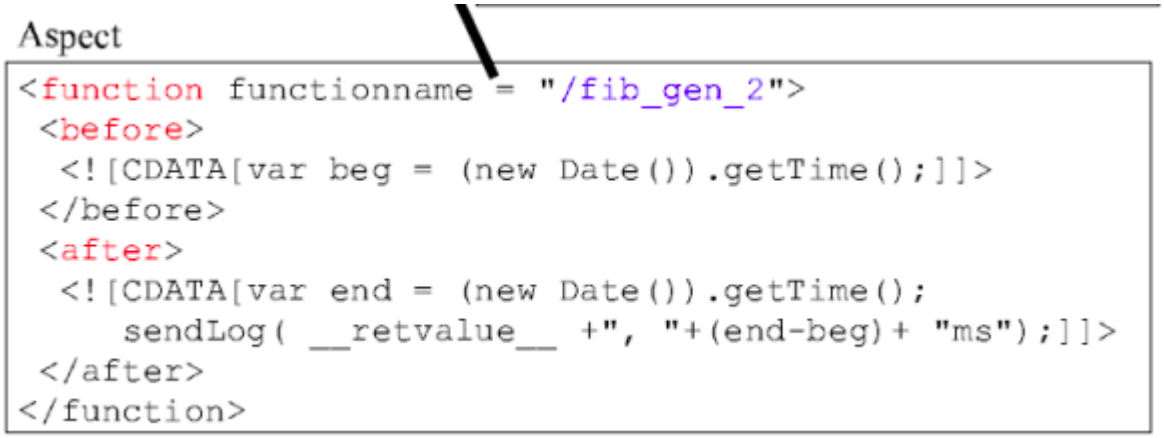
Figura 9 — Arquitetura AOJS



Fonte: Washizaki et al. (2009, p. 3)

O teste utilizando essa ferramenta não foi possível, pois fora a publicação não há referências sobre a mesma, não sendo possível encontrar um link de download e documentação, dessa forma, a ferramenta não se mostra uma boa alternativa para desenvolvedores que estão aprendendo a utilizar este novo paradigma. Além disso, apesar de utilizar *XML*, que é uma das linguagens de marcação mais utilizadas, é necessário misturar *XML* com *JavaScript* para obter os resultados desejados, o que prejudica a leitura e entendimento do código, como é possível visualizar no Código 3. Neste exemplo o aspecto está atuando em uma função que gera a Sequência de Fibonacci, antes de gerar, ou seja, foi criado um adendo *before* que inicia uma variável com o tempo de início e, após execução, adendo *after,* calcula o tempo que levou e envia o *log* com o valor de retorno da função somado ao tempo que levou para executar.

Código 3 — Exemplo de aspecto em AOJS



Fonte: Washizaki et al. (2009, p. 5)

3.3.2 ***AspectJS***

A ferramenta *AspectJS* (VAUGHAN, 2007) foi projetada e desenvolvida por Richard Vaughan e distribuída pela empresa Dodeca *Technologies* Ltd em 2007. Essa ferramenta implementa interceptação de chamadas de funções da linguagem *JavaScript*, utilizando uma estratégia de *proxy*, isto é, uma forma de adicionar uma nova função intermediária, entre a função alvo e uma nova chamada a função, isso se concretiza ao mudar a referência alvo para apontar para a intermediária, mas antes, garante que a função intermediária continue chamando a referência anterior da função alvo. A biblioteca foi desenvolvida em 2007 e, por isso, não usa o *Proxy* que é implementado no *ECMAScript* 2015 (ECMA INTERNATIONAL, 2015).

O método intermediário pode adicionar o prefixo, referente ao adendo *before*, ou sufixo, referente ao adendo *after*, em funções do código em qualquer momento de sua execução. Um exemplo de como adicionar um prefixo a uma função *myFunc()* pode ser vista no Código 4, no exemplo é adicionada a função principal um prefixo referente ao adendo de *before* que será executado antes da função referenciada for chamada.

Código 4 — Programação orientada a aspectos AspectJS

```
function prefixFunc(){
    /* something */
}
function myFunc(){
    /* something */
}
AJS.addPrefix(this,"myFunc",prefixFunc);
myFunc();
```

Fonte: Huang, He e Li (2015, p. 1)

Testes utilizando essa ferramenta não foram realizados, pois a ferramenta é paga, portanto está fora do escopo do projeto de facilitar a adoção de programação orientada a aspectos para novos desenvolvedores, e, além disso, o direito autoral do site que contém a documentação não é atualizado desde 2015, o que indica que a ferramenta não está tendo atualizações e manutenções frequentes.

3.3.3 ***aspectjs***

O *aspectjs* (FORD, 2015), assim como o *AspectJS*, apresentado no item anterior, é implementado utilizando *proxy*. Essa ferramenta é uma biblioteca *open-source* desenvolvida em 2015 por Philip Ford e disponível para instalação através do gerenciador de pacotes para *node.js*, o *NPM*[7]*.* A ferramenta implementa os adendos de *before*, *around* e *after* para pontos de junção de chamadas de métodos, utilizando uma sintaxe simples que será apresentada nas implementações do teste base.

Apesar de ser simples, a ferramenta apresenta alguns pontos negativos por misturar o código fonte com o código dos aspectos e, principalmente, não provê uma forma de definir pontos de corte, sendo necessário para cada um dos adendos, adicioná-los diretamente ao componente alvo, não existindo assim, um conceito bem definido de aspecto na implementação.

O Código 5 mostra a implementação do *LoggerAspect* do teste base, nele é possível ver a criação do aspecto utilizando uma classe padrão da linguagem, os nomes dos métodos utilizados seguem o padrão de adendos *before*, *after* e *around,* mas poderiam ser completamente arbitrários, o processo de costura desses adendos são realizados nas últimas linhas do código, com a utilização de métodos simples, que recebem os nomes dos adendos padrões de POA com argumentos referentes ao ponto de junção do sistema, referentes a classe *Math* e métodos *add* ou *divide.* Após definição do ponto de junção, é necessário chamar o método *add()* com argumentos referentes ao aspecto. Foi necessário para um único aspecto e apenas dois pontos de junção, fazer 12 chamadas de métodos,

7 NPM - Gerenciador de para JavaScript, acesso em: https://www.npmjs.com/

pensando em um escopo com uma aplicação que deseja fazer o tratamento de erro em todos os componentes dela, seria quase inviável, apenas com os recursos previstos pela ferramenta, registrar cada um dos métodos que se deseja tratar erro.

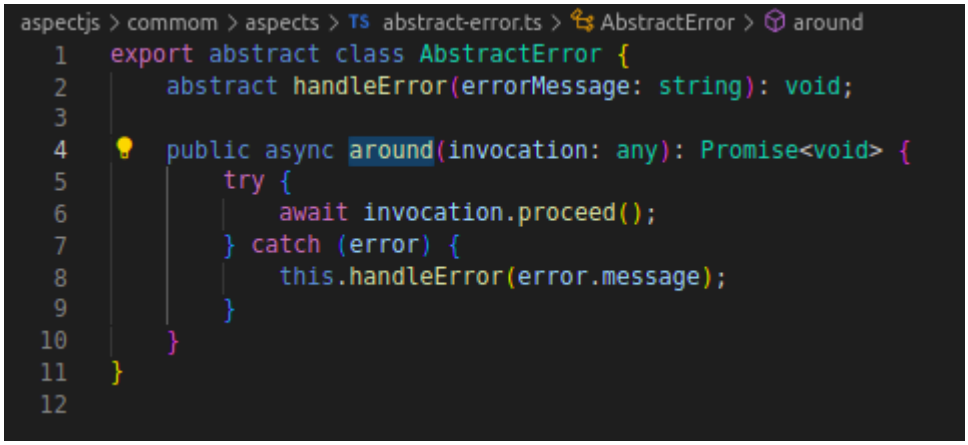
Código 5 — LoggerAspect aspectjs

```
import aspectjs from "aspectjs";
import Math from "../math";

class LoggerAspect {
    before(arg1: any, arg2: any) {
        console.log(`Chamou algum método com os argumentos: ${arg1} ${arg2}`);
    }
    after(arg1: any, arg2: any) {
        console.log(
            `Retornou após chamar algum método com os argumentos: ${arg1} ${arg2}`
        );
    }
    around(invocation: any) {
        console.log(
            `Chamou método '${
                invocation.name
            }' com os argumentos: ${Object.values(invocation.args).toString()}`
        );
        const result = invocation.proceed();
        console.log(`Retornou ${result} depois de chamar '${invocation.name}'`);
    }
}

const loggerAspect = new LoggerAspect();
aspectjs.before(Math, "add").add(loggerAspect, "before");
aspectjs.before(Math, "divide").add(loggerAspect, "before");
aspectjs.around(Math, "add").add(loggerAspect, "around");
aspectjs.around(Math, "divide").add(loggerAspect, "around");
aspectjs.after(Math, "add").add(loggerAspect, "after");
aspectjs.after(Math, "divide").add(loggerAspect, "after");

```

Fonte: O autor (2022)

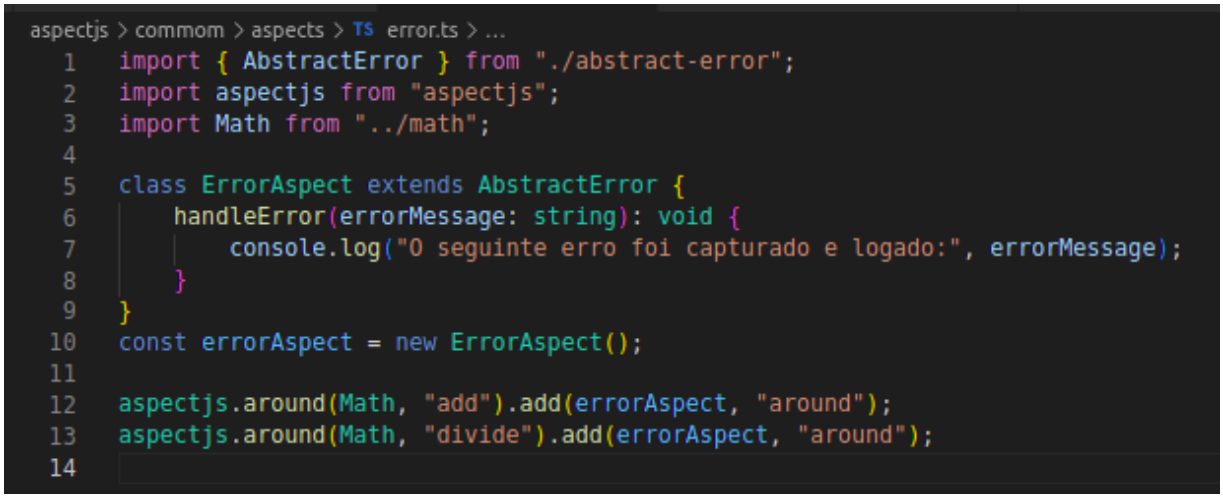
No Código 6 é possível ver a criação da classe abstrata de erro utilizando apenas os recursos da linguagem *TypeScript*, e no Código 7 a classe concreta que estendeu a classe abstrata de erro criada, implementando o método abstrato *handleError* para imprimir a mensagem de erro no *console.* Pode-se notar que a ferramenta não provê um adendo para tratar disparo de exceção, fez-se necessário, utilizar um adendo *around* e em seu código capturar se houve um erro.

Código 6 — AbstractError aspectjs



Fonte: O autor (2022)

Código 7 — ErrorAspect aspectjs

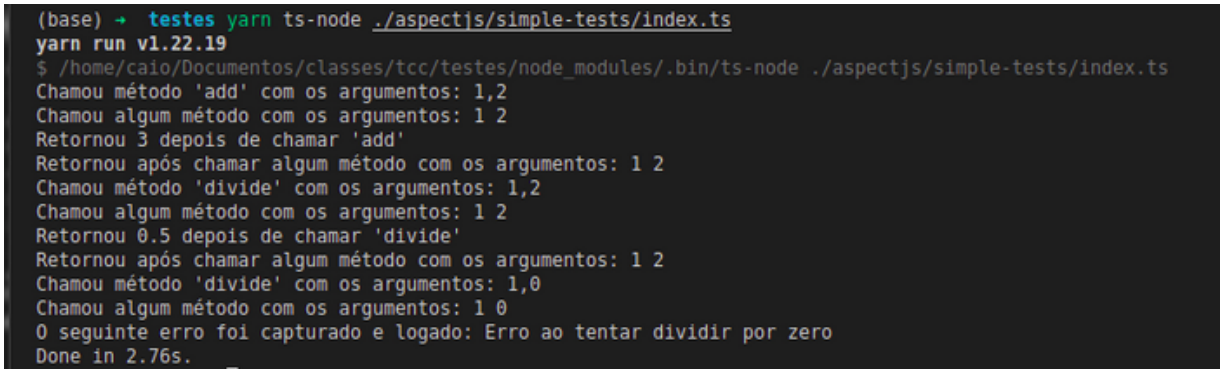


Fonte: O autor (2022)

Outro problema dessa implementação é a falta de metadados, os adendos de *before* e *after* só conseguem acessar os argumentos, enquanto no *around* é possível acessar além dos argumentos, o nome do método e o resultado ao invocar o *proceed*. Além disso, a ferramenta não possui tipagem do *TypeScript* implementada, prejudicando o entendimento e uso.

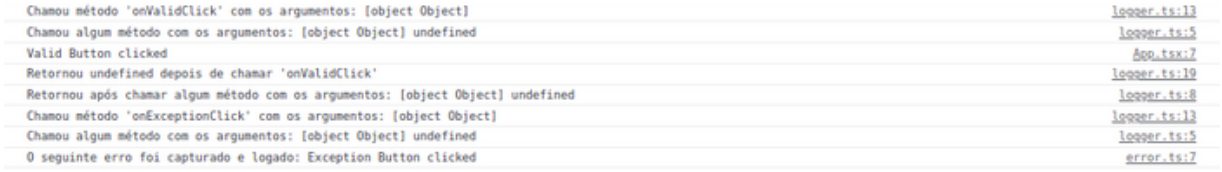
Apesar da limitação de recursos que a ferramenta dispõe, foi possível obter o resultado esperado, como pode ser visto nas Figura 10 e Figura 11, nos *consoles* o dobro de impressões estão sendo exibidas, pois foi mantido o adendo extra *around.* A aplicação *web* também apresentou mais limitações, foi necessário realizar uma pequena modificação para o funcionamento do teste, já que a classe *App* é instanciada internamente pelo *React* e a ferramenta depende da instância da classe para definir os pontos de corte, portanto, outra classe foi criada para acessar os métodos dos eventos dos botões.

Figura 10 — Terminal teste aspectjs



Fonte: O autor (2022)

Figura 11 — Console do navegador teste de aspectjs



Fonte: O autor (2022)

3.3.4 ***aspect.js***

O *aspect.js* (GECHEV, 2015) é uma implementação que utiliza a própria linguagem para sua implementação, ela faz o uso das novas sintaxes de decoradores disponíveis no *ECMAScript* 2016 (ECMA INTERNATIONAL, 2016). Essa implementação fornece um conjunto amplo de funcionalidades de POA, possibilitando utilizar os adendos de *before*, *around*, *after*, *onThrow* e *onThrowAsync, que podem ser* aplicados aos pontos de junção de chamadas de métodos de uma classe, para chamadas de métodos estáticos de uma classe e para acessadores (*Getter* e *Setter*) de uma classe. Além disso é uma biblioteca de código aberto e disponível para instalação através do *NPM*, a biblioteca foi implementada utilizando o *TypeScript*, portanto ao utilizar a ferramenta é possível utilizar a tipagem que facilita no seu entendimento e uso.

Porém, como a maioria das outras implementações, essa implementação mistura o código fonte com o código dos aspectos, e por utilizar os decoradores do *JavaScript* a extensão das classes de aspecto não funciona da maneira esperada, não sendo possível implementar aspectos reaproveitáveis, como será observado no experimento realizado. Por último, é importante ressaltar que desde sua publicação em 2015, a ferramenta foi atualizada mais de uma vez por ano, porém não há atualizações recentes, sendo a última realizada em 2019.

No Código 8 é possível visualizar a implementação da classe *LoggerAspect,*

diferentemente da implementação vista para o *aspectjs*, a ferramenta provê uma forma definir pontos de corte, isto é feito através dos parâmetros dos decoradores *@beforeMethod* e *@afterMethod,* na própria escolha do decorador é definida em qual tipo de junção irá atuar, nesse caso antes e depois da chamada de métodos, além disso, foi passado como parâmetro duas expressões regulares, o primeiro é um padrão para selecionar apenas pontos de junção das classes desejadas e o segundo é um padrão para definir os métodos.

Código 8 — LoggerAspect aspect.js

```
aspect.js > react-test > src > aspects > TS logger.ts > ...
1   import { afterMethod, beforeMethod, Metadata } from "aspect.js";
2
3   export class LoggerAspect {
4       @beforeMethod({
5           classNamePattern: /.*/,
6           methodNamePattern: /.*/,
7       })
8       before(meta: Metadata) {
9           console.log(
10              `Chamou método '${meta.method.name}' da classe '${meta.className}' com os argumentos: ${meta.method.args}`
11          );
12      }
13
14      @afterMethod({
15          classNamePattern: /.*/,
16          methodNamePattern: /.*/,
17      })
18      after(meta: Metadata) {
19          console.log(
20              `Retornou ${meta.method.result} depois de chamar '${meta.method.name}' da classe '${meta.className}'`
21          );
22      }
23  }
24
```

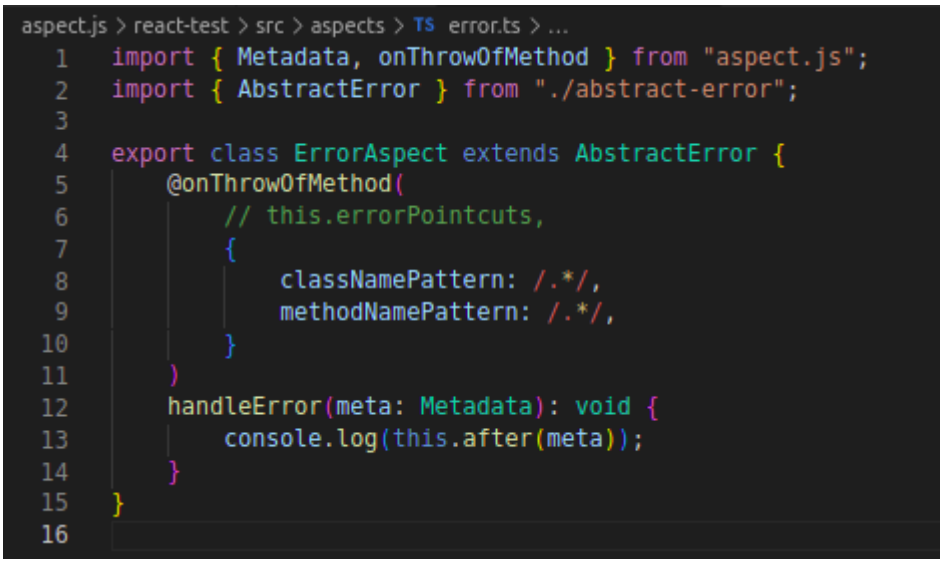
Fonte: O autor (2022)

No Código 9 e Código 10 é possível visualizar as implementações dos aspectos para erro utilizando *aspect.js*, a parte de reaproveitamento do aspecto precisou ser adaptada necessitando redeclarar os pontos de corte no aspecto concreto. Foi definido que o aspecto concreto deveria implementar o *handleError,* que é responsável somente por imprimir o erro no *console*, e o adendo *after* para tratamento de erro seria declarado no aspecto abstrato e o chamaria com a mensagem de erro, portanto utilizando a ferramenta, o decorador *@onThrowOfMethod* deveria estar no aspecto abstrato antes do método *after*, porém ao implementar da maneira proposta a ferramenta considerou que o *AbstractError* era um aspecto válido e ignorou o seu aspecto concreto *ErrorAspect, dessa forma o handleError* não será implementado, o que originou o erro mostrado na Figura 13.

Código 9 — AbstractError aspect.js

Fonte: O autor (2022)

Código 10 — ErrorAspect aspect.js

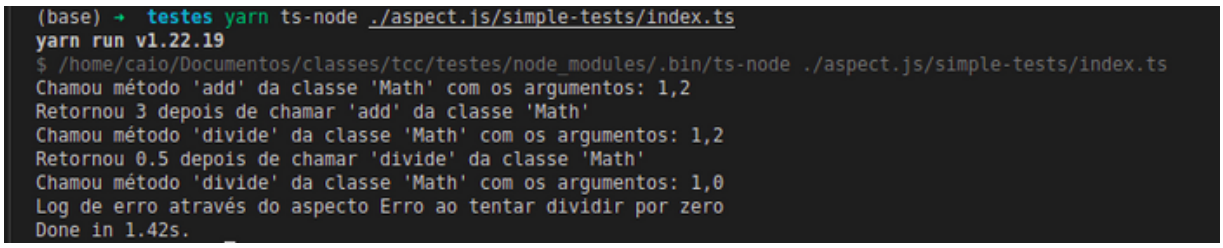


Fonte: O autor (2022)

Para concluir o experimento as funções foram adaptadas de forma que o método *handleError* virou o adendo de disparo de erro, dessa forma o método *after* da classe abstrata, agora não mais sendo utilizado como adendo, perdeu seu propósito, mas continuou sendo utilizado para verificar se é possível criar aspectos que estendem qualquer outra classe que não utiliza os decoradores. Com as modificações realizadas pela limitação da ferramenta, foi possível concluir o experimento, como é possível visualizar na Figura 12 referente ao que foi impresso no terminal e na Figura 14 referente ao que foi impresso no

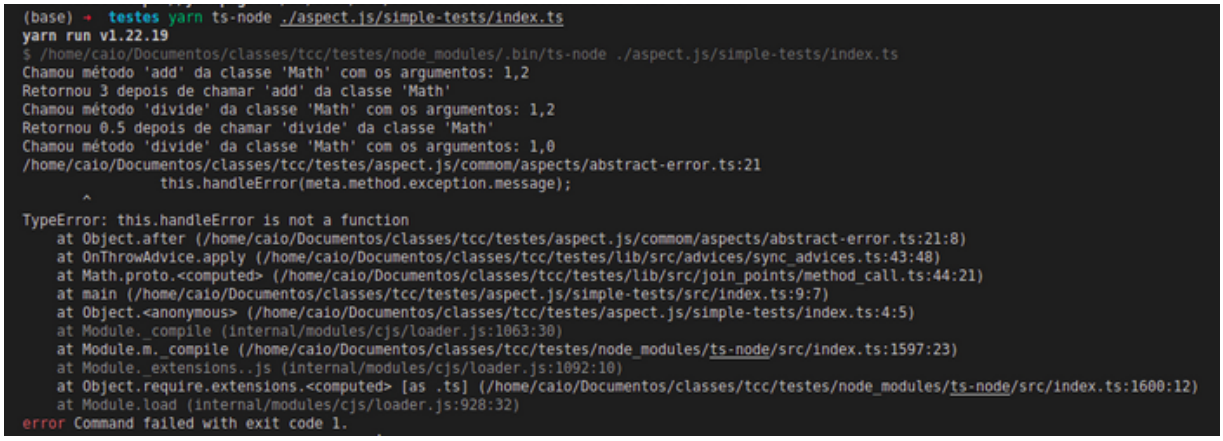
console do navegador. Portanto a implementação não permite criar aspectos reutilizáveis, isto é, ela só permite definir as regras em um aspecto ou classe concretos.

Figura 12 — Resultado teste simples de aspect.js - modificado



Fonte: O autor (2022)

Figura 13 — Teste simples de aspect.js - original



Fonte: O autor (2022)

Figura 14 — Teste do aspect.js utilizando React

Fonte: O autor (2022)

3.3.5 ***AspectScript***

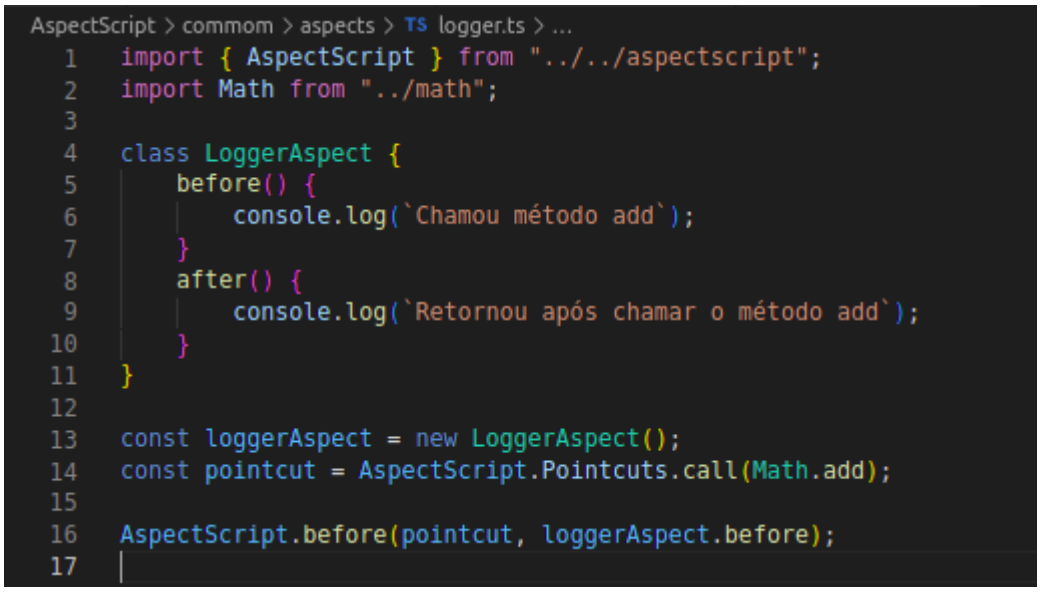
A ferramenta *AspectScript* (LEGER; TOLEDO, 2010) surgiu da necessidade de integrar os aspectos com algumas funcionalidades básicas do *JavaScript* como funções de primeira classe e de ordem superior, tipagem dinâmica e programação baseada em prototipagem. A implementação dos aspectos nessa ferramenta são realizados por meio de

funções normais do *JavaScript*, o que permite se beneficiar totalmente das funcionalidades da linguagem. Sua API de uso não é diferente das que foram vistas até agora e como pode ser visto nas implementações exemplos: Código 11 e Código 12.

Código 11 — Exemplo implementação AspectScript

Fonte: O autor (2022)

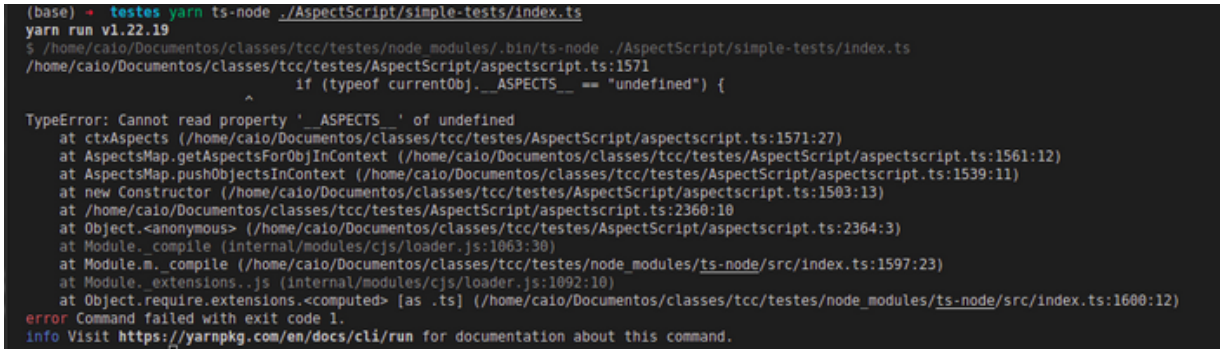
Código 12 — LoggerAspect AspectScript



Fonte: O autor (2022)

Está datado nas páginas de documentação e download da ferramenta que não há modificações desde 2010, mostrando que a ferramenta não apresenta manutenção recorrente e implementações de novas funcionalidades há algum tempo. A falta de manutenção se mostrou um problema ao executar o próprio código que a ferramenta dispõe, um erro interno na ferramenta foi disparado como pode ser visto na Figura 15, portanto o experimento completo proposto, não pôde ser concluído para o *AspectScript*.

Figura 15 — Erro ferramenta AspectScript



Fonte: O autor (2022)

3.3.6 ***Jackdaw***

O *Jackdaw* (SILVA, 2019) é a ferramenta desenvolvida mais recente, e foi desenvolvida por Ricardo Sá Loureiro Ferreira da Silva e supervisionado pelos doutores João Bispo e Tiago Carvalho. A ferramenta tem como principal objetivo garantir que o código dos aspectos não seja invasivo, isso é alcançado pelo desenvolvimento dos aspectos ser em uma linguagem diferente chamada LARA (PINTO et al., 2017), esta linguagem utilizada foi desenvolvida para permitir a criação de aspectos de maneira independente da linguagem utilizada para os componentes ou interesses principais do sistema, dessa forma o código fonte é mantido na forma original e é gerado um código destino após o processo de tecelagem dos aspectos no código fonte. No Código 13 é possível visualizar um exemplo adicionando um prefixo em uma função *fun1* com uma impressão no *console* de uma mensagem como visto em outros exemplos utilizando a ferramenta *Jackdaw*.

Código 13 — Exemplo de um aspecto escrito utilizando Jackdaw

```
1 aspectdef addPrefix
2
3   select function.blockStatement end
4   apply
5     if($function.name == "fun1"){
6       $blockStatement.children[0].insert("before","console.log('prefix code.');");
7     }
8   end
9 end
```

Fonte: Silva (2019)

Não foi possível fazer experimentos utilizando a ferramenta, apesar do download estar disponível e estar à disposição no site uma experimentação online, não há uma documentação ou tutorial de como instalar e rodar uma aplicação utilizando a ferramenta, ainda não sendo uma boa opção para novos desenvolvedores. Além disso, apresenta os mesmos problemas que o *AOJS* de adicionar código *JavaScript* no meio do código de declaração dos aspectos prejudicando a leitura e compreensão, ademais, exige o

desenvolvedor aprender uma nova sintaxe para utilizar POA e propõe uma forma que foge do padrão das implementações vistas e não utilizar termos claros na declaração do aspecto.

3.4 COMPARAÇÃO

Após o estudo realizado e experimentação das ferramentas é possível determinar onde cada ferramenta se enquadra seguindo os critérios definidos previamente, sendo possível visualizar na Tabela 1 a classificação de cada ferramenta.

A total separação do código fonte e dos aspectos é apenas alcançado pelas ferramentas *Jackdaw* que utiliza *LARA* para criar os aspectos e o *AOJS* que faz o uso do *XML*.

A brevidade que diz respeito a curva de aprendizado para utilizar a ferramenta tem como destaque positivo a implementações baseadas em *proxy aspectjs* e *AspectJS*, assim como o *AspectScript*, que fornecem APIs simples de utilizar e são implementadas utilizando a própria linguagem *JavaScript*. Em seguida nesse critério, se destaca a implementação do *aspect.js*, por também fazer o uso da própria linguagem para definir os aspectos, porém como observado por Silva (2019), a sintaxe dos decoradores propostos pela ferramenta em conjunto com o código normal da linguagem, fazem o código final dos aspectos se tornarem confusos. O *Jackdaw* e *AOJS* apesar de não serem invasivos ao código fonte envolvem outras linguagens, dessa forma, exigiria o desenvolvedor entender outra linguagem.

No critério da maturidade se destaca positivamente o *aspect.js* que cobre a maior parte das funcionalidades básicas de programação orientada a aspectos, enquanto as programações baseadas em *proxy* possuem o escopo limitado e não alcançam o sucesso das anteriores. O *AspectScript* não está com o projeto sendo mantido, o que ocasionou uma falha no experimento realizado, portanto obteve também uma classificação negativa na Tabela 1. As implementações que não utilizam *JavaScript* tem como principal contrapartida a falta dos recursos disponibilizados pela linguagem.

Apenas a implementação *aspect.js* possui suporte a *TypeScript* implementado e os experimentos só foram possíveis nas ferramentas *aspect.js* e *aspectjs*, ambas funcionando em *React*, todas as outras ferramentas apresentaram barreiras para que o experimento fosse realizado.

Tabela 1 — Comparação ferramentas existentes

	AOJS	AspectJS	aspectjs	aspect.js	AspectScript	Jackdaw
Invasividade	+	-	-	-	-	+
Brevidade	-	++	++	+	++	-
Maturidade	-	-	-	+	-	-

Fonte: O autor (2022)

3.5 CONCLUSÃO

Das ferramentas estudadas, *AOJS* e *Jackdaw, são as* que resolvem o problema de invasividade do código fonte*,* mas demonstram possuir uma curva de aprendizado maior para novos desenvolvedores. Isso porque, as mesmas necessitam de uma maior configuração para uso, além de ser necessário aprender uma sintaxe de linguagem diferente e apresentarem API's de uso mais complexas, dessa forma não são opções vantajosas para programadores iniciarem suas experiências com POA.

A maioria das outras implementações utilizam somente *JavaScript*, apesar de se mostrarem fáceis de utilizar, disponibilizam funcionalidades limitadas, prejudicando a escalabilidade do uso da ferramenta. Por outro lado, a ferramenta *aspect.js*, cobre uma maior quantidade de funcionalidades da POA, mas como mencionado anteriormente, faz o uso de decoradores para declarar o comportamento do aspecto, fazendo com que sua API de uso não seja tão eficiente quanto a *AspectJS*, *aspectjs* e *AspectScript*, dificultando o uso dos recursos que a ferramenta disponibiliza.

Outro fator essencial para que o uso da POA seja mais difundido entre os desenvolvedores é possibilitar a criação de aspectos de maneira reaproveitável, das ferramentas que concluíram o teste o *aspect.js* não possibilitou esta criação, já o *aspectjs* possibilitou por só utilizar os próprios recursos da linguagem, mas não definiu uma forma clara de criar um aspecto, apenas definiu uma forma de inserir adendos um a um. A possibilidade de criar aspectos desta maneira terá impacto na reutilização do código, estratégia necessária para adoção do paradigma em grandes projetos. Além disso, nos tempos atuais a contribuição e criação de bibliotecas de código aberto está crescendo ano a ano, portanto com a possibilidade de reutilização de aspectos, novos aspectos podem ser disponibilizados desta maneira e difundidos entre diferentes projetos, fomentando o uso da programação orientada a aspectos.

4 **FERRAMENTA PROPOSTA: *ASPECTTS***

No capítulo anterior foram analisadas as ferramentas atuais e levantados alguns pontos nos quais as ferramentas prejudicam o uso do paradigma por novos desenvolvedores. A nova ferramenta proposta neste projeto, nomeada *aspectts*, seguirá como base a implementação mais madura das analisadas a *aspect.js*, mas removendo o uso de decoradores na implementação dentro dos códigos dos aspectos, buscando facilitar o uso e entendimento do aspecto sendo criado, diminuindo assim a curva de aprendizado para novos desenvolvedores. Além disso, a ferramenta vai buscar utilizar formas claras e auto-descritivas para declarar o aspecto utilizando termos já consolidados de POA. Por último, na ferramenta *aspectjs* foi possível concluir o experimento utilizando abstração, isso foi feito com os próprios recursos do *TypeScript*, portanto a ferramenta proposta irá aproveitar da mesma estratégia para permitir a reutilização dos aspectos.

4.1 TECNOLOGIAS UTILIZADAS

Neste tópico serão listados as tecnologias mais relevantes utilizadas para implementar a ferramenta proposta. Para cada tecnologia será listado seu nome em conjunto com uma breve descrição e como foi utilizado.

4.1.1 ***TypeScript***

Como mencionado ao longo do texto, *TypeScript* é uma extensão da linguagem *JavaScript* que enriquece a linguagem com a tipagem estática opcional, interfaces e outras funcionalidades que visam auxiliar o programador. A ferramenta proposta foi inteiramente implementada utilizando a linguagem, tanto o código fonte da ferramenta, quanto os exemplos desenvolvidos a utilizam, dessa forma o código dos aspectos é o mesmo do código fonte diferentemente do *Jackdaw* e do *AOJS*.

4.1.2 ***Jest***

O *Jest*[8] é um *framework* de testes para *TypeScript* desenvolvido pelo Facebook, a ferramenta foi utilizada para realizar testes unitários e de integração da ferramenta desenvolvida.

4.2 PROTOCOLOS ESSENCIAIS PARA ENTENDIMENTO E USO DA FERRAMENTA

Neste tópico serão apresentados os protocolos, também chamados de interfaces, que são essenciais para compreensão e uso da ferramenta.

8 Jest - https://jestjs.io/

4.2.1 ***JoinpointType***

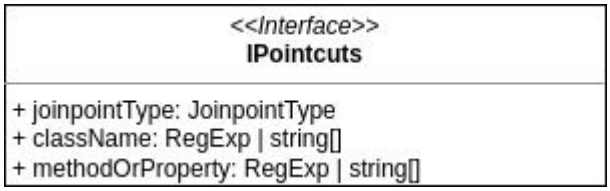
Um protocolo também foi definido para os tipos de pontos de junção implementados pela ferramenta, o protocolo neste caso apenas definem as possíveis strings aceitas a seguir:

- *MethodCall*: São os pontos de junção de chamadas a métodos de instâncias de classes;
- *StaticMethodCall*: São os pontos de junção de chamadas a métodos estáticos de classes;
- *GetterCall*: São os pontos de junção de acesso a propriedades das instância de classe, aplicado quando se faz uso do *get* do *JavaScript*;
- *SetterCall*: São os pontos de junção de alteração a propriedades das instância de classe, aplicado quando se faz uso do *set* do *JavaScript*.

4.2.2 ***IPointcuts***

A interface *IPointcuts* é responsável por definir a estrutura utilizada para definir os pontos de corte do aspecto, na Figura 16 é possível visualizar o protocolo a ser conformado ao declarar o atributo *pointcuts* do aspecto que será introduzido no 4.2.4. O protocolo proposto baseia-se no que é utilizado para *AspectJ*, definindo um tipo de ponto de junção, um padrão para classe e um padrão para o método ou propriedade, mas em vez de declarar todos de uma só vez como se fosse uma *string*, o modelo proposto é um objeto que tornará a declaração auto-descritiva e de fácil compreensão. Na figura são apresentadas três propriedades, a primeira delas diz respeito ao tipo de ponto de junção declarado no tópico anterior, a segunda delas define a regra para classes com nome que seguem o padrão da expressão regular definida, mas há também a possibilidade de utilizar um *array* de *strings*. A terceira define as regras de quais das propriedades ou métodos serão selecionadas, utilizando o mesmo formato de dados. O suporte a *array* de *strings* foi feito para facilitar a definição dos pontos de corte, visando atender programadores que não dominam expressões regulares.

Figura 16 — IPointcuts



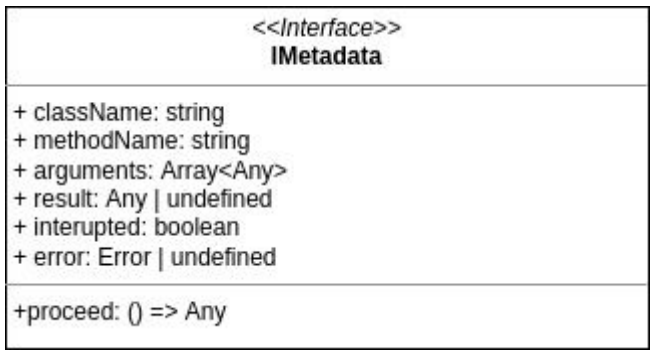
Fonte: O autor (2022)

4.2.3 ***IMetadata***

Esta interface é responsável por guardar os metadados utilizados pelos aspectos, eles possuem um papel muito importante de guardar e compartilhar dados do ponto de junção da aplicação, que por sua vez são acessados por cada um dos adendos dos aspectos mostrados na Figura 18 e que serão introduzidos no tópico subsequente.

Na Figura 17 é possível visualizar como essa interface foi definida. Sendo eles os nomes da classe e do método do ponto de junção sendo chamado, os argumentos recebidos, o resultado encontrado, que pode ainda não estar disponível dependendo do momento de execução do adendo. Além disso é possível utilizar o valor *booleano* nomeado *interrupted*, que quando alterado para Verdadeiro, irá interromper o prosseguimento das chamadas sendo realizadas e retornando o resultado atual do fluxo. Há também uma propriedade responsável por guardar o erro disparado e pode ser utilizado pelo adendo *onThrow*, que possui seu momento de execução quando um erro é disparado no ponto de junção. Por último, um método *proceed* utilizado pelo *around*, que possui momento de execução ao redor do ponto de junção e para executar ao redor, é necessário fazer uma chamada ao método disponibilizado dando sequência ao fluxo e retornando ao final.

Figura 17 — IMetadata



Fonte: O autor (2022)

4.2.4 ***IAspect***

A interface *IAspect* define a estrutura a ser seguida na criação de um aspecto que utiliza a ferramenta proposta, a interface busca simplificar a forma que foi implementada pelo *aspect.js* mas disponibilizar os mesmos recursos.

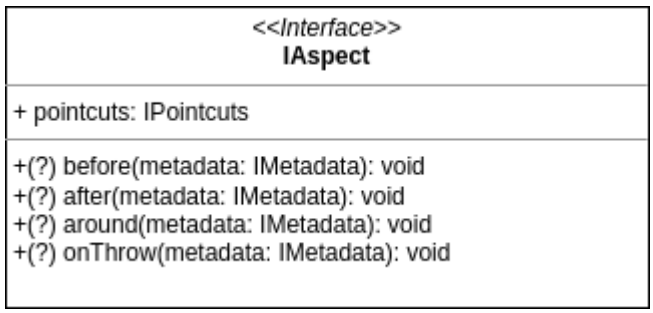
Na Figura 18 é apresentada a interface na qual os aspectos a serem desenvolvidos deverão conformar. Nela é definida uma propriedade essencial que deve ser obrigatoriamente definida o *pointcuts,* que definem os pontos de corte do aspecto nos pontos de junção dos componentes da aplicação.

Além das propriedades definidas, há também na Figura 18 quatro funções com assinaturas idênticas, representando cada um dos *Advices* implementados pela ferramenta, os adendos identificados pelo nome de cada uma das funções são os momentos em que

elas serão executadas nos pontos de junção, que por sua vez são selecionados pelos pontos de corte descritos no aspecto. Todas as funções recebem um parâmetro com os metadados, estes podem ser compartilhados por diferentes aspectos em um mesmo ponto de junção.

Ainda sobre os adendos declarados, a interface *IAspect* permite que o programador quando for criar um aspecto para sua aplicação, implemente apenas os adendos que desejar, o *TypeScript* permite a criação de interfaces que declaram assinaturas para os métodos como apresentado na Figura 18, mas também permite que continuem a não ser definidos, isto é feito na linguagem utilizando uma sintaxe com interrogação na declaração, essa sintaxe foi representada no diagrama com o (?) após o sinal de +, sendo dessa forma interpretado como: um método público que recebe o parâmetro *IMetadata* e retorna vazio, mas pode não ter sido definido.

Figura 18 — IAspect



Fonte: O autor (2022)

A estratégia de utilizar uma interface para criação dos aspectos, permite que os aspectos sejam implementados como classes padrões da linguagem *JavaScript*, sendo possível utilizar a extensão de uma classe que segue o protocolo apresentado, possibilitando assim. a reutilização de aspectos apenas com recursos disponibilizados pela linguagem. O *TypeScript*, por sua vez, introduz a linguagem os conceitos de POO *abstract* e *implements*, conceitos nos quais permitem criar classes abstratas e implementações de interfaces, portanto será possível criar um aspecto conformando com o protocolo e utilizar ele na ferramenta proposta, que terá seu comportamento introduzido no próximo tópico, dessa forma, a verificação se o aspecto conforma com o protocolo será tempo de compilação ou, até mesmo antes, se a IDE[9] para desenvolvimento do código utilizada possuir *IntelliSense*[10] para *TypeScript*. Apesar disso não ser possível em *JavaScript*, ainda será possível utilizar a ferramenta proposta, pois para todo aspecto registrado na ferramenta, será feita uma checagem em tempo de execução, que irá informar quando o aspecto não está de acordo com o protocolo e, portanto, precisa ser alterado.

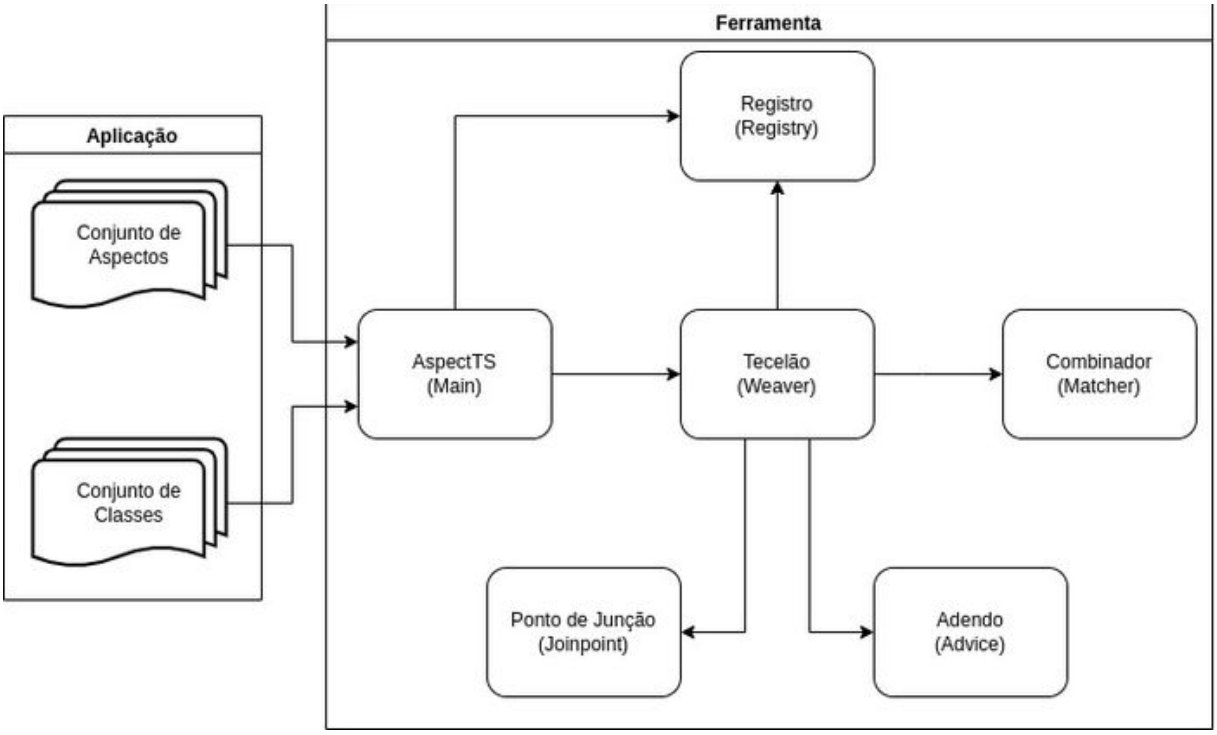
9 Um ambiente de desenvolvimento integrado (IDE) é um software para criar aplicações que combina ferramentas comuns de desenvolvimento em uma única interface gráfica do usuário.
10 IntelliSense é um nome dado a um conjunto de recursos que tornam a codificação mais conveniente.

4.3 FUNCIONAMENTO DA FERRAMENTA PROPOSTA

Assim como a maioria das soluções vistas no Capítulo 3, a ferramenta proposta foi implementada de maneira a realizar o processo de costura dos aspectos ao código fonte em tempo de execução, utilizando uma estratégia de registros de aspectos e classes por meio de métodos disponibilizados pela ferramenta. Na Figura 19 é possível visualizar os módulos mais relevantes para a implementação da ferramenta e como é feita a interação entre eles. À esquerda são mostrados os dois conjuntos a serem registrados pela aplicação que desejar utilizar POA, sendo eles, os aspectos implementados conformando com os protocolos vistos anteriormente e as classes da aplicação que serão possíveis alvos de cortes dos aspectos.

A cada novo registro feito pela aplicação, internamente na ferramenta, o aspecto ou classe são armazenados e passarão pelo processo de costura (*Weaving*). O processo de costura é a etapa central da ferramenta e será orquestrada pelo Tecelão, que interage com todos os outros módulos apresentados, sendo eles o Registro, o Combinador, o Ponto de Junção e o Adendo. Cada módulo da ferramenta será explicado em detalhes nos tópicos subsequentes.

Figura 19 — Módulos relevantes no funcionamento da ferramenta proposta



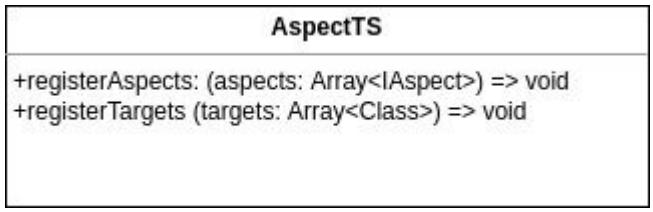
Fonte: O autor (2022)

4.3.1 ***Main***

O módulo *main* é a camada mais externa da ferramenta, sua principal função é isolar o comportamento interno, disponibilizando apenas métodos essenciais para a abordagem

adotada de POA, mencionada neste tópico. Portanto, para que a aplicação registre suas classes e também seus aspectos, foram disponibilizados dois métodos chamados *registerAspects* e *registerTargets,* como podem ser vistos na Figura 20. A função de registro de classes recebe um *array* de classes da própria linguagem *JavaScript*, enquanto o de aspectos recebe um *array* de *IAspect* mostrado anteriormente na Figura 18.

Figura 20 — AspectTS



Fonte: O autor (2022)

4.3.2 **Registro**

O módulo de Registro tem como principal função armazenar os elementos[11] registrados pela aplicação que utiliza a ferramenta. A Main a utiliza para registrar novos elementos e o Tecelão para recuperar o que já foi registrado e fazer novas costuras. O Registro também tem um papel de verificar o que está sendo registrado na ferramenta, dessa forma, só irá salvar aspectos válidos, portanto se o aspecto não estiver de acordo com o protocolo definido ou nenhum adendo ser implementado, o Registro dispensará a entrada, emitindo um aviso para que o aspecto seja revisto.

4.3.3 **Combinador**

O Combinador tem como principal função verificar as regras definidas no protocolo de pontos de corte, o *IPointcuts,* mostrado anteriormente na Figura 16, portanto ele verifica se uma *string* recebida referente ao nome da classe de um ponto de junção ou nome do método ou de uma propriedade atende às regras definidas, isto é, o Combinador irá retornar Verdadeiro no caso do nome seguir o padrão definido na expressão regular ou se o nome está incluso no *array* e Falso caso contrário, por padrão o *array* vazio obrigatoriamente fará a resposta do Combinador ser verdadeira. A verificação é realizada em duas ocasiões pelo Tecelão, primeiro para verificar o nome da classe e depois para verificar o nome dos métodos ou propriedades dos pontos de junção, nos tópicos subsequentes será explicado o motivo da checagem ser feita em dois tempos.

4.3.4 **Ponto de Junção**

Este módulo é responsável por tratar cada um dos tipos de pontos de junção

11 Elemento é a nomeclatura adotada neste capítulo para denotar a união do que representa um aspecto ou classe alvo.

definidos no tópico anterior, este módulo possui duas funções importantes para o processo de costura, primeiro ele é utilizado para retornar apenas os nomes das propriedades ou funções referentes ao que foi definido no aspecto, por exemplo, se o tipo de junção definido for de chamadas a métodos da instância da classe, ele não deve retornar os métodos estáticos da classe e vice-versa.

Além disso, após o ponto de corte ser aplicado por completo, isto é, verificou-se que o ponto de junção será definitivamente selecionado, faz-se necessário pegar a função referenciada neste ponto, caso alguma costura já tenha sido feita, apenas retornará a função referente ao ponto de junção, caso contrário significa que o método oculto *_advices_wrapping* configurado pela ferramenta não foi definido.

Na primeira costura a ser realizada no ponto de junção, o módulo substituirá a função original, a fim de criar metadados no início de sua execução, para isso, é necessário também fazer uma chamada a função oculta[12], que é criada pela ferramenta neste mesmo processo, ao final, retornará o resultado do objeto com os metadados. A função será no início a função original, aplicando os metadados nela, para ser possível, por exemplo, alterar os argumentos passados para a função original, ou para interromper sua execução. O Código 14, mostra a construção das função descritas, ou seja, a construção dos metadados, chamada a função oculta e retorno do resultado, assim como da modificação feita na chamada original.

Código 14 — Substituição inicial feita no processo de costura

```
const buildedFunction = function (...args: any[]) {
    const metadata = self.getMetadata(classTarget, propertyName, args);
    self.getSpecificJoinpointMethod(
        classTarget,
        propertyName
    )._advices_wrapping(metadata, this);
    return metadata.result;
};

buildedFunction._advices_wrapping = (
    metadata: IMetadata,
    context: Function
) => {
    if (metadata.interrupted) {
        return;
    }
    metadata.result = originalMethod.apply(context, metadata.arguments);
};
```

Fonte: O autor (2022)

Por fim, é importante mencionar a estratégia utilizada para implementar o módulo, há

12 As funções em JavaScript são também objetos, por isso é possível criar novas propriedades nestas funções, a função oculta não é um termo existente, apenas faz referência a criação dessa função não convencional que foi adicionada como propriedade de outra função.

uma classe abstrata central que implementa parte das funções do módulo, mas cada tipo de ponto de junção possui suas particularidades, sendo elas a forma de encontrar a função do ponto de junção, a forma de carregar os pontos de junção de uma classe e a substituição feita na primeira vez que o ponto de junção passa pela costura. Dessa forma, cada um dos tipos de pontos de junção: *MethodCall*, *StaticMethodCall, GetterCall* e *SetterCall* possuem uma implementação concreta diferente destes métodos.

4.3.5 **Adendo**

A principal função do módulo é finalizar o processo de costura, para isso faz um processo de empacotamento em cima da função oculta existente, a substituindo. Este processo é demonstrado pelo Código 15, que mostra a substituição sendo realizada, além de uma chamada a um método que constrói uma nova função passando como argumento a referência a função atual. Assim como feito para o Ponto de Junção, há diferentes tipos de adendo, por isso, também foi utilizado o conceito de abstração, dessa forma, cada classe concreta implementa sua particularidade que envolve, principalmente, os diferentes momentos de execução.

Código 15 — Processo de aplicação dos adendos

```
weave(joinpointMethod: IAdvice.PreparedJoinpointMethod) {
    const currentWrappingMethod = joinpointMethod._advices_wrapping;

    joinpointMethod._advices_wrapping = this.buildNewWrappingMethod(
        currentWrappingMethod
    );
}
```

Fonte: O autor (2022)

4.3.6 **Tecelão**

O Tecelão é o módulo de maior importância da ferramenta desenvolvida, e também, de qualquer ferramenta de POA, este módulo foi apresentado no capítulo 2 como *Weaver* e sua responsabilidade é orquestrar o processo de costura ou combinação dos aspectos ao código principal da aplicação, porém diferentemente do que foi proposto inicialmente por Kiczales et al. (1997) e também ilustrado na Figura 3 — que introduziram o *Weaver* como um compilador com função de gerar um novo código compilado para o sistema, fazendo a combinação das partes código fonte e código dos aspectos — o Tecelão da ferramenta proposta performa em tempo de execução da aplicação, isto é, para cada novo registro de aspecto ou classe alvo ele inicia um novo processo de costura para o novo registro feito com base nos registros feitos previamente. O Tecelão foi implementado de maneira a não limitar

a ordem em que os elementos são registrados e nem o número de vezes que os registros são feitos, ou seja, não é necessário registrar primeiros os aspectos e depois as classes e vice-versa, permitindo até mesmo alternar entre eles.

Internamente a cada novo processo de costura iniciado, o Tecelão faz o uso do Registro para carregar os elementos já cadastrados, dessa forma, para uma nova costura de aspecto são carregadas as classes alvo registradas previamente e, por sua vez, para uma nova classe são carregadas os aspectos registrados. Em ambos os casos, são formados pares para cada aspecto e classe.

Após o carregamento e formação dos pares, o processo de costura é realizado para cada um deles, este processo pode ser visualizado no Código 16, o primeiro passo é utilizar o Combinador (*Matcher*) para verificar se a classe é um dos pontos de corte descritos pelo aspecto, utilizando a regra descrita no *classNam*e. Após essa validação ser feita, o módulo de Ponto de Junção específico ao tipo de ponto de junção, que foi determinado nos pontos de corte do aspecto, é utilizado para recuperar os nomes das propriedades da classe podendo também ser métodos, a depender do tipo de junção.

Após recuperar os nomes dos pontos de junção da classe, novamente faz uma verificação em cada um dos nomes das propriedades utilizando o Combinador, desta vez com a regra do *methodOrProperty*, para os pontos de junção que continuem selecionados pelos pontos de corte, primeiro o método do ponto de junção é carregado, nesta etapa, caso seja a primeira que está acessando o método para costurar os adendos, o módulo de Ponto de Junção executa o processo de construção inicial mencionada anteriormente. Assim que tiver acesso ao método, cada um dos adendos são costurados ao ponto de junção, seguindo o processo definido no tópico anterior.

Código 16 — Código da costura de um para aspecto e classe alvo

```
private weaveAspectForTarget(aspect: IAspect, classTarget: Function) {
    if (
        !this.matcher.match(
            classTarget.prototype.constructor.name,
            aspect.pointcuts.className
        )
    )
        return;

    const joinpoint = this.joinpoints[aspect.pointcuts.joinpointType];
    const propertyNames = joinpoint.getProperties(classTarget);
    const advices = this.getAdvicesForAspect(aspect);
    propertyNames.forEach((propertyName) => {
        if (
            this.matcher.match(
                propertyName,
                aspect.pointcuts.methodOrProperty
            )
        ) {
            const joinpointMethod = joinpoint.getJoinpointMethod(
                classTarget,
                propertyName
            );
            advices.forEach((advice) => {
                advice.weave(joinpointMethod);
            });
        }
    });
}
```

Fonte: O autor (2022)

5 **TESTES**

Neste capítulo serão apresentados as aplicações referentes ao teste base na ferramenta desenvolvida, assim como outros testes adicionados para testar outras funcionalidades implementadas, no fim serão descritos como foi utilizado testes automatizados na aplicação.

5.1 TESTE BASE

Para testar a ferramenta e ser possível comparar com as ferramentas já existentes, utilizando o mesmo escopo, o teste base do capítulo 3 também foi aplicado na ferramenta desenvolvida.

Primeiramente, para demonstrar o uso da ferramenta no teste base, foi criado o aspecto mostrado no Código 17, que consiste na implementação do *LogAspect,* descrito no tópico 3.2.1. Os pontos de corte nesse aspecto foram definidos em um objeto, onde é atribuído, que o tipo de ponto de junção é de chamada de método, e, aplicados as regras de quaisquer classe e métodos, pois o array vazio por padrão adotado, sempre aceitará todos os pontos de junção. Além disso, são mostrados os adendos *before* e *after* com acesso aos metadados disponibilizados pela ferramenta, sendo utilizados neste exemplo, os que se referem ao nome do método, ao nome da classe, aos argumentos e ao resultado.

Código 17 — LogAspect aspectts

```
import { IAspect, IMetadata, IPointcuts } from "aspectts";

You, 15 seconds ago | 1 author (You)
export class LogAspect implements IAspect {
    pointcuts: IPointcuts = {
        joinpointType: "MethodCall",
        className: [],
        methodOrProperty: [],
    };
    before(metadata: IMetadata): void {
        console.log(
            `Chamou '${metadata.methodName}' da '${metadata.className}' com: ${metadata.arguments}`
        );
    }
    after(metadata: IMetadata): void {
        console.log(
            `Retornou ${metadata.result} depois de chamar '${metadata.methodName}' da '${metadata.className}'
        );
    }
}
```

Fonte: O autor (2022)

Um dos pontos mais importantes que foram discutidos ao longo deste capítulo e do anterior, foi a possibilidade de criar aspectos reutilizáveis. Para solucionar essa questão levantada, foi proposta a criação do protocolo *IAspect*, com o objetivo de criar aspectos utilizando os próprios recursos da linguagem, bastando apenas continuar conformando com o protocolo proposto. Dessa forma, foi possível implementar o teste base com reutilização de aspectos, que podem ser vistos no Código 18, nele pode-se visualizar a implementação

do aspecto de tratamento de erro. Na ferramenta desenvolvida, a captura de erro é feita pelo adendo *onThrow,* este faz uma chamada para o método abstrato *handleError*, implementado pelo aspecto concreto, também é possível ver os pontos de corte sendo definidos no aspecto abstrato, portanto é possível ver como a implementação de reutilização de aspectos está sendo feita de maneira simples e intuitiva. É importante ressaltar que era possível definir como abstrata também os pontos de corte ou adendos, não precisando implementar diretamente na classe abstrata.

Código 18 — Exemplo ErrorAspect e AbstractErrorAspect

```
import { IAspect, IPointcuts, IMetadata } from "aspectts";

You, 12 minutes ago | 1 author (You)
export abstract class AbstractErrorAspect implements IAspect {
    abstract handleError(errorMessage: string): void;
    pointcuts: IPointcuts = {
        joinpointType: "MethodCall",
        className: [],
        methodOrProperty: [],
    };
    onThrow(metadata: IMetadata): void {
        this.handleError(metadata.error.message);
    }
}

You, 4 hours ago | 1 author (You)
export class ErrorAspect extends AbstractErrorAspect {
    handleError(errorMessage: string) {
        console.log(`O seguinte erro foi capturado: ${errorMessage}`);
    }
}
```

Fonte: O autor (2022)

Por último fez-se necessário na definição da ferramenta utilizar formas de registrar aspectos e classes alvos, é recomendado que sejam iniciados o mais cedo possível da aplicação. O Código 19 mostra os dois registros sendo feitos antes do método *main.*

Código 19 — Registro de aspectos na inicialização da aplicação

```
examples > base-test-node > TS index.ts
       You, 4 hours ago | 1 author (You)
1    import { main, Math } from "./src";
2    import AspectTS from "aspectts";
3    import { LogAspect, ErrorAspect } from "base-aspects";
4    AspectTS.registerAspects([new LogAspect(), new ErrorAspect()]);
5    AspectTS.registerTargets([Math]);
6    main();
7
```

Fonte: O autor (2022)

Os testes base rodaram sem problemas como pode ser visualizado na Figura 21, no *console* de cima, mostra a aplicação *React* executada, nela o aspecto de *log* foi executada nas primeiras duas vezes antes e depois do *render*, assim como antes e depois do botão válido clicado, por fim executou antes do *onExceptionClick* e mostrou o erro capturado. Na parte debaixo mostra o que foi executado no teste em *node*, uma sequência de impressões similar é vista, na primeira operação de adicionar e dividir não acontece o erro, portanto é possível ver as impressões com *before* e *after* sendo executadas, por último a impressão acontece antes da divisão e ao detectar o erro, ele é impresso.

Figura 21 — Execução dos testes base aspectts

```
Chamou 'render' da 'App' com:                                          index.ts:10
Retornou [object Object] depois de chamar 'render' da 'App'            index.ts:15
Chamou 'render' da 'App' com:                                          index.ts:10
Retornou [object Object] depois de chamar 'render' da 'App'            index.ts:15
Chamou 'onValidClick' da 'App' com: [object Object]                    index.ts:10
Valid Button clicked                                                     App.tsx:6
Retornou undefined depois de chamar 'onValidClick' da 'App'            index.ts:15
Chamou 'onExceptionClick' da 'App' com: [object Object]                index.ts:10
O seguinte erro foi capturado: Exception Button clicked                index.ts:17
>
```

```
(base) ➜  aspect.ts git:(master) ✗ yarn base-test-node start
yarn run v1.22.19
$ yarn workspace base-test-node start
$ ts-node ./index.ts
Chamou 'add' da 'Math' com: 1,1
Retornou 2 depois de chamar 'add' da 'Math'
Chamou 'divide' da 'Math' com: 2,1
Retornou 2 depois de chamar 'divide' da 'Math'
Chamou 'divide' da 'Math' com: 1,0
O seguinte erro foi capturado: Erro ao tentar dividir por zero
Done in 2.47s.
```

Fonte: O autor (2022)

5.2 TESTE ADICIONAIS PARA DEMONSTRAÇÃO DE API DE USO

Neste tópico serão executados mais alguns testes para demonstrar o funcionamento e uso de diferentes funcionalidades implementadas pela ferramenta. Além dos exemplos apresentados nesse tópico, está disponível no Anexo A uma demonstração de utilização de um aspecto de cache.

5.2.1 **Demonstração de *around* aplicado a chamada estática**

Para esta demonstração tomemos de exemplo a lista de tarefas do 2.1.1, mas agora adicionando uma listagem das tarefas e outras pequenas alterações que podem ser vistas no Código 20, neste exemplo, sem o aspecto para iniciar e finalizar comunicação com o banco de dados, será apenas impresso que a tarefa foi registrada, listada e removida, isto pode ser visto na parte de cima da Figura 22.

Código 20 — Exemplo Tarefa teste estático com around

```
export class Tarefa {
    static tarefas: string[] = [];
    static registrar(nome: string): string {
        console.log(`${nome} registrada com sucesso`);
        Tarefa.tarefas.push(nome);
        return nome;
    }
    static remover(nome: string): void {
        console.log(`${nome} removida com sucesso`);
    }
    static listar(): void {
        Tarefa.tarefas.forEach((nome) =>
            console.log(`Tarefa listada: ${nome}`)
        );
    }
}

export const main = () => {
    const tarefa = Tarefa.registrar("Tarefa 1");
    Tarefa.listar();
    Tarefa.remover(tarefa);
};
```

Fonte: O autor (2022)

O uso do adendo *around* foi citado no capítulo 4.2.3 no protocolo de *IMetadata*, foi mencionada a existência de um método *proceed* nos metadados, que tem como principal objetivo ser utilizado por este adendo, o objetivo dele é para, após ter sido chamado inicialmente para executar ao redor, dar sequência ao fluxo de execução do ponto de junção

e retornar novamente para o adendo para executar o bloco de código após a chamada do método *proceed*. No Código 21, pode ser visto um exemplo de uso deste método, primeiro irá imprimir que começou a comunicação com o banco e dará sequência ao ponto de junção, após executá-lo, o fluxo retorna para imprimir o encerramento da comunicação. Além disso, ao adicionar o aspecto abaixo de interesse entrecortante de comunicação do banco, apenas para os pontos de junção de registrar e remover, apenas eles terão as impressões extras ao redor de início e fim de comunicação com o banco de dados, como pode ser visto na Figura 22.

Código 21 — Aspecto banco utilizando around e chamada estática

```
import { IAspect, IPointcuts, IMetadata } from "aspectts";
export class AspectoBanco implements IAspect {
    pointcuts: IPointcuts = {
        joinpointType: "StaticMethodCall",
        className: ["Tarefa"],
        methodOrProperty: ["registrar", "remover"],
    };
    around(metadata: IMetadata): void {
        console.log(`Iniciou comunicação com banco de dados!`);
        metadata.proceed();
        console.log(`Finalizou comunicação com banco de dados!`);
    }
}
```

Fonte: O autor (2022)

Figura 22 — Impressões produzidas teste utilizando tarefas e simulação de comunicação com banco de dados

```
Tarefa 1 registrada com sucesso
Tarefa listada: Tarefa 1
Tarefa 1 removida com sucesso
Done in 1.97s.
(base) ➜  around-with-static-test git:(mast
yarn run v1.22.19
$ ts-node ./index.ts
Iniciou comunicação com banco de dados!
Tarefa 1 registrada com sucesso
Finalizou comunicação com banco de dados!
Tarefa listada: Tarefa 1
Iniciou comunicação com banco de dados!
Tarefa 1 removida com sucesso
Finalizou comunicação com banco de dados!
Done in 2.43s.
```

Fonte: O autor (2022)

5.2.2 **Demonstração de uso do acessador *getter* e uso de metadados para alterar fluxo**

Para uma última demonstração de uso, podemos tomar de exemplo a criação de um usuário e impedir que seja possível visualizar sua senha. No Código 22 é possível ver uma simples aplicação que cria um usuário e depois imprime o seu nome e senha, o resultado produzido pode ser visto na primeira parte da Figura 23

Código 22 — Exemplo de criação de usuário com acessadores para nome e senha

```
export class Usuario {
    private _nome: string;
    private _senha: string;
    constructor(nome: string, senha: string) {
        console.log(`Usuario criado!`);
        this._nome = nome;
        this._senha = senha;
    }
    get nome(): string {
        return this._nome;
    }
    get senha(): string {
        return this._senha;
    }
}

export const main = () => {
    const usuario = new Usuario("Caio", "123456");
    console.log(`Nome: ${usuario.nome}`);
    console.log(`Senha: ${usuario.senha}`);
};
```

Fonte: O autor (2022)

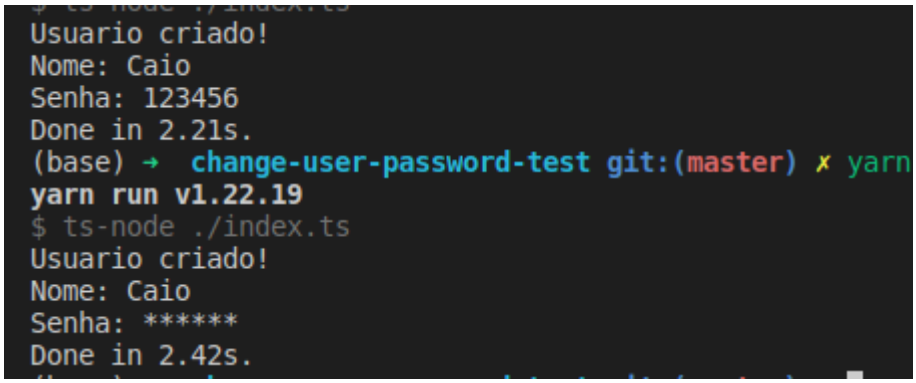
No capítulo anterior, foi mencionado a possibilidade de utilizar os metadados para alterar o comportamento do método que seria executado no ponto de junção, sendo possível manipular os argumentos, resultados e alterar o *interrupted* para interromper a execução em qualquer momento de execução do ponto de junção. O Código 23, apresenta uma implementação de um aspecto que terá seu momento de execução antes do ponto de junção de acesso a senha do usuário criado, seu comportamento descreve que o resultado será alterado para não ser retornado a real senha do usuário e sim uma *string* contendo asteriscos, em seguida, o valor de *interrupted* é alterado para Verdadeiro. Internamente a ferramenta no processo de costura, colocou uma checagem para verificar este valor, caso ele altere, o fluxo do ponto de junção é interrompido e o resultado atual armazenado no metadado é retornado, isto produzirá o efeito desejado como pode ser visto na Figura 23.

Código 23 — Aspecto com função de esconder a senha do usuário

```
import { IAspect, IPointcuts, IMetadata } from "aspectts";
export class AspectoSenha implements IAspect {
    pointcuts: IPointcuts = {
        joinpointType: "GetterCall",
        className: ["Usuario"],
        methodOrProperty: ["senha"],
    };
    before(metadata: IMetadata): void {
        metadata.result = "******";
        metadata.interrupted = true;
    }
}
```

Fonte: O autor (2022)

Figura 23 — Resultado do teste de utilização de aspecto com a função de esconder senha do usuário



Fonte: O autor (2022)

5.2.3 **Testes Automatizados**

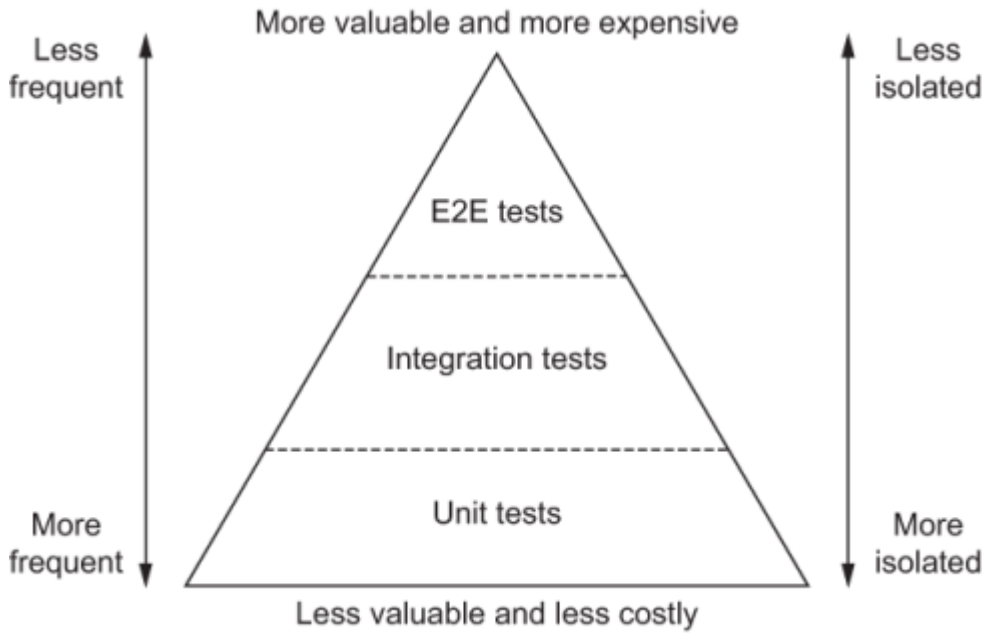
Para garantir o comportamento da ferramenta e, também, criar um pilar para manter a integridade do software em futuras implementações de novas funcionalidades, foram realizadas uma série de testes automatizados de software. Além disso, o uso dos testes tem um papel fundamental na previsibilidade do software, diminuindo o tempo gasto para detecção e depuração de um comportamento não esperado.

> Testes automatizados são programas que automatizam a tarefa de teste do seu software. Eles interagem com sua aplicação para executar ações e comparar o resultado atual ao que era esperado, que você definiu previamente. (COSTA, 2021, p. 6)[13].

13 Automated tests are programs that automate the task of testing your software. They interface with your application to perform actions and compare the actual result with the expected output you have previously defined.

Na Figura 24 abaixo é possível visualizar os 3 tipos de testes mostrados no anterior da pirâmide: os unitários (*Unit tests)*, os de integração (*Integration tests)* e os ponta a ponta (*E2E tests*), além disso, na base da pirâmide está descrito que os testes são mais frequentes, mais isolados, com menor custo, mas de menor valor, enquanto no topo é descrito o oposto. Dessa forma, os primeiros testes feitos na aplicação foram os unitários, que estão situados na base da pirâmide, esses testes foram escritos para cara um dos módulos descritos no capítulo anterior, testando unitariamente cada um dos métodos implementados e de forma isolada, isto resultou em um maior número desses testes. No centro da pirâmide está situado os testes de integração, que foram utilizados na aplicação para garantir a integração dos módulos, como são mais custosos, um menor número de testes deste tipo foram implementados.

Figura 24 — Pirâmide de teste



Fonte: Costa (2021, p. 21)

Para exemplificar os testes automatizados implementados para a ferramenta, o Código 24 mostra parte do que foi realizado para testar unitariamente o módulo Combinador, que é responsável por verificar se uma *string* está seguindo uma regra. Neste código é possível visualizar uma estrutura muito bem definida para realização dos testes, pois, "todos os testes seguem uma fórmula similar: eles preparam um cenário, disparam uma ação, e checam os resultados produzidos."[14] (COSTA, 2021, p. 54). Cada bloco de teste é delimitado pelo *it,* no primeiro dos 3, o cenário é iniciado recebendo um *sut*[15]*,* que neste caso é o Combinador, em seguida, uma ação é disparada com a chamado do *match,* capturando o resultado da verificação se a *string* segue o padrão da expressão regular, por

14 All tests follow a similar formula: they set up a scenario, trigger an action, and check the results produced.
15 System Under Test (SUT) - Sistema sob teste

fim é esperado que a resposta seja verdadeira, já que *MyClass* contém a sequência de caracteres "*My.*". No teste seguinte o nome passado é alterado, desta vez não segue o padrão e, dessa forma, foi esperado um resultado negativo. Por fim, no capítulo 4 foi definido que para um *array* vazio, o resultado será verdadeiro independentemente do nome recebido, portanto o último teste do Código 24 espera que o resultado também seja verdadeiro.

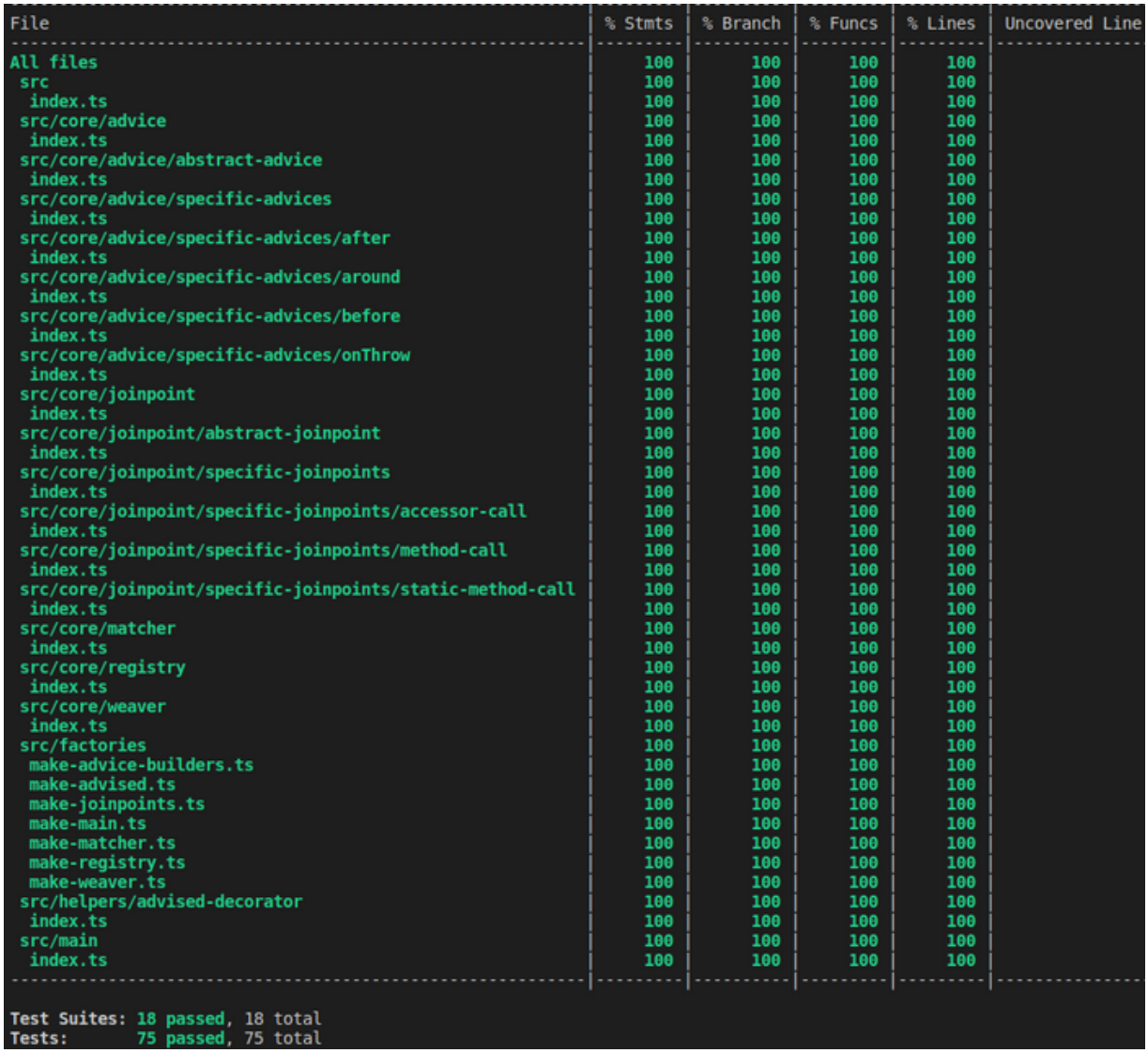
Código 24 — Parte dos testes unitários realizados para o Combinador

```
it("should return true if string matches regex rule that has been received", () => {
    const { sut } = makeSut();

    const result = sut.match("MyClass", /My.*/);

    expect(result).toBe(true);
});

Run | Debug
it("should return false if string does not match regex rule that has been received", () => {
    const { sut } = makeSut();

    const result = sut.match("AnotherClass", /My.*/);

    expect(result).toBe(false);
});

Run | Debug
it("should return true if empty array has been received", () => {
    const { sut } = makeSut();

    const result = sut.match("MyClass", []);

    expect(result).toBe(true);
});
```

Fonte: O autor (2022)

Para verificar que nenhum comportamento fora do esperado aconteça, é possível gerar relatórios sobre cobertura que os testes fazem no código fonte, na Figura 25 é possível visualizar o relatório que está sendo gerado para a ferramenta, a primeira coluna mostra os arquivos da aplicação, na segunda o percentual de declarações cobertas pelos testes, na terceira o percentual de ramificações alcançadas, na quarta o percentual de funções executadas e na quinta a percentual das linhas cobertas, finalmente, na sexta o número das linhas que não foram cobertas. Além disso, ao final da Figura 25 são exibidos o número de conjuntos de testes criados e o número de testes realizados dentro desses conjuntos. Dessa forma, os testes realizados foram capazes de cobrir todo o código fonte da ferramenta.

Figura 25 — Cobertura de testes



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line
All files	100	100	100	100	
src	100	100	100	100	
index.ts	100	100	100	100	
src/core/advice	100	100	100	100	
index.ts	100	100	100	100	
src/core/advice/abstract-advice	100	100	100	100	
index.ts	100	100	100	100	
src/core/advice/specific-advices	100	100	100	100	
index.ts	100	100	100	100	
src/core/advice/specific-advices/after	100	100	100	100	
index.ts	100	100	100	100	
src/core/advice/specific-advices/around	100	100	100	100	
index.ts	100	100	100	100	
src/core/advice/specific-advices/before	100	100	100	100	
index.ts	100	100	100	100	
src/core/advice/specific-advices/onThrow	100	100	100	100	
index.ts	100	100	100	100	
src/core/joinpoint	100	100	100	100	
index.ts	100	100	100	100	
src/core/joinpoint/abstract-joinpoint	100	100	100	100	
index.ts	100	100	100	100	
src/core/joinpoint/specific-joinpoints	100	100	100	100	
index.ts	100	100	100	100	
src/core/joinpoint/specific-joinpoints/accessor-call	100	100	100	100	
index.ts	100	100	100	100	
src/core/joinpoint/specific-joinpoints/method-call	100	100	100	100	
index.ts	100	100	100	100	
src/core/joinpoint/specific-joinpoints/static-method-call	100	100	100	100	
index.ts	100	100	100	100	
src/core/matcher	100	100	100	100	
index.ts	100	100	100	100	
src/core/registry	100	100	100	100	
index.ts	100	100	100	100	
src/core/weaver	100	100	100	100	
index.ts	100	100	100	100	
src/factories	100	100	100	100	
make-advice-builders.ts	100	100	100	100	
make-advised.ts	100	100	100	100	
make-joinpoints.ts	100	100	100	100	
make-main.ts	100	100	100	100	
make-matcher.ts	100	100	100	100	
make-registry.ts	100	100	100	100	
make-weaver.ts	100	100	100	100	
src/helpers/advised-decorator	100	100	100	100	
index.ts	100	100	100	100	
src/main	100	100	100	100	
index.ts	100	100	100	100	

Test Suites: 18 passed, 18 total
Tests: 75 passed, 75 total

Fonte: O autor (2022)

6 **ANÁLISE DA FERRAMENTA PROPOSTA**

Após serem feitos os testes base da ferramenta, assim como testes e demonstrações de uso de recursos oferecidos pela ferramenta desenvolvida, é possível fazer uma análise entre o Código 8 do *aspect.js* e o Código 17 da ferramenta desenvolvida, que estão implementando o mesmo comportamento descrito no tópico 3.2.1. Na ferramenta proposta é possível visualizar um código mais limpo e compreensível ao criar um aspecto, além de possuir um código mais auto-descritivo, isto é, os pontos de atuação, tipo de ponto de junção e as classes e métodos/propriedades e adendos, estão bem definidos no processo de criação de um aspecto, sendo possível observar cada conceito que envolve a programação orientada a aspectos. Com isso, esta forma de declaração poderá facilitar o aprendizado e fixação dos termos e conceitos por trás de POA para quem estiver iniciando os estudos e experimentos sobre o tema. Isso demonstra um potencial diferencial da ferramenta, já que, foi visto que nas ferramentas existentes, a maior parte delas, não possuem uma forma bem definida para criação de um aspecto.

Outro fator importante mencionado, foi sobre a reutilização de aspectos e solucionado pela ferramenta como apresentado no Código 18, a abordagem explorando os recursos de POO da própria linguagem *TypeScript* para criação de aspectos se mostrou uma solução simples e eficiente. Além disso, foi possível implementar uma variedade ampla de recursos, isto é, os metadados, os 4 tipos de pontos de junção e, também, 4 adendos.

Apesar dos exemplos e testes feitos, a ferramenta ainda precisa passar por um processo de validação, principalmente, pelos escopos simples definidos nas aplicações utilizadas como exemplo. Outros ponto negativo é a necessidade de registrar todos os componentes alvos da aplicação e também os aspectos, uma forma de facilitar são os decoradores do *aspect.js* antes da classe, utilizando o *@Advised,* o suporte a esse decorador foi mantido na ferramenta, mas deveria ser possível ter soluções ainda melhores que esta.

Por último, apesar da ferramenta ter sido capaz de encontrar uma solução mais clara e bem definida para o aspecto, só é possível definir uma vez os pontos de corte, no *aspect.js* era possível fazer isso para cada um dos adendos.

6.1 PUBLICAÇÃO

A ferramenta desenvolvida foi publicada e está disponível para uso através do gerenciador de pacotes NPM. Mas o nome adotado previamente no texto era similar ao de outras bibliotecas e por isso não foi possível publicar com o mesmo nome, desta forma, a biblioteca está disponível para instalação através do nome *@caiofeiertag/aspectts*[16].

16 https://www.npmjs.com/package/@caiofeiertag/aspectts

7 **CONSIDERAÇÕES FINAIS**

Entendemos que o projeto renovou os estudos sobre programação orientada a aspectos por meio da análise de ferramentas existente de *JavaScript* e propôs uma nova ferramenta em TypeScript que se mostrou eficiente na reutilização de aspectos, apresentou um conjunto amplo de recursos, implementando diferentes tipos de pontos de junção e adendos, além da possibilidade de uso dos metadados para alterações durante execução de um ponto de junção. Além disso, a ferramenta se mostrou compatível com o *framework React* e para ela desenvolvemos uma base de testes automatizados que abrem caminho para que novas funcionalidades de POA sejam comportadas pela ferramenta.

Além disso, interpretamos que a ferramenta apresentou uma melhora na API de uso em comparação com outras ferramentas, principalmente o fato de ter se tornando mais auto-descritiva, podendo assim auxiliar na compreensão e fixação dos conceitos de POA. Porém, é importante ainda validar esta interpretação, além do uso da ferramenta em si.

Um fator preocupante foi a baixa disponibilidade das ferramentas apresentadas e baixa manutenção, além de ainda não existir uma ferramenta madura como o *AspectJ* para *TypeScript* ou *JavaScript*. Dessa forma, pode ser de interesse de programadores começarem a testar novas ferramentas de POA, mesmo que ainda em suas primeiras versões.

7.1 TRABALHOS FUTUROS

Entendemos que ainda há diversas funcionalidades que a ferramenta pode implementar, dentre elas a implementação do ponto de junção de chamada a construtor, tratar chamadas assíncronas, suporte para métodos e não somente classes e registro de classes alvo de forma automática. Além disso, como mencionado anteriormente, ainda é importante validar a API de uso da ferramenta através do teste com outros programadores, principalmente por quem está buscando aprender o paradigma.

Outro trabalho futuro que complementaria o estudo feito, seria verificar o desempenho das ferramentas estudadas e da ferramenta desenvolvida. Para continuar a facilitar a adoção do paradigma é importante prover um ferramental para auxiliar no desenvolvimento de testes automatizados nas aplicações que utilizam as ferramentas que implementam suporte a POA. Também seria interessante fazer um estudo sobre ou criar uma ferramenta para *JavaScript* que permita a costura em tempo de compilação, já que, a desenvolvida no projeto, foi feita em tempo de execução.

REFERÊNCIAS

ALMEIDA, João. **Desenvolvimento de Software Orientado a Aspectos**. Monografias UFJF. Juiz de Fora, Minas Gerais, 2007. 88 p. Disponível em: http://monografias.ice.ufjf.br/tcc-web/exibePdf?id=8. Acesso em: 20 set. 2021.

BIERMAN, Garvin; ABADI, Martín; TORGENSEN, Mads. Understanding TypeScript*. In:* JONES, R. (EDS) ECOOP 2014 – OBJECT-ORIENTED PROGRAMMING. Springer, Berlin, Heidelberg, 2014.

CARVALHO, Thiago. **Orientação a Objetos**: Aprenda seus conceitos e suas aplicabilidades de forma efetiva. Editora Casa do Código, 2016.

COSTA, Lucas da. **Testing JavaScript Applications**. Manning Publications, f. 256, 2021. 512 p.

ECMA INTERNATIONAL. **ECMAScript 2015 Language Specification**. http://www.ecma-international.org. Geneva, 2015. Disponível em: https://262.ecma-international.org/6.0/. Acesso em: 15 jul. 2022.

ECMA INTERNATIONAL. **ECMAScript 2016 Language Specification**. http://www.ecma-international.org. 2016. Disponível em: https://262.ecma-international.org/7.0/. Acesso em: 14 jul. 2022.

FIGUEIREDO, Eduardo. **Uma Abordagem Quantitativa para Desenvolvimento de Software Orientado a Aspectos**. Rio de Janeiro, 2006 Dissertação - Pontifícia Universidade Católica do Rio de Janeiro.

FORD, Philip. **aspectjs**. 2015. Disponível em: https://github.com/pford68/aspectjs. Acesso em: 6 jun. 2022.

GECHEV, Minko. **aspect.js**. 2015. Disponível em: https://github.com/mgechev/aspect.js. Acesso em: 6 jun. 2022.

GOETTEN, Junior; WINCK, Diogo. **AspectJ**: Programação Orientada a Aspectos com Java. São Paulo - SP: Novatec, 2006.

GRADECKI, Joseph D.; LESIECKI, Nicholas. **Mastering AspectJ**: Aspect-Oriented Programming in Java. John Wiley & Sons, f. 228, 2003. 456 p.

HILSDALE, Erik; HUGUNIN, Jim. **Advice Weaving in AspectJ**. 2004. Disponível em: https://dl.acm.org/doi/pdf/10.1145/976270.976276. Acesso em: 14 jul. 2022.

HUANG, Wenhao; HE, Chengwan; LI, Zheng. A Comparison of Implementations for AspectOriented JavaScript . *In:* INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND INTELLIGENT COMMUNICATION (CSIC 2015) . 2015. 2015.

HÜRSCH, Walter; LOPES, Cristina. **Separation of concerns**. CiteSeerX. Boston, Massachusetts, Estados Unidos da América, 1995. 21 p. Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.2723&rep=rep1&type=pdf. Acesso em: 20 set. 2021.

KICZALES, Gregor *et al*. An Overview of AspectJ. **ECOOP 2001 --- Object-Oriented Programming**, Berlin, Heidelberg, p. 327-354, 2001. Disponível em:

https://doi.org/10.1007/3-540-45337-7_18. Acesso em: 14 jul. 2022.

KICZALES, Gregor *et al*. Aspect-oriented programming*. In:* AKŞIT, M., MATSUOKA, S. (EDS) ECOOP'97 — OBJECT-ORIENTED PROGRAMMING, Berlin, Heidelberg, 1997, p. 220-242.

KICZALES, Gregor. **Aspect-oriented programming**. ACM Digital Library . 1996, p. 154-es. Disponível em: https://doi.org/10.1145/242224.242420. Acesso em: 30 set. 2021.

KURDI, Heba. **Review on Aspect Oriented Programming**. (IJACSA) International Journal of Advanced Computer Science and Applications. Arabia Saudita, 2013. 6 p. Disponível em: https://pdfs.semanticscholar.org/e14f/f4ef93475eebc90d9909e8e5ebffb7d6a056.pdf. Acesso em: 30 set. 2021.

LADDAD, Ramnivas. **AspectJ in Action**: Practical Aspect-oriented Programming. 2 ed. Manning Publications Company, f. 241, 2003. 512 p.

LEGER, Paul; TOLEDO, Rodolfo. **AspectScript**. Pleiad Lab. 2010. Disponível em: https://pleiad.cl/aspectscript/. Acesso em: 6 jun. 2022.

META PLATFORMS, INC.. **React**. 2013. Disponível em: https://pt-br.reactjs.org/. Acesso em: 30 nov. 2021.

MICROSOFT. **TypeScript**. Redmond, Boston, SF & Dublin, 2012. Disponível em: https://www.typescriptlang.org/. Acesso em: 30 set. 2021.

PINTO, Pedro *et al*. LARA as a language-independent aspect-oriented programming approach. **Proceedings of the Symposium on Applied Computing**, Marrakech, Morocco, p. 1623-1630, 2017. Disponível em: https://doi.org/10.1145/3019612.3019749. Acesso em: 1 ago. 2022.

RESENDE, Antônio; SILVA, Claudiney. **Programação Orientada a Aspectos em Java**. Rio de Janeiro: Brasport, v. 1, f. 88, 2005. 176 p.

SILVA, Kelli. **Análise comparativa entre programação orientada a objetos e orientada a aspectos**. FURB Universidade de Blumenau. Blumenau, 2005. 95 p. Disponível em: http://dsc.inf.furb.br/arquivos/tccs/monografias/TCC2005-2-13-VF-KelliABBDSilva.pdf. Acesso em: 20 set. 2021.

SILVA, Ricardo Sá Loureiro Ferreira da. **Aspect-Oriented Programming for Javascript using the Lara Language**. Portugal, 2019. 59 p Tese (Engenharia Informática e Computação) - Faculdade de Engenharia Universidade do Porto. Disponível em: https://repositorio-aberto.up.pt/bitstream/10216/122210/2/351106.pdf. Acesso em: 6 jun. 2022.

SILVA, Wendel. **O Estado da Arte em Programação Orientada a Aspectos. São José dos Campos**. São José dos Campos, 2004 Trabalho de Conclusão de Curso.

SOARES, Sérgio; LAUREANO, Eduardo; BORBA, Paulo. **Implementing Distribution and Persistence Aspects with AspectJ**. 2002 Dissertação - Federal University Of Pernambuco.

ULLMAN, Larry. Primeiro Capítulo*. In:* ULLMAN, Larry Edward. **Modern JavaScript**: Develop and Design. Peachpit Press, f. 306, 2011. 611 p.

VAUGHAN, Richard. **AspectJS**: Proxies in JavaScript. AspectJS. Londres, Reino Unido,

2007. Disponível em: https://www.aspectjs.com/. Acesso em: 4 ago. 2022.

WASHIZAKI, Hironori *et al*. **AOJS**: Aspect-Oriented JavaScript Programming Framework for Web Development. Tokyo, Japão, 2009, p. 31-36. Disponível em: https://www.researchgate.net/publication/234803056_AOJS_Aspect-Oriented_JavaScript_programming_framework_for_web_development. Acesso em: 4 ago. 2022.

GLOSSÁRIO

weaver	O termo em inglês 'weaver' é utilizado para referenciar o processo de tecer o código dos componentes ao código fonte
before	O termo em inglês 'before' é utilizado para referenciar o adendo executado antes
around	O termo em inglês 'around' é utilizado para referenciar o adendo executado ao redor
after	O termo em inglês 'after' é utilizado para referenciar o adendo executado após

ANEXO A — EXEMPLO DE CACHE PARA UMA PÁGINA WEB QUE MOSTRA MODELOS DE CARROS

Um caso de uso que aplica Programação Orientada a Aspectos é a aplicação do interesse entrecortante de cacheamento de dados retornados de uma API, para isso foi desenvolvido um exemplo em uma aplicação React, que mostra modelos de carros e que faz requisição a uma API para carregar os modelos. Para demonstrar o uso da reutilização e abstração de aspectos, foi desenvolvido um aspecto de cache abstrato que pode ser utilizado em qualquer aplicação que necessita do interesse entrecortante. Além disso, foi desenvolvido um aspecto concreto para indicar os pontos de cortes da aplicação dos modelos de carros. Além dos códigos mostrados abaixo, é possível rodar o exemplo desenvolvido através do repositório <https://github.com/CaioFeiertag/aspectts-cache-demo>.

No código abaixo é possível a implementação do aspecto abstrato de cache para o exemplo, definindo os adendos antes para retornar o resultado já cacheado e o depois para salvar o resultado.

```
import { IMetadata, IAspect, IPointcuts } from "@caiofeiertag/aspectts";

export abstract class AbstractCacheAspect implements IAspect {
    abstract pointcuts: IPointcuts;

    before(metadata: IMetadata) {
        const key = this._getCacheKey(metadata);
        if (this._cache[key]) {
            metadata.result = this._cache[key];
            metadata.interrupted = true;
            console.log("Retornou resultado cacheado!");
        }
    }
    after(metadata: IMetadata) {
        const key = this._getCacheKey(metadata);
        if (!this._cache[key]) {
            Promise.resolve(metadata.result).then((result) => {
                this._cache[key] = result;
                console.log("Salvou resultado no cache!");
            });
        }
    }

    private _cache: Record<string, any> = {};
    private _getCacheKey(metadata: IMetadata) {
        return `${metadata.className}.${metadata.methodName}`;
    }
}
```

No código abaixo é possível ver o aspecto concreto de cache aplicado ao ponto de corte da requisição a API e estendendo o aspecto visto na imagem anterior. Além disso, é

possível visualizar como é feito o registro de aspectos utilizando a ferramenta.

```
import { IPointcuts } from "@caiofeiertag/aspectts";
import { AbstractCacheAspect } from "./AbstractCacheAspect";
import AspectTS from "@caiofeiertag/aspectts";

class CacheAspect extends AbstractCacheAspect {
    pointcuts: IPointcuts = {
        joinpointType: "MethodCall",
        className: ["CarModelApi"],
        methodOrProperty: ["getModels"],
    };
}
AspectTS.registerAspects([new CacheAspect()]);
```

No código abaixo é possível ver a implementação da classe responsável por fazer a requisição a API de modelos de carros e mostrando o registro da classe também utilizando a ferramenta na última linha.

```
import AspectTS from "@caiofeiertag/aspectts";

const carBrands = [
    "volkswagen",
    "chevrolet",
    "hyundai",
    "toyota",
    "jeep",
    "honda",
    "fiat",
];
export class CarModelApi {
    async getModels(): Promise<Record<string, string[]>> {
        const models: Record<string, string[]> = {};
        for (const brand of carBrands) {
            const response = await fetch(
                `https://vpic.nhtsa.dot.gov/api/vehicles/getmodelsformake/${brand}?format=json`
            );
            const json = await response.json();
            models[brand] = json.Results.map(
                (result: { Model_Name: string; url: string }) =>
                    result.Model_Name
            );
        }
        return models;
    }
}
AspectTS.registerTargets([CarModelApi]);
```