CURSO DE CIÊNCIA DA COMPUTAÇÃO – TCC	
(X) PRÉ-PROJETO ( ) PROJETO	ANO/SEMESTRE: 2022/2

AGREGADORES DE MÓDULOS TYPESCRIPT: UMA SOLUÇÃO EM TEMPO DE COMPILAÇÃO

Luis Felipe Zaguini Nunes Ferreira

Prof. Marcel Hugo – Orientador

1 INTRODUÇÃO

Segundo pesquisa (STACK OVERFLOW, 2021), o TypeScript é uma linguagem de programação amplamente utilizada por desenvolvedores. Seu principal trunfo é a adição de tipos ao JavaScript, estendendo a linguagem em um conceito chamado de *superset*. O TypeScript inclui um compilador e seu código-alvo é JavaScript, tornando possível a execução dos artefatos em ambientes que não necessariamente suportem tipos, como navegadores para Internet (FREEMAN, 2021).

Por ser um *superset*, o TypeScript herda a funcionalidade JavaScript de importação e exportação de símbolos (funções, constantes, classes, e outros tipos de elementos) através de arquivos, chamados de módulos (MOZILLA, 2022). O desenvolvedor pode exportar algum símbolo em determinado módulo e acessá-lo por meio de uma declaração de importação, desta forma utilizando tal símbolo em um outro módulo dentro do projeto. Essa funcionalidade permite maior modularidade e, por consequência, uma arquitetura da informação que faça mais sentido do ponto de vista do desenvolvedor.

Porém, em projetos de grande escala, não é incomum ver um número considerável de linhas com declarações de importação, o que prejudica a legibilidade e manutenção da base de código. Por causa disso, um padrão de projeto que se tornou popular foi a utilização de *barrels* – "barris", em tradução livre do inglês. Na definição de Basarat (2017, tradução nossa), "os barris são uma maneira de acumular exportações de vários módulos em um único módulo conveniente. O próprio barril é um arquivo de módulo que reexporta exportações selecionadas de outros módulos". Posto de outra forma, um arquivo de barril é um módulo auxiliar que contém exportações a serem importadas posteriormente de maneira simplificada. Esses módulos agregadores reduzem a redundância na importação de símbolo pertencentes a um mesmo domínio, porém em módulos diferentes.

O principal problema dessa técnica é a necessidade de definir explicitamente, com um módulo auxiliar, os símbolos a serem exportados. Esse módulo não tem utilidade semântica para o projeto, e o processo de criação e manutenção deste é, geralmente, manual. Por ser algo que precisa ser mantido pelos desenvolvedores, é propenso a erros, potencialmente introduzindo problemas na aplicação. Além disso, é um processo maçante que não agrega valor à base de código.

Diante do contexto apresentado, este trabalho propõe o desenvolvimento de um conjunto de ferramentas que mantenha a experiência de desenvolvimento proporcionada pela utilização de barris, mas que torne dispensável a necessidade de criação e manutenção de arquivos auxiliares.

1.1 OBJETIVOS

Este trabalho tem como objetivo disponibilizar um conjunto de ferramentas a serem utilizadas tanto por desenvolvedores finais quanto por criadores de ferramentas com o propósito de melhorar a experiência de desenvolvimento de aplicações TypeScript.

Os objetivos específicos são:

a) disponibilizar um *plugin* que se conecte ao compilador TypeScript, possibilitando a manipulação de código-fonte para resolver barris implícitos, isto é, substituir importações agregadas por importações individuais com caminho real em tempo de compilação;
b) disponibilizar um *plugin* para o Servidor de Linguagem do TypeScript que possibilite a resolução de barris implícitos em tempo de desenvolvimento, proporcionando uma experiência de desenvolvedor ótima enquanto dentro do ambiente de desenvolvimento integrado (IDE).

2 TRABALHOS CORRELATOS

A seguir são apresentadas ferramentas com características semelhantes aos principais objetivos do estudo proposto.

A primeira ferramenta é um executável que gera arquivos auxiliares (barris) TypeScript sob demanda (COVENEY, 2017). A segunda ferramenta também gera barris TypeScript, com a diferença de que observa mudanças nos arquivos e atualiza os arquivos auxiliares em tempo real (WANG, 2017). Finalmente, a terceira

ferramenta, que é a mais similar ao projeto proposto, é um transformador de código-fonte em tempo de execução para o *framework* de testes Jest: o transformador analisa a entrada (código-fonte TypeScript), aplica transformações nos usos de barris e executa o código transformado, sem um barril envolvido (SINGH, 2022).

Todas as ferramentas citadas, com exceção da última, foram submetidas ao mesmo ambiente de testes, ilustrado na Figura 1, com o objetivo de explorar funcionalidades e seus resultados.

Figura 1 – Estrutura e conteúdo dos arquivos do ambiente de teste.

Fonte: elaborado pelo autor.

Em sentido horário: o módulo `components/box.ts` exporta o símbolo `BoxDefault` de maneira padrão, assim como o símbolo `BoxConst` de maneira nomeada.

O módulo `components/data.json` é um arquivo *JavaScript Object Notation* (JSON), que pode ser importado e será tratado como um objeto JavaScript em tempo de execução.

O módulo `components/spacer.ts` exporta o símbolo `SpacerConstForwarded` através de uma declaração de exportação com especificadores, além de exportar o símbolo `SpacerConst` como um pseudônimo chamado `SpacerConstAliased`. Além disso, também exporta o símbolo `SpacerProps` de maneira nomeada.

O módulo `other/box.ts`, exporta de maneira padrão o símbolo `BoxDefaultFromOther`, assim como o símbolo `BoxConstFromOther` de maneira nomeada.

O módulo `other/info.ts` exporta de maneira padrão o símbolo `InfoObject` e, por último, o módulo `other/missing-identifier.ts` exporta de maneira padrão um símbolo sem nome específico.

2.1 BARRELSBY: BARRIS TYPESCRIPT AUTOMÁTICOS PARA SUA BASE DE CÓDIGO

Essa ferramenta de *software* livre disponibilizada por Coveney (2017) se trata de um executável que permite gerar de forma recursiva – isto é, em diretórios aninhados – barris para bases de código TypeScript. Ele expõe uma variedade de configurações, como "modos de execução": criar o barril apenas no diretório especificado, ou no diretório especificado e nos subdiretórios, entre outros.

Como ponto forte, pode-se mencionar a facilidade de utilização e a forma pouco invasiva em que a ferramenta atua: não é necessário modificar o projeto em que deseja implementar a funcionalidade, apenas executá-la e os arquivos de barris são gerados.

Como ponto fraco, há a necessidade de executar a ferramenta a cada alteração na estrutura do projeto – módulos, símbolos exportados, ou ambos – a fim de manter o barril atualizado: a geração de barris não é automática. Um outro ponto negativo, se comparado ao projeto proposto, é a necessidade de ter um arquivo físico para os barris, o que é, como mencionado acima, inútil de um ponto de vista semântico.

Testando o Barrelsby com o comando `barrelsby -d components -d other -E`, especificando pastas com o argumento `-d` e especificando que também se quer exportar os símbolos padrões através do argumento `-E`, obtém-se o resultado ilustrado na Figura 2.

Figura 2 – Resultado da execução do Barrelsby.

Fonte: elaborado pelo autor.

Se reparar no barril gerado para a pasta `components` (`components/index.ts`), percebe-se que apesar de o módulo `components/spacer.ts` não ter uma exportação padrão declarada, ainda assim foi incluído no barril gerado pelo Barrelsby, o que demonstra que o algoritmo de geração de barril é ingênuo, não verificando a existência de exportações padrão antes de incluí-las. Como resultado, tem-se um erro de compilação, afinal não é possível reexportar um símbolo que não existe. Além disso, o símbolo da exportação padrão do módulo `components/box.ts`, nomeado `BoxDefault`, foi reexportado como `box`, o que potencialmente causará confusão aos ~~usuários finais (no caso, desenvolvedores)~~ devido a essa diferença no nome dos símbolos. Finalmente, o módulo `data.json` não foi incluído no barril.

No caso do barril para a pasta `other` (`other/index.ts`), além dos problemas supramencionados, a reexportação do símbolo padrão do módulo `other/missing-identifier.ts` é inválida, pois apenas letras e números podem ser utilizados como identificadores. Note que a IDE já reporta o erro ao desenvolvedor, apesar do executável do Barrelsby não reportar quaisquer erros.

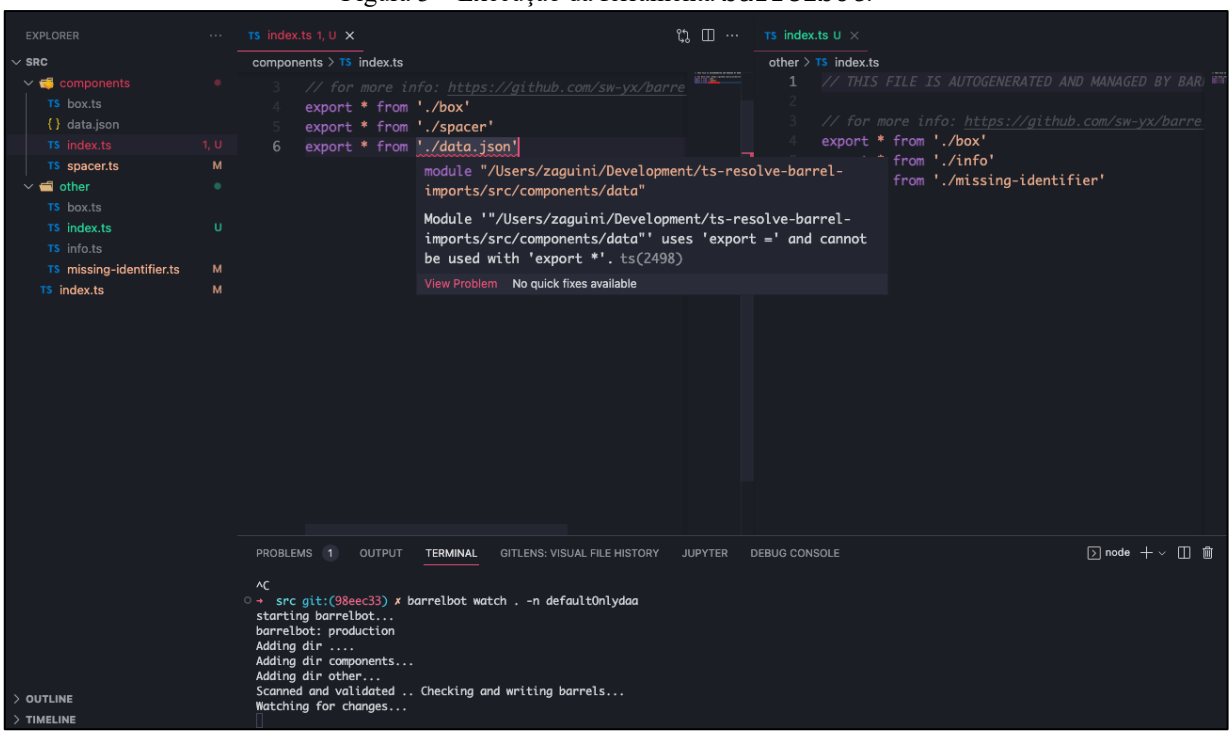
2.2 BARRELBOT: O GERENCIADOR DE BARRIS AUTOMATIZADO: RECURSIVAMENTE OBSERVA PASTAS E GERA BARRIS PARA SEU PROJETO JAVASCRIPT OU TYPESCRIPT

A ferramenta de *software* livre criada por Wang (2017) observa mudanças nos diretórios e atualiza os barris gerados automaticamente.

O comando `barrelbot watch .` foi utilizado para gerar arquivos auxiliares em todas as pastas descendentes da pasta principal, e o resultado pode ser observado na Figura 3. Nota-se de imediato que as reexportações de exportações padrão não foram incluídas. Além disso, o arquivo `data.json` foi tratado como um módulo comum e todos os seus símbolos foram reexportados, apesar de ter apenas uma exportação padrão por ser um arquivo JSON, invalidando o barril.

Na versão atual (v0.0.6), a documentação afirma que é possível reexportar os símbolos de exportação padrão através do argumento `--namespace defaultOnly`. Porém, ao adicionar este argumento no comando utilizado, não há diferença na saída. Mesmo se houvesse diferença, segundo a documentação, todas as reexportações seguiriam o modelo de reexportação de exportação padrão (isto é, `export { default as foo } from './foo'`) e isso também não seria desejado, afinal as exportações nomeadas seriam ignoradas.

Figura 3 – Execução da ferramenta `barrelbot`.



Fonte: elaborado pelo autor.

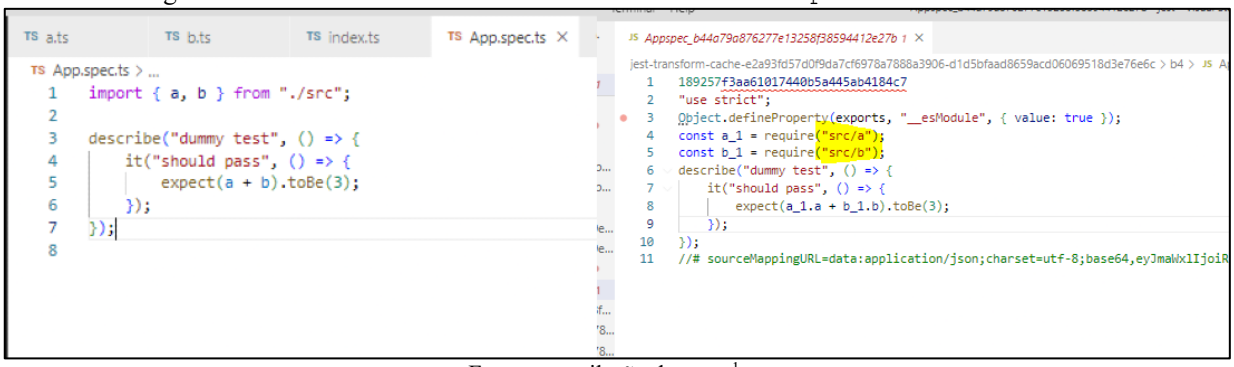
2.3 TS-BARREL-IMPORT-TRANSFORMER: EXEMPLO DE TRANSFORMADOR DE IMPORTAÇÕES DE BARRIS

Essa ferramenta-exemplo – a que mais se parece com o projeto proposto – criada por Singh (2022) expõe um transformador de código-fonte em tempo de execução.

A ferramenta Jest, um *framework* de testes para o ecossistema JavaScript, expõe um conjunto de configurações que possibilitam conectar na transformação de código-fonte. Com isso, é possível inserir, modificar ou remover declarações antes que o código seja executado pelo interpretador JavaScript. Tudo isso em memória, ou seja, os arquivos não são emitidos, porque o objetivo do Jest é rodar suítes de teste, não gerar código-fonte.

O `ts-barrel-import-transformer` utiliza destes transformadores e se conecta ao Jest para alterar as declarações de importação de barris. Uma vez fornecido o código-fonte, a ferramenta cria uma árvore sintática TypeScript através das ferramentas expostas pelo próprio TypeScript e visita os nós dessa árvore, procurando por declarações de importação por barril. Ao encontrar, realiza a substituição dessas declarações por declarações de importação individuais com os símbolos respectivos e seus caminhos completos. Na Figura 4 é possível observar o código-fonte original (esquerda, em TypeScript) e o código-fonte transformado (direita, em JavaScript), salvo na memória e executado pelo Jest enquanto roda as suítes de teste.

Figura 4 – Entrada e saída do transformador `ts-barrel-import-transformer`.



Fonte: compilação do autor[1].

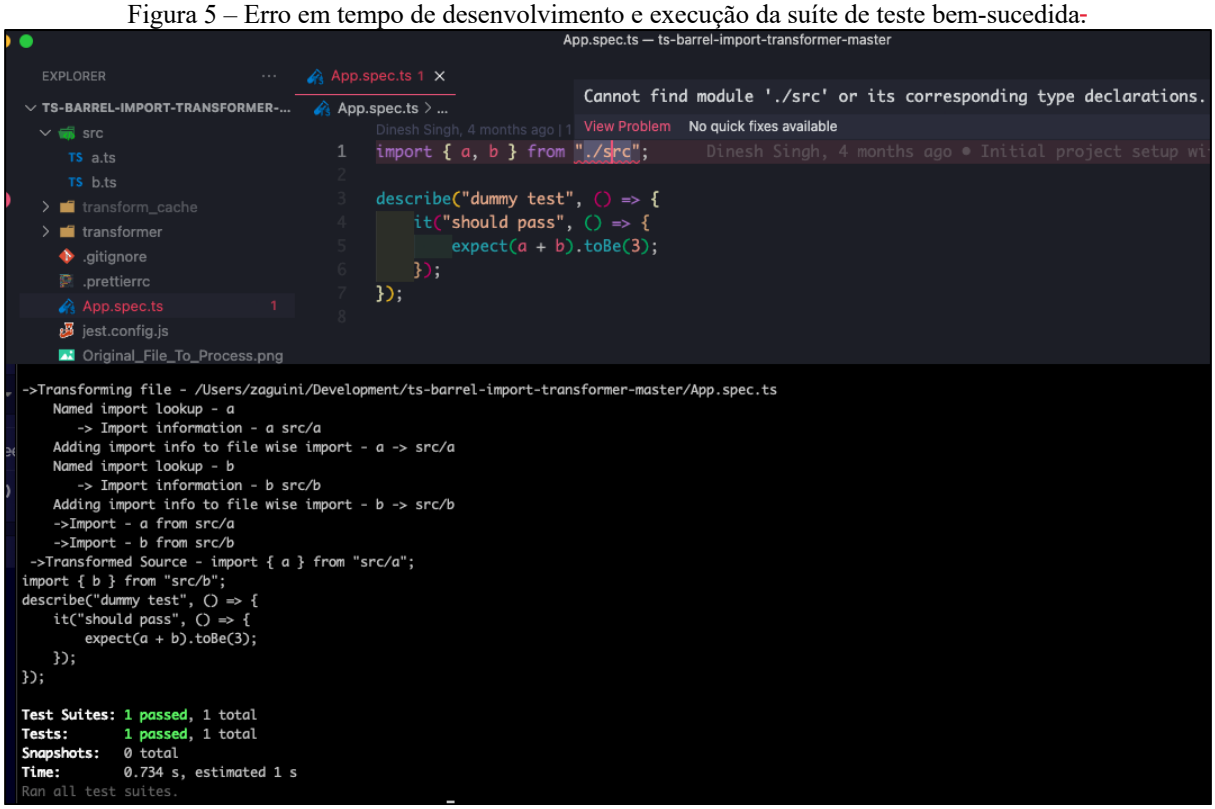
A principal diferença em relação ao projeto proposto é que essa ferramenta não emite código-fonte, apenas transforma em tempo de execução o código que será interpretado. Outro ponto importante é que ela está limitada

[1] Montagem a partir de imagens coletadas do documento de Singh (2022).

ao ambiente do Jest, não sendo possível utilizar o transformador enquanto desenvolve código-fonte a ser utilizado pelos usuário-final.

Ademais, um barril prévio é necessário, pois sua ausência causa erros em tempo de desenvolvimento devido a falta de um transformador adicional para tratar as transformações de código-fonte. No exemplo dado pelo criador do `ts-barrel-import-transformer` (Figura 4, lado esquerdo), ao ser removido o arquivo `./src`, apesar da IDE relatar erros, a execução ainda é bem sucedida. A Figura 5 ilustra o erro obtido em tempo de desenvolvimento (acima), e o resultado da execução (abaixo).

Figura 5 – Erro em tempo de desenvolvimento e execução da suíte de teste bem-sucedida.



Fonte: elaborado pelo autor.

Do ponto de vista de funcionalidades, essa ferramenta não lida com exportações padrão e não lida com arquivos JSON, o que de certa forma limita sua usabilidade. Arquivos JSON podem ser especialmente úteis nesse contexto, servindo como massa de teste e/ou especificações de entrada e saída.

3 PROPOSTA

Nesta sessão será apresentada a relevância deste trabalho. Além disso, serão exibidos os principais Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF), a metodologia a ser utilizada e o cronograma a ser seguido no decorrer do trabalho.

3.1 JUSTIFICATIVA

O Quadro 1 apresenta um comparativo das características mais notáveis entre os trabalhos correlatos. Nas linhas são descritas as características e nas colunas os trabalhos.

Quadro 1 - Comparativo dos trabalhos correlatos.

Trabalhos Correlatos / Características	Barrelsby – Coveney (2017)	Barrelbot – Wang (2017)	Ts-Barrel-Import-Transformer – Singh (2022)
Ferramenta trabalha junto ao compilador	Não	Não	Sim (Jest)
Gera arquivo auxiliar (barril)	Sim	Sim	Não
Reexporta exportações padrão	Sim	Não	Não
Preserva identificador de exportações padrão	Não	Não	Não
Reexporta exportações nomeadas	Sim	Sim	Sim
Reexporta arquivos JSON de maneira correta	Não	Não	Não
Cobre o projeto TypeScript por completo	Sim	Sim	Não
Necessita do arquivo adicional para gerar diagnósticos corretos no ambiente de desenvolvimento ao utilizar o barril	Sim	Sim	Sim

Fonte: elaborado pelo autor.

Conforme constatado no Quadro 1, a ferramenta `ts-barrel-import-transformer` é a única que trabalha junto ao compilador, mesmo que limitada ao transformador de arquivos de teste do Jest, não necessitando de execução anterior ou posterior para alcançar o objetivo desejado. Todas as ferramentas, com exceção da citada, geram arquivos adicionais que, como discutido previamente, são irrelevantes de um ponto de vista semântico.

Um dos três trabalhos correlatos reexportam as exportações padrão, mas nenhum deles preserva o identificador do símbolo definido pelo desenvolvedor.

Dos três trabalhos correlatos analisados, nenhuma ferramenta reexporta corretamente arquivos JSON.

A partir do comparativo das características, pode-se concluir que nenhuma das ferramentas avaliadas atende todas as características listadas. O trabalho proposto neste projeto entrega valor pois trabalha junto ao compilador, suportando o projeto TypeScript em sua totalidade. Além disso, ele não gerará o arquivo auxiliar, ao mesmo tempo em que entregará informações corretas ao ambiente de desenvolvimento. Ele reexportará exportações padrões preservando o identificador definido, assim como reexportará exportações nomeadas e arquivos JSON. Os trabalhos correlatos citados serviram como tomada de decisão no que toca às funcionalidades mínimas esperadas pelo projeto proposto.

Diante deste contexto, o presente trabalho apresenta uma contribuição prática/social ao ecossistema TypeScript pois irá proporcionar aos desenvolvedores uma melhor experiência de desenvolvimento, salvando tempo ao utilizar barris e excluindo a etapa de manutenção. A ferramenta proposta se conectará tanto ao compilador TypeScript quanto ao ambiente de desenvolvimento, sendo simples de utilizar e com instruções claras.

3.2 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos da ferramenta são:

a) transformar, em tempo de compilação, declarações de importações de barril em declarações de importação comum, resolvendo o caminho para o módulo para cada símbolo importado pelo barril (Requisito Funcional – RF);
b) emitir, em tempo de compilação, diagnósticos ~~corretos~~ sobre barris implícitos (RF);
c) emitir, em tempo de desenvolvimento, diagnósticos corretos sobre barris implícitos no ambiente de desenvolvimento (RF);
d) permitir ao usuário (desenvolvedor), enquanto no IDE, ir para a definição de um símbolo especificado na importação de barril em tempo de desenvolvimento (RF);
e) disponibilizar uma biblioteca base de modo que seja possível estender a funcionalidade proposta no trabalho (RF);
f) os itens 3.2.a a 3.2.d devem ser *plugins* que se conectarão ao compilador TypeScript (Requisito Não Funcional – RNF);
g) utilizar a linguagem de programação TypeScript para desenvolver os itens 3.2.a a 3.2.e (RNF);
h) armazenar o código-fonte dos *plugins* e da biblioteca base em um monorepositório (RNF);
i) expor as ferramentas propostas nos itens 3.2.a a 3.2.e como pacotes NPM a fim de disponibilizar as funcionalidades à comunidade de *software* livre (RNF);
j) os artefatos deverão ser executáveis JavaScript, a fim de serem executados em ambientes que não

estejam adaptados ao interpretador TypeScript (RNF);
k) a ferramenta poderá ser executada em qualquer sistema operacional desde que este tenha capacidade de executar comandos JavaScript através do interpretador NodeJS (RNF);
l) documentar as ferramentas descritas nos pontos 3.2.a a 3.2.e, de modo que sejam claras as instruções de utilização (RNF).

3.3 METODOLOGIA

O trabalho será desenvolvido observando as seguintes etapas:

a) levantamento bibliográfico: realizar levantamento bibliográfico sobre TypeScript, seu compilador e métodos de conexão para estender a linguagem, desenvolvimento de *plugins* e manipulação de código-fonte e diagnósticos;
b) elicitação dos requisitos: com base no levantamento bibliográfico e nos objetivos do trabalho, especificar requisitos funcionais e não funcionais da ferramenta proposta;
c) protótipo: elaborar projeto em menor escala baseado nos requisitos com objetivo de avaliar se o estado atual das ferramentas supre as necessidades do projeto;
d) preparação do ambiente de desenvolvimento: criar um monorepositório com a ferramenta Turborepo que possibilitará a manutenção de toda a base de código em apenas um lugar;
e) especificação dos testes de requisitos: elaborar suítes de teste utilizando a ferramenta Jest com o objetivo de verificar se as entradas e saídas estão de acordo com os diferentes cenários propostos pela ferramenta;
f) desenvolvimento do *plugin* transformador de código-fonte: com base nas especificações, implementar em TypeScript o *plugin* que irá transformar o código-fonte em tempo de compilação com base nas ferramentas expostas pelo compilador TypeScript;
g) desenvolvimento do *plugin* corretor de diagnósticos de compilador: com base nas transformações aplicadas pelo *plugin* transformador de código, implementar um *plugin* que adapte os diagnósticos gerados pelo compilador e inclua novas entradas em caso de utilização incorreta de barris implícitos;
h) desenvolvimento do *plugin* para o Servidor de Linguagem: com base nas especificações, implementar um *plugin* que estenderá o Servidor de Linguagem, reconhecendo os barris implícitos como código TypeScript legítimo e melhorando a experiência de desenvolvedor na IDE;
i) abstração de código e remoção de redundância: extrair como uma biblioteca funcionalidades similares a serem utilizadas pelas ferramentas especificadas nos itens f, g e h;
j) elaboração de documentação: a nível de projeto, documentar motivação, casos de uso, e instruções de utilização;
k) disponibilização na comunidade de *software* livre: disponibilizar código-fonte de maneira pública no Github, sendo possível a colaboração e manutenção pela comunidade.

As etapas serão realizadas nos períodos relacionados no Quadro 2.

Quadro 2 - Cronograma

	2023									
	fev.		mar.		abr.		mai.		jun.	
etapas / quinzenas	1	2	1	2	1	2	1	2	1	2
levantamento bibliográfico										
elicitação dos requisitos										
protótipo										
preparação do ambiente de desenvolvimento										
especificação dos testes de requisitos										
desenvolvimento do *plugin* transformador de código-fonte										
desenvolvimento do *plugin* corretor de diagnósticos de compilador										
desenvolvimento do *plugin* para o Servidor de Linguagem										
abstração de código e remoção de redundância										
elaboração de documentação										
disponibilização na comunidade de software livre										

Fonte: elaborado pelo autor.

4 REVISÃO BIBLIOGRÁFICA

Esta seção descreve brevemente sobre os assuntos que fundamentarão o estudo a ser realizado: TypeScript, árvores abstratas de sintaxe (ASTs), declarações de importação, *plugin*, ambiente de desenvolvimento integrado (IDE) e protocolo de Servidor de Linguagem.

O TypeScript é uma linguagem de programação de propósito geral desenvolvida pela Microsoft e mantida pela comunidade de *software* livre. Ela funciona como uma extensão ao JavaScript, tornando-o uma linguagem

tipificada. Ela adiciona funcionalidades relativas a esses tipos, tornando a experiência de desenvolvedor mais amigável e possibilitando detectar erros de sintaxe antes mesmo de executar o código. O TypeScript inclui um compilador que remove os tipos (que são necessários apenas em tempo de desenvolvimento), transformando o código-fonte em JavaScript compatível com o navegador ou com interpretadores, como o NodeJS (TYPESCRIPT, 2022).

Arquivos de código-fonte são carregados pelo compilador para serem transformados, frequentemente, em linguagem de máquina ou alguma outra linguagem. Em geral, os compiladores carregam o texto do arquivo de código-fonte e geram uma estrutura de dados conhecida como árvore abstrata de sintaxe (AST). Essa estrutura de dados permite a travessia do código-fonte de maneira mais eficaz (AHO *et al.*, 2013).

Uma das maneiras possíveis de transformar código-fonte é através de um padrão de projeto chamado visitador (VISITOR PATTERN, 2016). O visitador, no contexto do compilador TypeScript, é uma função que recebe um nó da AST e o transforma, retornando zero, um ou mais nós, substituindo o nó original na árvore. Cada tipo de nó, especificado pelo compilador, pode ter um visitador (BASARAT, 2020). Um dos tipos de nó explorados pelo compilador TypeScript é o de declaração de importação (BASARAT, 2020). Ela é formada por dois elementos principais: o caminho do módulo a ser importado, e os elementos a serem importados (MOZILLA, 2022). Uma vez incluídos, os elementos podem ser utilizados pelo desenvolvedor como desejado.

De acordo com Sterne (2019, tradução nossa), "um *plugin* [...] é um software de computador que adiciona novas funcionalidades a um programa base sem alterá-lo". No contexto deste trabalho, o compilador TypeScript permite a incorporação de *plugins* para transformar o código-fonte gerado e para modificar os diagnósticos emitidos pelo compilador (DOUGALL, 2019).

O Servidor de Linguagem é um protocolo que intermedia a comunicação entre o compilador da linguagem de programação e o ambiente de desenvolvimento. Ele é responsável por prover informações ricas para o IDE possibilitando funcionalidades como autocompletar, fatoração de código e visita à definição e/ou referências (MICROSOFT, 2021).

REFERÊNCIAS

AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compilers: Principles, Techniques, and Tools**. Pearson, 2013.

BASARAT, Ali S. Barrel. In: **TypeScript Deep Dive**, 2017. Disponível em: https://basarat.gitbook.io/typescript/main-1/barrel. Acesso em: 22 set. 2022.

BASARAT, Ali S. TIP: SyntaxKind enum. In: **TypeScript Deep Dive**, 2020. Disponível em: https://basarat.gitbook.io/typescript/overview/ast/ast-tip-children. Acesso em: 22 set. 2022.

BASARAT, Ali S. TIP: Visit Children. In: **TypeScript Deep Dive**, 2020. Disponível em: https://basarat.gitbook.io/typescript/overview/ast/ast-tip-children. Acesso em: 22 set. 2022.

COVENEY, Ben. **bencoveney/barrelsby: Automatic TypeScript barrels (index.ts files) for your entire code base**, 2017. Disponível em: https://github.com/bencoveney/barrelsby. Acesso em: 22 set. 2022.

DOUGALL, Michael. **madou/typescript-transformer-handbook: A comprehensive handbook on how to create transformers for TypeScript with code examples**, 2019. Disponível em: https://github.com/madou/typescript-transformer-handbook. Acesso em: 22 set. 2022.

FREEMAN, Adam. Understanding TypeScript. In: **Essential TypeScript 4**. Berkeley, CA: Apress, 2021.

MICROSOFT. **Official page for Language Server Protocol**, 2021. Página inicial. Disponível em: https://microsoft.github.io/language-server-protocol. Acesso em: 22 set. 2022.

MOZILLA. **Import – Javascript | MDN**, 2022. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/import. Acesso em: 22 set. 2022.

SINGH, Dinesh. **dsmalik/ts-barrel-import-transformer: Typescript - Sample Barrel Import Transformer**, 2022. Disponível em: https://github.com/dsmalik/ts-barrel-import-transformer. Acesso em: 22 set. 2022.

STACK OVERFLOW. **Stack Overflow Developer Survey 2021**, 2021. Página inicial. Disponível em: https://insights.stackoverflow.com/survey/2021#technology-most-loved-dreaded-and-wanted. Acesso em: 20 set. 2022.

STERNE, Jonathan. Plug-in. In: Encyclopedia Britannica, 2019. Disponível em: https://www.britannica.com/technology/plug-in. Acesso em: 22 set. 2022.

TYPESCRIPT. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2022. Disponível em: https://pt.wikipedia.org/wiki/TypeScript. Acesso em: 22 set. 2022.

VISITOR PATTERN. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2016. Disponível em: https://en.wikipedia.org/wiki/Visitor_pattern. Acesso em: 22 set. 2022.

WANG, Shawn. **stolinski/barrelbot: The Automated barrel file manager: Recursively watches a folder and generates barrel files for your JS/TS project**, 2017. Disponível em: https://github.com/stolinski/barrelbot. Acesso em: 22 set. 2022.

FORMULÁRIO DE AVALIAÇÃO BCC – PROFESSOR AVALIADOR – PRÉ-PROJETO

Avaliador(a): Gilvan Justino

Atenção: quando o avaliador marcar algum item como atende parcialmente ou não atende, deve obrigatoriamente indicar os motivos no texto, para que o aluno saiba o porquê da avaliação.

	ASPECTOS AVALIADOS	Atende	atende parcialmente	não atende
ASPECTOS TÉCNICOS	1. INTRODUÇÃO O tema de pesquisa está devidamente contextualizado/delimitado?	X		
	O problema está claramente formulado?	X		
	2. OBJETIVOS O objetivo principal está claramente definido e é passível de ser alcançado?		X	
	Os objetivos específicos são coerentes com o objetivo principal?	X		
	3. TRABALHOS CORRELATOS São apresentados trabalhos correlatos, bem como descritas as principais funcionalidades e os pontos fortes e fracos?	X		
	4. JUSTIFICATIVA Foi apresentado e discutido um quadro relacionando os trabalhos correlatos e suas principais funcionalidades com a proposta apresentada?	X		
	São apresentados argumentos científicos, técnicos ou metodológicos que justificam a proposta?	X		
	São apresentadas as contribuições teóricas, práticas ou sociais que justificam a proposta?	X		
	5. REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO Os requisitos funcionais e não funcionais foram claramente descritos?	X		
	6. METODOLOGIA Foram relacionadas todas as etapas necessárias para o desenvolvimento do TCC?	X		
	Os métodos, recursos e o cronograma estão devidamente apresentados e são compatíveis com a metodologia proposta?	X		
	7. REVISÃO BIBLIOGRÁFICA Os assuntos apresentados são suficientes e têm relação com o tema do TCC?	X		
	As referências contemplam adequadamente os assuntos abordados (são indicadas obras atualizadas e as mais importantes da área)?	X		
ASPECTOS METODOLÓGICOS	8. LINGUAGEM USADA (redação) O texto completo é coerente e redigido corretamente em língua portuguesa, usando linguagem formal/científica?	X		
	A exposição do assunto é ordenada (as ideias estão bem encadeadas e a linguagem utilizada é clara)?	X		