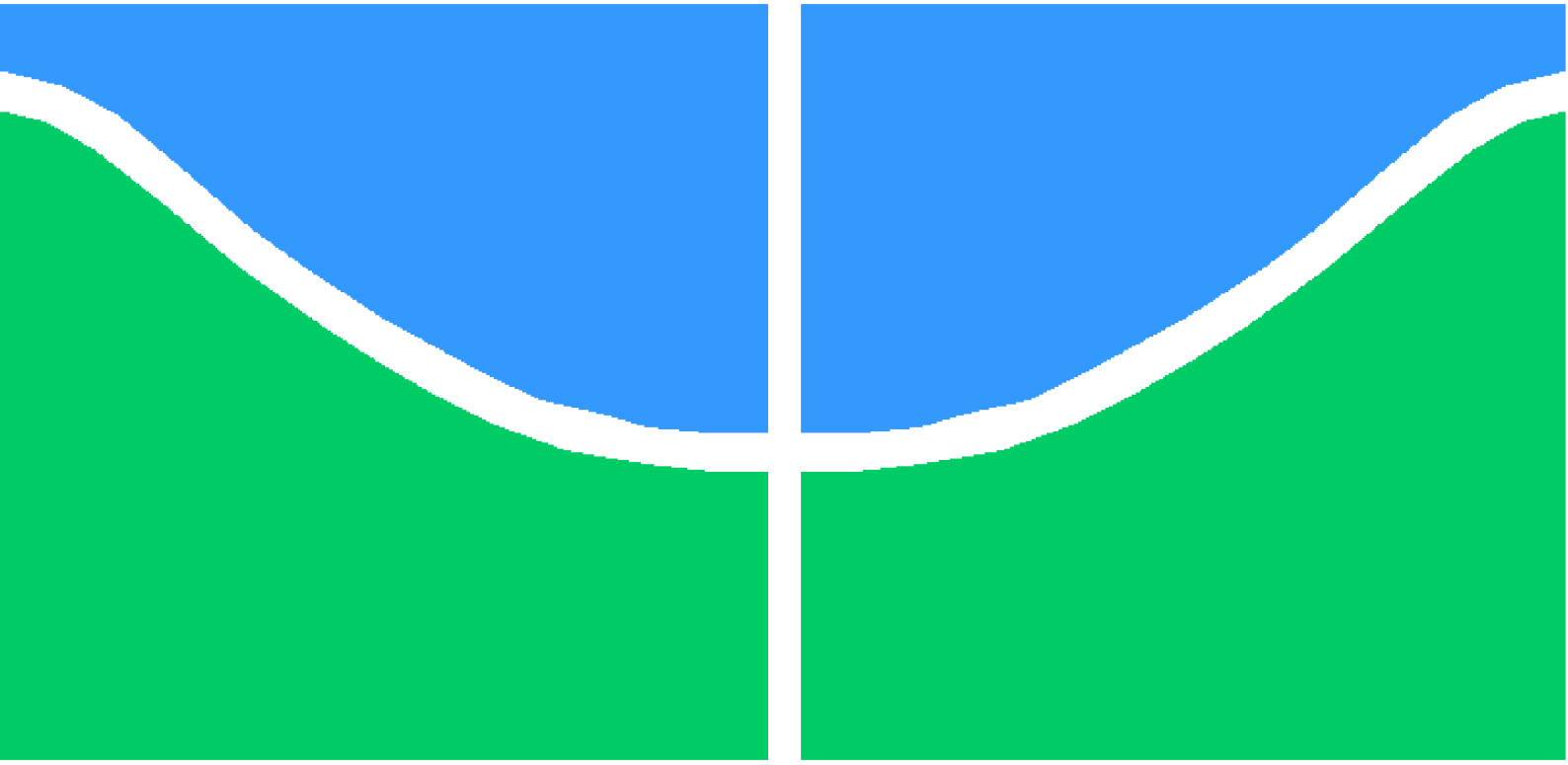

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

cp-tools: evolução de um formatador de problemas de competição para comunicação com a plataforma Polygon

Autor: Vitor Falcão Habibe Costa

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2021

Vitor Falcão Habibe Costa

cp-tools: evolução de um formatador de problemas de competição para comunicação com a plataforma Polygon

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2021

Vitor Falcão Habibe Costa

cp-tools: evolução de um formatador de problemas de competição para comunicação com a plataforma Polygon

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, :

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. John Lenon Cardoso Gardenghi
Convidado 1

Prof. Dr. Daniel Saad Nogueira Nunes
Convidado 2

Brasília, DF
2021

"Quando você acha que as pessoas morrem?
Quando elas levam um tiro no coração pela bala de uma pistola? Não.
Quando elas são devastadas por uma doença incurável? Não.
Quando elas tomam uma sopa feita de cogumelos venenosos? Não!
É quando... elas são esquecidas.
(Dr. Hiluluk, One Piece)

Resumo

Este trabalho teve como objetivo a implementação da comunicação entre a ferramenta formatadora de problemas `cp-tools` e a plataforma Polygon. Esta implementação também incluiu o desenvolvimento de algoritmos para tratamento de conflitos, diversas evoluções significativas na arquitetura e na qualidade do código, e padronizações em uma interface comum para se comunicar com diversas plataformas. O processo de desenvolvimento adotou metodologias da Engenharia de *Software* como o *Domain Driven Design* e a Metodologia Ágil. Além disso boas práticas como o uso de testes automatizados e manutenções corretivas constantes foram aplicados com êxito. Os objetivos foram atingidos parcialmente a partir da implementação da parte mais complexa da comunicação com o Polygon. Diversas evoluções futuras são explicitadas ao fim deste trabalho.

Palavras-chaves: competições de programação. formatação de problemas.

Abstract

This undergraduate thesis aimed at developing the communication between the tool `cp-tools` and the Polygon platform. This implementation also included the development of algorithms for dealing with conflicts, several evolutions on tool's architecture and on code's quality, and deciding default behaviours for a common interface so the tool can communicate with various external platforms easily. The development process adopted Software Engineering based metodologies, as Domain Driven Design and Agile. Some good practices as using automated testing and applying constant corrective maintenance were used succesfully. The objectives were partially achieved with the implementation of the most complex part of the communication with Polygon. A variety of future evolutions and improvements are shown by the end of this thesis.

Key-words: programming contests. problems builder.

Lista de ilustrações

Lista de abreviaturas e siglas

API	*Application Programming Interface*
ICPC	*International Collegiate Programming Contest*
HTML	*HyperText Markup Language*
PDF	*Portable Document Format*
HTTPS	*Hypertext Transfer Protocol Secure*
JSON	*JavaScript Object Notation*
DDD	*Domain Driven Design*
TDD	*Test-Driven Development*
BDD	*Behaviour-Driven Development*

Sumário

Introdução

A programação competitiva é um esporte mental no qual os competidores recebem um conjunto de problemas e devem implementar códigos-fonte que os resolvam. Os problemas são divididos em diversas categorias como programação dinâmica, matemática, grafos, estrutura de dados, etc. A equipe ou o indivíduo vencedor é aquele que conseguir resolver mais problemas em menos tempo (HALIM et al., 2013).

Essas competições acontecem tanto de maneira presencial quanto a distância. Em ambos os casos, é comum utilizar plataformas que hospedem as competições. Elas são responsáveis pelo controle do tempo, número de acertos, número de erros, conferir se a resolução é válida, placares, e outras funcionalidades de gerência da competição (WASIK et al., 2018).

Diversas empresas do ramo de tecnologia usam problemas de competições de programação como uma das maneiras de avaliar as habilidades do candidato. Além disso, muitas também possuem suas próprias competições nas quais oferecem prêmios, podendo ser desde dinheiro a vagas de trabalho na empresa. Como exemplo, podemos citar a Google, que possui o Google Code Jam[1], ou o Facebook, que possui o Facebook Hacker Cup[2].

A melhor forma de aprender e se preparar para competições de programação é participando delas. Plataformas como o Codeforces[3], URI[4], LeetCode[5], e HackerRank[6] são conhecidas por oferecer inúmeros problemas e competições para treino. Essas são classificadas como juízes online por terem gerenciamento de usuários, base de problemas, competições, editoriais, e outras funcionalidades. Juízes eletrônicos contemplam apenas a etapa de correção automatizada, e usualmente juízes online fazem uso de juízes eletrônicos.

Nem todas as plataformas oferecem a opção de ter sua própria competição personalizada: os juízes online Codeforces e vJudge[7], e o juíz eletrônico BOCA[8] são exemplos de plataformas que tem essa funcionalidade. A possibilidade de criação de competições é uma vantagem ao lecionar programação competitiva, pois permite focar em uma necessidade ou em um assunto específico, por exemplo um assunto específico como grafos ou programação dinâmica.

[1] https://codingcompetitions.withgoogle.com/codejam
[2] https://facebook.com/codingcompetitions/hacker-cup
[3] https://codeforces.com/
[4] https://www.urionlinejudge.com.br/judge/en/login
[5] https://leetcode.com/
[6] https://www.hackerrank.com/
[7] https://vjudge.net/
[8] https://github.com/cassiopc/boca

Motivação

A comunidade de programação competitiva e seus colaboradores tem crescido no decorrer do tempo. Novos programadores se tornam competidores, assim como programadores já experientes se tornam autores de novos problemas. Os problemas de diversos autores são combinados para produzir novas competições em plataformas como o Codeforces, que oferece competições regularmente.

Diversas plataformas de competição permitem que um usuário insira seus próprios problemas e crie suas próprias competições, porém cada uma delas tem uma interface diferente. Essa desigualdade faz com que o autor de um problema tenha que adaptar seus arquivos, escolhendo uma plataforma específica ou preparando múltiplas versões dos arquivos de modo que o problema possa ser usado em diversas plataformas.

O reaproveitamento de questões feitas para uma plataforma se torna mais difícil e demorado se a plataforma alvo não for a mesma que a qual o problema foi originalmente construído. A criação de novas questões se tornam mais lenta quando o objetivo é que ela seja usada em múltiplas plataformas. Sempre que a intenção for utilizar uma nova plataforma, é preciso aprender sobre a interface dela. Esses problemas afastam novos contribuidores e reduz a produtividade dos já existentes.

A ferramenta, cuja evolução e manutenção é resultado deste trabalho, é o `cp-tools`. Esse software possui o objetivo de criar uma interface comum entre diversos formatadores de problemas já existentes e automatizar toda essa comunicação entre a ferramenta e as diversas plataformas.

A criação da interface comum apresentada nesta ferramenta atende a esses problemas. Ela abstrai as diversas interfaces das plataformas e adapta elas a sua própria. À vista disso, o usuário só precisa aprender uma interface e como desenvolver com ela, e o problema de adaptar os arquivos às numerosas plataformas é solucionado, provendo ao usuário a oportunidade de se imergir apenas na qualidade do problema que está criando e reduzir o tempo de formatação e elaboração do problema.

Objetivos

Este trabalho tem como objetivo evoluir e manter o cp-tools, ferramenta que facilita o trabalho dos autores de problemas, reduzindo a curva de aprendizado necessária e o tempo de produção do problema. A evolução terá como foco a integração com o Polygon e suas automatizações, como enviar e receber os arquivos do problema por meio de uma interface de linha de comando.

Os seus objetivos específicos são:

- evoluir a interface em linha de comando para dar suporte a novos comandos;
- implementar a comunicação entre o `cp-tools` e o Polygon a partir dos comandos *push* e *pull*;
- evoluir e fazer a manutenção da interface única de criação de problemas utilizada pelo `cp-tools`;
- identificar e resolver conflitos entre os repositórios de problemas do Polygon e o local do `cp-tools`.

Estrutura do trabalho

Este trabalho é composto de quatro capítulos. A Fundamentação Teórica traz os conceitos necessários para a compreensão do trabalho. A Metodologia explica como foi desenvolvido, quais ferramentas foram utilizadas, as etapas e o fluxo do projeto. Os Resultados trazem o que foi feito, os problemas encontrados, como foram superados e as melhorias obtidas. Por fim as Considerações Finais traz um desfecho ao trabalho e evoluções possíveis.

1 Fundamentação Teórica

Neste capítulo é inicialmente apresentado a programação competitiva, uma das bases deste trabalho. Em seguida explica-se o Polygon, plataforma formatadora de problemas, e sua API. Na próxima seção apresenta-se a ferramenta cp-tools, cuja evolução e manutenção é o objetivo deste trabalho. O capítulo finaliza com a comparação da ferramenta entre o `cp-tools` e outros formatadores de problema.

1.1 Programação Competitiva

A programação competitiva espera que o competidor desenvolva o código o mais rápido possível para solucionar um dado problema. Alguns livros, como o *Competitive Programmer's Handbook* (LAAKSONEN, 2018) e o *Competitive programming 3* (HALIM et al., 2013), tem o objetivo de ensinar sobre essa modalidade de competição. Além disso, ambos também focam em explicar algoritmos e soluções de problemas de competições anteriores ou já conhecidos pela Ciência da Computação.

De acordo com Halim et al. (2013), a programação competitiva pode ser definida com a seguinte frase: "Dado problemas bem conhecidos da Ciência da Computação, resolva-nos o mais rápido possível". Essa frase significa que vamos lidar com problemas cuja solução já é conhecida pela Ciência da Computação, e conhecê-la ou descobri-la durante a competição é necessário para resolver o problema. Evidentemente, a segunda parte desta afirmação estabelece que o competidor mais rápido terá vantagem na pontuação sobre os outros.

As competições de programação são reconhecidas mundialmente. Um exemplo de competição internacional é o ICPC[1] (*International Collegiate Programming Contest*), que anualmente reúnem times que representam suas universidades do mundo inteiro. Empresas de grande porte, como a Google e o Facebook, tem suas próprias competições, a Google Code Jam[2] e a Facebook Hacker Cup[3] respectivamente, cujas premiações consistem em dinheiros ou oportunidades de emprego.

Um dos atributos principais das competições de programação é o uso de juízes online. Eles são ferramentas de automatização encarregadas de grande parte do gerenciamento das competições. Eles possuem a função de manter o placar, receber e avaliar códigos dos competidores, enviar o resultado da avaliação o mais rápido possível, disponibilizar a descrição dos problemas e gerenciar o tempo das competições. Em Wasik et al.

[1] https://icpc.global/
[2] https://codingcompetitions.withgoogle.com/codejam
[3] https://www.facebook.com/codingcompetitions/hacker-cup

(2018), diversas dessas ferramentas são pesquisadas e expostas. Uma delas é o Codeforces, que oferece competições frequentemente, tem classificações de níveis para os competidores, permite que um usuário crie sua própria competição, entre outras funcionalidades.

De acordo com Ribeiro e Guerreiro (2008), a entrada no mundo da programação competitiva é difícil para novatos que ainda não tem conhecimento suficiente de programação. O autor afirma que o uso de juízes online, linguagem funcional e problemas de competição aumentaram a produtividade e engajamento dos alunos. A causa da produtividade foi associada ao resultado instantâneo vindo dos juízes online. Trabalhos como Gonzalez-Escribano et al. (2019), Coore e Fokum (2019) e Lawrence (2004) também utilizam as competições de programação como base, usando elas para aumentar a colaboração entre competidores ou aumentar o aprendizado dos alunos.

Apesar do uso de juízes online ter demonstrado melhorias em sistemas de aprendizado e competição, Wasik et al. (2018) reforça a dificuldade de implementar um juiz confiável e estável. Ele ressalta os problemas de segurança e confiabilidade, providos pela execução de código possivelmente malicioso e os picos de requisições geradas principalmente nos momentos finais das competições. Ele também enfatiza a importância de bons casos de teste, por exemplo, soluções incorretas podem ser consideradas corretas pelo juiz eletrônico caso o conjunto de testes não cubra todos os *corner cases* possíveis.

Para manter as competições interessantes e desafiadoras é necessário a utilização de problemas inéditos. A reutilização de problemas em competições, além de dar uma vantagem para o usuário que já os solucionou, abre portas para a reutilização de códigos já implementados anteriormente. Isso viola diretamente a parte das competições em que o tempo de resolução é um dos critérios de pontuação. Para evitar isso, são criados novos problemas para cada nova competição.

A criação de problemas inéditos pode ser uma tarefa árdua e repetitiva por causa da portabilidade para diversas plataformas. Para contornar estas dificuldades foram desenvolvidos os formatadores de problemas. Esses são ferramentas com o objetivo de auxiliar na formação de novos problemas. Alguns juízes online dispõem de seus próprios formatadores de problemas, que no caso do Codeforces é o Polygon[4], que oferece automatizações para diversas partes do processo, desde a criação da descrição do problema até a validação dos casos de teste. Além de fornecer o formatador de problema, o Polygon também oferece uma API com diversos *endpoints* para que possamos fazer nossas próprias automatizações.

1.2 Polygon

O Polygon é uma ferramenta de formatação de problema disponível a partir de uma interface web. Nele é possível a criação de problemas e competições que podem ser

[4] https://polygon.codeforces.com/

disponibilizados no Codeforces. Ele também é a única maneira de inserir seus próprios problemas na plataforma, logo é necessário utilizá-lo se quiser ter seus problemas ou competições personalizadas. A Figura 1 apresenta a tela inicial do Polygon.

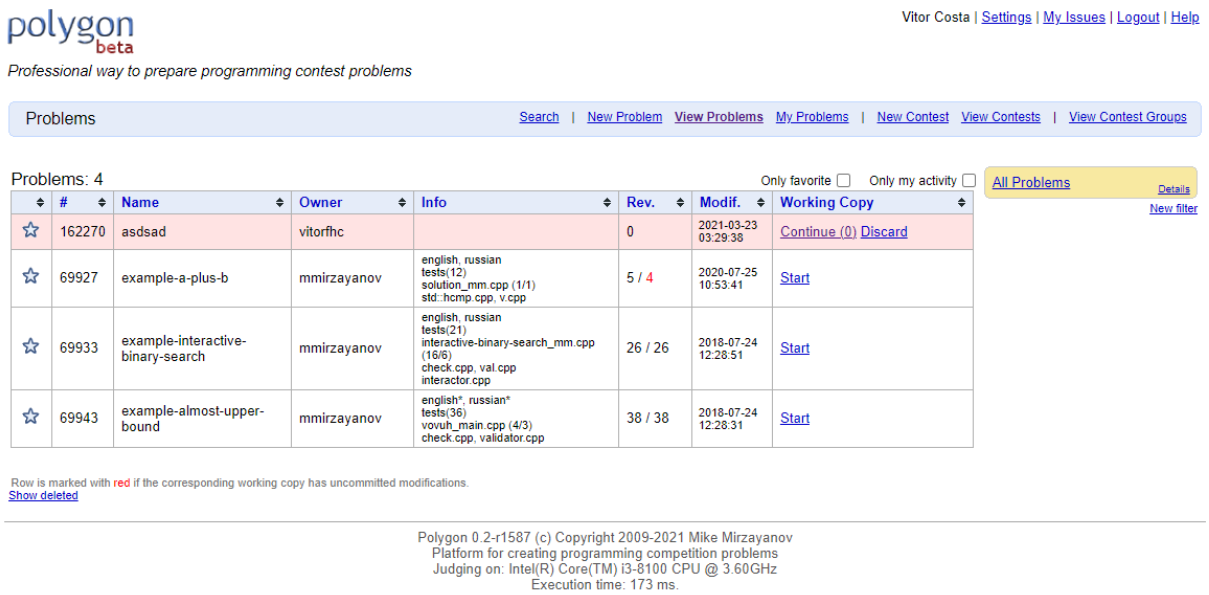

Figura 1 – Captura de tela da interface web inicial do Polygon

Na Figura 2 podemos ver uma visão geral da ferramenta. Um usuário comum possui os problemas, que por sua vez são compostos destas cinco partes: descrição (*statement*), soluções (*solutions*), validador (*validator*), verificador (*checker*) e os testes (*tests*). Um conjunto de problemas podem formar uma competição, e também é possível reutilizar o mesmo problema em diversas competições.

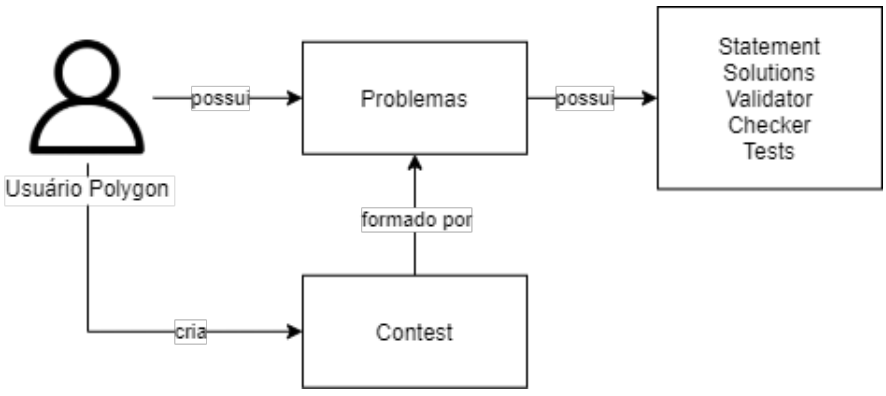

Figura 2 – Representação superficial do Polygon

1.2.1 Componentes do Problema

Todo problema possui um nome único para o criador e suas informações gerais (*general info*). Enquanto o conjunto do nome e autor funcionam como um identificador

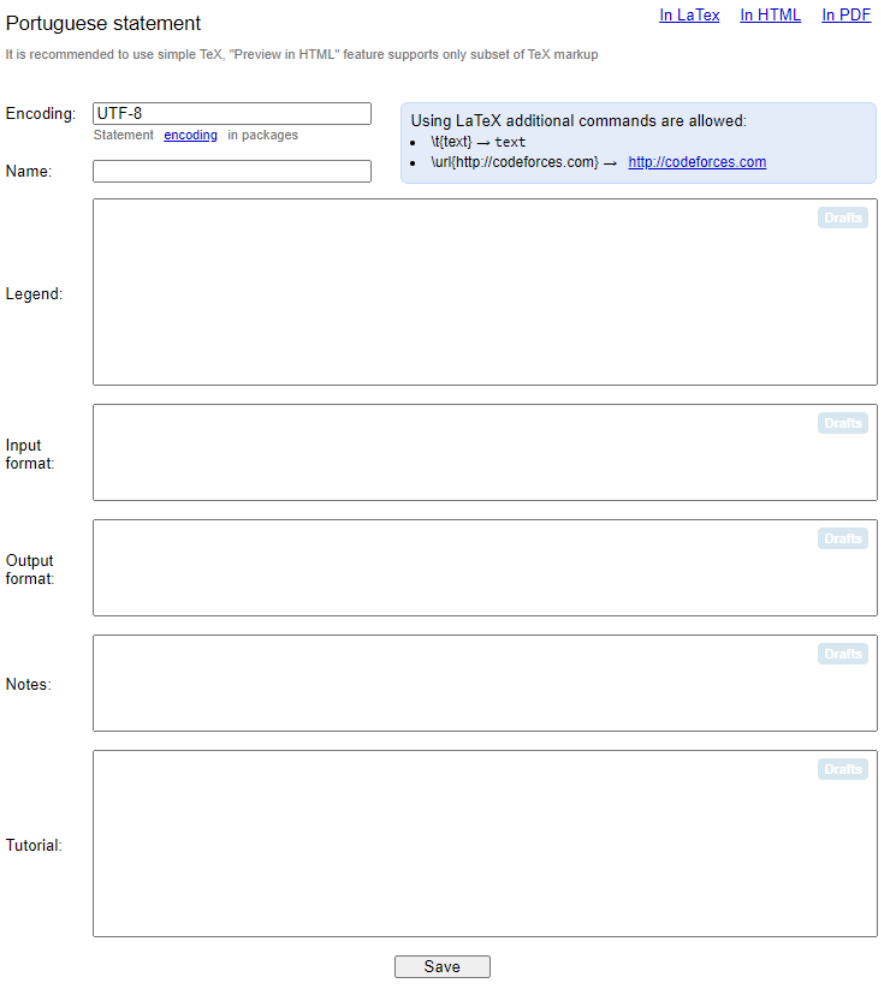

Figura 3 – Captura de tela da interface web do Polygon durante a criação da descrição do problema

do problema, a seção de informações gerais apresenta restrições e especificações. Nela existe qual será o arquivo de entrada e saída (*input and output file*), o tempo e a memória limite ao qual as soluções devem se restringir.

Na descrição colocamos o problema a ser solucionado e as restrições as quais o usuário deve estar atento. No Polygon, ao criá-la, ele nos provê os seguintes campos: codificação de caracteres, nome, legenda, formato da entrada, formato da saída, notas e o tutorial. Cada um dos campos com uma função específica, e além disso possui suporte para adicionar imagens. Ao terminar é possível exportá-la em três formatos: HTML, LaTeX e PDF. A Figura 3 mostra essa interface.

Um problema pode ter um ou mais conjuntos de teste. Cada conjunto é composto de diversos arquivos que são numerados a partir do número um. Existem dois tipos de teste: manuais e *scripts*. Os testes manuais são arquivos contendo o teste digitado manualmente, enquanto os do tipo *script* são comandos que geram casos de teste.

Um teste do tipo *script* poderia ser: `aleatorio 0 100`. Esse *script* possui uma

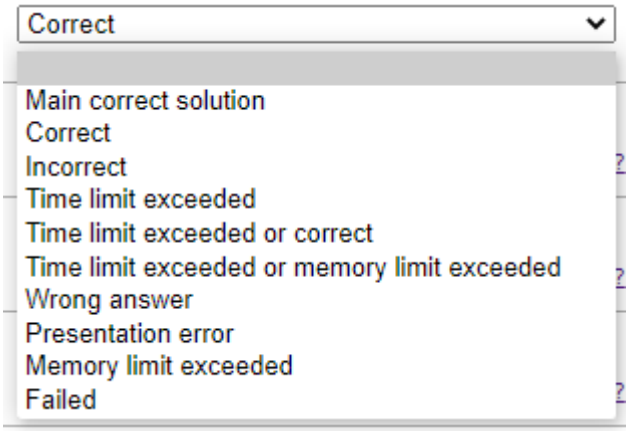

Figura 4 – Captura de tela da seleção do tipo de uma solução no Polygon

dependência que é o executável `aleatorio`, responsável por gerar um número aleatório entre seu primeiro e segundo parâmetro. Para que o teste funcione é preciso existir esse executável dentro dos arquivos do problema.

As soluções são arquivos contendo códigos que produziram executáveis, os quais receberam os testes como entrada e produzirão saídas que devem ser compatíveis com a classificação dada a cada solução. De acordo com a documentação[5] do Polygon, há oito tipos de soluções: correta principal, apenas correta, incorreta, fora do tempo limite, fora do limite de memória, falha, com resposta errada ou com erro de apresentação.

O arquivo de solução marcado como solução correta principal (*main correct solution*) é utilizado para gerar as respostas usadas pelo juiz online e só é permitido existir uma solução deste tipo. Para as demais soluções o Polygon procede da seguinte forma: se um arquivo de tipo resposta incorreta (*wrong answer*) retornar algo diferente de resposta incorreta o formatador do problema é notificado. O mesmo acontece para todos os outros tipos, por exemplo, quando marcado com o tipo solução correta (*correct solution*) o usuário é notificado caso o retorno não seja solução correta.

A documentação oficial do Polygon diverge do que é visto na sua interface web. Ao acessar pela interface, mostrado na Figura 4, podemos ver dois tipos de soluções que não existem na documentação, são elas: tempo limite excedido ou correta, e tempo limite excedido ou memória limite excedida. Elas representam interseções de dois tipos.

Como mencionado na publicação oficial[6] sobre validadores no blog do Codeforces, é importante ter cuidado com o conjunto de testes, um assunto também abordado por Wasik et al. (2018). Por exemplo, um problema que diz que a entrada será um número inteiro positivo menor que cem deve garantir que isso seja verdade para todos seus testes. Para isso existem validadores, os quais validam todos os testes de todos os conjuntos. Os

[5] https://polygon.codeforces.com/static/polygon.rtf
[6] https://codeforces.com/blog/entry/18426

validadores podem ser simples, no sentido que apenas validam os intervalos e tipos das informações contidas nos testes, ou mais complexos, realizando tarefas como verificar a existência de uma conexão entre dois pontos em um grafo. Os validadores são códigos escritos pelo criador do problema e que usam os testes como arquivos de entrada.

Quando um teste pode ter mais de uma resposta correta utiliza-se idealmente os verificadores, apesar de também serem utilizados quando as respostas são únicas. Esses são programas que recebem a resposta dada pelo competidor como entrada e retornam um veredito, tal como resposta correta, resposta incorreta ou se algum erro acontecer durante a execução. É essencial mencionar que a equipe do Codeforces reforça[7] o uso da `Testlib`[8]. Ela é usada para facilitar e evitar problemas que podem ser causados ao escrever seu próprio verificador ou validador.

1.2.2 Arquivos e Sessões de Edição

A plataforma utiliza uma categorização dos arquivos utilizados no problema. A documentação apresenta os arquivos do tipo: recursos (*resources*), fonte (*source*) e adicionais (*additional*). Os arquivos de recurso são os arquivos de bibliotecas e cabeçalhos, porém também podem ser arquivos do tipo LaTeX. Esses arquivos serão movidos para o diretório de compilação durante o processo de construção de cada executável. Ao criar um teste to tipo *script* é necessário possuir um arquivo do tipo recurso com o mesmo nome do executável utilizado. Os arquivos fonte são os executáveis ou validadores e verificadores que podem ser tanto arquivos de código-fonte quanto executáveis, enquanto arquivos adicionais são aqueles que não se encaixam nas outras categorias. A documentação cita, como exemplo, arquivos do Microsoft Word para essa categoria. O objetivo de arquivos adicionais depende do usuário, que pode os utilizar para guardar informações, anotações, imagens e documentos do problema, por exemplo.

O Polygon possui sessões de edição (*edit sessions*) que são cópias das modificações feitas por um usuário e salvas no servidor deles. O usuário pode optar por fazer o *commit* de novas modificações ou descartá-las. Não existe um tratamento de conflitos, logo toda vez que mudanças são aceitas elas sobreescrevem os arquivos anteriores.

1.2.3 Pacotes

O Polygon oferece um empacotamento do problema. A plataforma cria um arquivo comprimido com todos os arquivos do problema dentro dele. Além disso, também insere alguns executáveis como o `doall.bat` que gera os casos de testes a partir dos geradores *script*. Para criar um pacote (*package*) é necessário não ter nenhuma sessão de edição em aberto.

[7] https://codeforces.com/blog/entry/18431
[8] https://github.com/MikeMirzayanov/testlib

De acordo com a documentação a intenção era prover mais formas de empacotar o problema além das existentes, porém isso nunca foi implementado. Hoje a plataforma oferece duas maneiras de empacotar: a padrão (*standard*) e a completa (*full*). A diferença entre as duas maneiras de empacotar não é explicada pela documentação.

1.2.4 API

Em 2016 foi publicada a documentação[9] da API do Polygon. Tudo que pode se fazer pela interface *web* é factível também pela API. Ela utiliza requisições HTTPS e as responde utilizando o formato JSON ou o texto do arquivo, dependendo do *endpoint* utilizado.

Como qualquer acesso a qualquer problema pela interface exige autenticação, a API age da mesma maneira. Os usuários podem não ter nenhuma autorização sobre o problema, não permitindo nem que ele o leia, ou podem ter a permissão de leitura. Para efetuar modificações é preciso a permissão de escritura que é sempre concedida ao dono do problema e ele pode adicionar mais escritores. A API exige que em todas requisições a qualquer *endpoint* sejam enviados parâmetros que autentiquem quem fez a requisição.

A documentação apresenta três parâmetros obrigatórios para serem incluídos como forma de autenticação: `apiKey`, `time` e `apiSig`. A `apiKey` é fornecida pelo Polygon junto com o `secret`, que será utilizado para gerar a `apiSig`. O parâmetro `time` exige a hora atual no formato Unix e não pode ter uma margem de erro maior do que cinco minutos. Como alguns *endpoints* exigem argumentos, tal como o identificador de um problema, esses são adicionados normalmente junto com os outros exigidos pela autenticação.

A maior complexidade da autenticação se apresenta na formação do parâmetro `apiSig`. Ele é formado pela junção de diversas partes e depois criptografado utilizando o algoritmo de *hash* do tipo SHA-512. A sua composição se dá da seguinte maneira:

```
<rand>/<methodName>?param1=value1...&paramN=valueN#<secret>,
```

onde `rand` é um número aleatório de seis dígitos criado pelo cliente da requisição, o `methodName` é o nome do método a ser utilizado, seguido pelos parâmetros ordenados lexicograficamente e por fim o `secret`.

Quando o resultado de uma requisição está no formato JSON, esse resultado é a representação de um dos objetos do Polygon. Existem nove tipos de objetos, como o que representa um problema (*Problem*), ou o que representa os dados de um arquivo (*File*). Todos são descritos na documentação e em caso de falha na requisição um campo de erro com a descrição da falha é criado no JSON.

[9] https://docs.google.com/document/d/1mb6CDWpbLQsi7F5UjAdwXdbCpyvSgWSXTJVHl52zZUQ

1.3 cp-tools

O cp-tools[10] é uma ferramenta de interface de linha de comandos, usada para formatação de problemas. Suas padronizações de arquivos e forma de funcionamento foram baseadas no Polygon, por isso apresentam diversas características em comum. Todavia, além de um formatador de problemas, o cp-tools tem como um dos seus objetivos a comunicação com diversas plataformas de formatação de problemas. A primeira integração começou a ser desenvolvida com este trabalho e tem como objetivo integrar com a plataforma externa Polygon. Ela é feita a partir dos comandos `polygon push` e `polygon pull`, que serão apresentados ao decorrer desta seção.

1.3.1 Comandos do cp-tools

No arquivo `README.md`[11] do repositório oficial existe um resumo dos comandos. Existem seis comandos, podendo cada um deles ter subcomandos, porém atualmente apenas o comando `polygon` os possui. Por ser uma interface de linha de comando também existem os argumentos para cada um dos comandos, esses tem a funcionalidade desde mostrar mensagens de ajuda quanto a utilização do comando até argumentos para uma personalização do funcionamento do mesmo.

O comando `init` é responsável pela criação de um diretório contendo os arquivos iniciais da formatação de um problema, tendo como exemplo o `config.json`, usado para armazenar configurações e informações daquele problema em específico. O comando também insere alguns arquivos de exemplo, como validadores, verificadores, testes e uma descrição. A criação desse diretório reforça a padronização de arquivos utilizadas pelo cp-tools.

Para gerar uma versão em PDF do problema existe o comando `genpdf`, o qual depende de diversos componentes como a descrição e os testes marcados para serem usados de exemplo. O seu equivalente para gerar um arquivo LaTeX é o comando `gentex`. Para gerar o arquivo em formato PDF, a ferramenta primeiro o gera em formato LaTeX e depois compila para PDF. Como esses comandos geram arquivos intermediários automaticamente, também há o comando `clean` para removê-los depois.

Para que o usuário tenha confiança no seu validador, no seu verificador ou nos testes ele deve utilizar o comando `check`. A escolha de qual recurso ele pretende validar depende do argumento usado. Por exemplo, a *flag* `-v` após o comando faz a verificação dos validadores. Essa verificação consiste em compilar o código fonte do validador para garantir que não há erro e executá-lo em testes pré-determinado para validar que o resultado seja o esperado. Ao testar o verificador, é esperado que ele não tenha erros de

[10] https://github.com/edsomjr/competitive-problems-tools
[11] https://github.com/edsomjr/competitive-problems-tools/blob/master/README.md

compilação e que os testes retornem os resultados esperados.

O comando `judge` é responsável por julgar as soluções. Ao utilizá-lo é necessário especificar qual o arquivo contendo o código da solução pretende-se julgar. A ferramenta compila e executa a solução utilizando os testes como arquivos de entrada e nesse caso é esperado sempre a resposta correta daquele teste. Esse comando se assemelha a um juiz eletrônico como os utilizados por juízes online como o Codeforces, porém não tão robusto quanto eles, já que carece de uma implementação mais complexa que isole o ambiente em que o código é executado.

Por fim, existe o comando `polygon`, cujo desenvolvimento é o propósito deste trabalho. Esse é responsável pela comunicação com a plataforma Polygon e isso é feito de três maneiras: sem nenhum subcomando, utilizando o subcomando `pull` ou o subcomando `push`. Quando nenhum subcomando é fornecido é feito um teste de conexão, e esse teste tem como principal objetivo garantir que o usuário tenha as permissões corretas para acessar o problema na plataforma Polygon. Já os comandos `pull` e `push` utilizam dessa conexão autenticada para efetuar alterações e leituras no problema salvo nos servidor.

O teste de conexão com o Polygon exige a existência de duas informações: a chave de API do usuário e o segredo da API do usuário. As informações de autenticação ficam em um arquivo de configuração comum para todos os problemas locais. Ao fazer o teste de comunicação é utilizado o método da API de listar os problemas do usuário (`problems.list`), o qual não demanda nenhuma outra informação, tal como o identificador de um problema, então é possível testar a qualquer momento a conexão sem a necessidade de um problema local.

O subcomando `pull` é responsável por se conectar com o Polygon, ler as informações e arquivos de um problema específico e copiá-lo para os arquivos locais. Assim o usuário possui uma versão do problema igual às dos servidores do Polygon. O subcomando *push* faz o caminho inverso, fazendo com que o problema salvo no servidor altere-se para ficar igual os arquivos locais do usuário. Para isso é usado métodos da API para alterar os arquivos e informações do problema no servidor. Atualmente não há nenhum tratamento de conflitos em nenhum dos subcomandos. Ambos precisam também de informações presentes tanto dos dados de autenticação com o Polygon quanto o identificador do problema presente na plataforma Polygon, que fica salvo no arquivo de configuração `config.json`.

1.3.2 Padronização de Arquivos

O arquivo de configuração `config.json` é responsável por manter o caminho para cada arquivo necessário pela ferramenta, tal como o verificador, o validador, os testes, etc. Um arquivo modelo[12] é usado ao criar um problema com o comando `init` e ele reforça os

[12] https://github.com/edsomjr/competitive-problems-tools/blob/master/templates/config.json

padrões como manter os testes em um diretório denominado `tests`.

O arquivo `config.json`, além de manter os caminhos dos arquivos de cada recurso, também possui outras informações do problema como o autor e seus dados, o título do problema, o idioma usado na descrição, o tempo limite, a memória limite, palavras-chaves, a competição na qual o problema foi aplicado primeiramente, o tipo de cada solução descrita, etc. Esse é um arquivo importante e utilizado constantemente pela ferramenta para garantir persistência nas configurações de um problema específico.

Também existe um arquivo de configuração universal que é independente do problema sendo desenvolvido, de modo que ele não está contido no diretório de um problema. Em sistemas operacionais Linux ele é criado no diretório principal do usuário (*home directory*) com o nome de `.cp-tools-config.json`. Nele estão presentes informações como as de autenticação do Polygon. Como esse arquivo possui informações sensíveis é importante que ele tenha permissão de leitura apenas pelo usuário ao qual pertence, para isso são utilizadas as funcionalidades de segurança do próprio sistema operacional.

1.4 Testes de *Software*

O ato de testar um *software* tem como objetivo aferir a qualidade do mesmo e identificar defeitos e problemas. Para isso utiliza-se da comparação de um conjunto finito de casos de testes com seus comportamentos esperados (ABRAN et al., 2004).

Em Cohn (2010) é exibido uma variação da Figura 5 que representa a pirâmide de testes. Ela é fracionada em três níveis, sendo a base os testes unitários, seguidos dos testes de integração e por fim os testes de ponta a ponta. Quanto mais próximo da base maior o isolamento e o esforço necessário para cobrir mais casos de teste é menor.

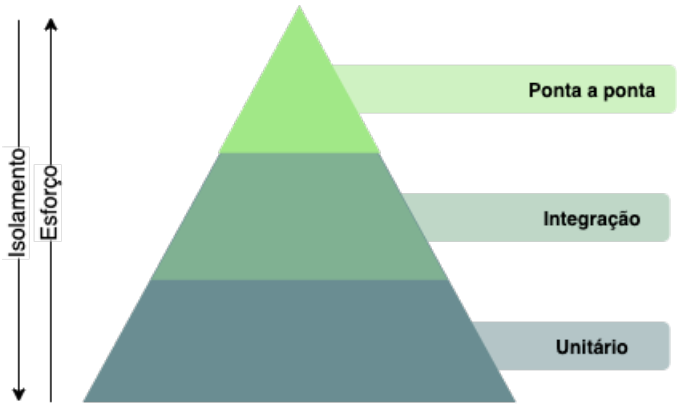

Figura 5 – Representação da pirâmide de testes

Enquanto os testes unitários verificam as menores partes do *software* de maneira isolada, são também os testes mais simples de serem implementados. Isso é uma consequên-

cia de que eles não dependem de outras partes do código, apenas de partes atômicas como uma função ou classe.

Os testes de integração validam as interações de duas ou mais partes do *software*. Eles quebram a ideia de isolamento imposta pelos testes unitários. Apesar de se aproximarem mais do funcionamento do *software* em um caso de uso real, eles exigem maior empenho para serem implementados.

Por último os testes de ponta a ponta que focam no comportamento do *software* por inteiro, reproduzindo o máximo possível da perspectiva do uso de um usuário. Esses possuem baixo isolamento e consomem bastante tempo e esforço para serem implementados.

1.5 Trabalhos Correlatos

Outras plataformas de juiz online como o URI[13] também possuem seus formatadores de problemas próprios, que nesse caso é o URI Builder[14]. Essa plataforma apresenta uma interface web porém nenhuma outra maneira de criar e gerenciar problemas como uma API. Ela também possui muitas características em comum com o cp-tools, como a possibilidade de descrever o problema em mais de um idioma, especificar entradas e saídas de exemplo e inserir a instituição de ensino a qual está afiliado o responsável pela criação do problema.

Algumas diferenças do URI Builder quando comparado ao cp-tools são relacionadas a especificidades da plataforma URI, tal como a categorização do problema (*category*), que pode ser uma das oito apresentadas e que não necessariamente vão ser iguais a de outras plataformas. Outra diferença é o estilo do problema (*style*) que altera os campos necessários a serem preenchidos para a criação do problema. Um dos estilos é o de editorial no qual, ao invés de apresentar um problema a ser resolvido, se insere a solução de um problema.

A plataforma Kattis[15] é um arquivo de problemas e competições disponíveis para o usuário solucionar a qualquer momento. Existe uma documentação[16] que esclarece como empacotar um problema para a Kattis. O desenvolvimento do padrão de empacotamento é aberto em um repositório do GitHub[17], possibilitando receber contribuições externas. A ferramenta responsável pela construção dos problemas ainda está em desenvolvimento e diversas partes da documentação são grifadas para representar partes ainda não implementadas.

[13] https://www.urionlinejudge.com.br/
[14] https://www.urionlinejudge.com.br/builder
[15] https://open.kattis.com/
[16] https://www.kattis.com/problem-package-format/
[17] https://github.com/Kattis/problem-package-format

A Kattis possui suporte para 19 linguagens, muitas não suportadas pelo cp-tools, tais como C#, Pascal, Objective-C e Prolog. Possui também diversas informações em cada problema como argumentos a serem passados para o validador e a separação entre autor e o detentor dos direitos do problema. A maior diferença entre a Kattis e o cp-tools é a rigidez na estrutura de arquivos. Enquanto o cp-tools apresenta uma flexibilidade oferecida pelo arquivo de configuração `config.json`, a Kattis define e limita como lidar com os arquivos e diretórios do pacote, de modo que o usuário não tem a opção de configurar de maneira personalizada essa parte. Ao possuir maior rigidez na estrutura de diretórios e arquivos do problema as configurações do problema se tornam mais simples por possuírem menos opções. Por exemplo, as soluções são categorizadas de acordo com o diretório que está presente ao invés de ter sua categoria exposta na configuração por aquivo, enquanto nas configurações do cp-tools existem chaves e valores a serem preenchidos com essas informações.

Uma outra ferramenta notável é o `polygon-cli`[18] que implementa uma interface em linha de comando para se comunicar com o Polygon. A maior diferença entre ele e o `cp-tools` é que o `polygon-cli` não tem a intenção de ser uma interface comum entre várias plataformas. Na verdade, ela é apenas uma interface para uma única plataforma. Apesar de sua adesão não ser alta dentro da comunidade do Polygon, é relevante já que implementa comandos que estão sendo implementados dentro do `cp-tools`.

O comando do `polygon-cli` equivalente ao `push` do `cp-tools` é o `commit`, enquanto o equivalente ao `pull` é o `update`. Além disso ele apresenta outras funcionalidades como o comando `diff` que gera a diferença entre os arquivos locais e remotos sem necessariamente sobrescrevê-los. O projeto também é de código aberto e possui diversos contribuidores, assim como o `cp-tools`.

[18] https://github.com/kunyavskiy/polygon-cli

2 Metodologia

Este capítulo explica a metodologia utilizada neste trabalho. Na Seção Entendimento do Domínio é explicado como foi a abordagem para obter o entendimento do domínio, a Seção Desenvolvimento do Código descreve como foi feito o desenvolvimento do código e por fim a Seção Ferramentas expões quais e como foram utilizadas as ferramentas neste projeto.

2.1 Entendimento do Domínio

Antes de iniciar o desenvolvimento foi necessário um entendimento do domínio ao qual pertence o *software*. De acordo com o DDD (*Domain Driven Design*), ter especialistas do domínio e desenvolvedores em constante comunicação e troca de conhecimento é importante e benéfico para o projeto (VERNON, 2013). No caso deste trabalho os desenvolvedores atuaram como ambas as partes.

Para evoluir a compreensão sobre o domínio da criação de problemas para competições foi feito o uso da ferramenta formatadora de problemas Polygon. Ao utilizá-la foi possível compreender melhor o processo de concepção de um problema de competição, os arquivos e os recursos necessários. Diversos termos existentes dentro do domínio da formatação de problemas foram apresentados e compreendidos, tais como *checker* e *validator*.

Após o entendimento do domínio usando o Polygon foi necessário o entendimento da ferramenta cp-tools para compreender melhor como ela funciona. De acordo com Collofello (1989), o entendimento do *software* é uma das etapas para sua manutenção, seguido de propostas para melhorias do mesmo. Após o uso então foram propostas algumas melhorias e apontados alguns defeitos, como por exemplo comandos que não estavam efetuando o processo esperado por eles. Foram efetuadas manutenções corretivas cujo objetivo foi também se familiarizar com o código e as ferramentas utilizadas para desenvolvimento.

2.2 Desenvolvimento do Código

Durante todo o desenvolvimento da ferramenta foi priorizado o uso de dois princípios da Metodologia Ágil. De acordo com o Manifesto para Desenvolvimento Ágil de Software (BECK et al., 2001), é importante entregar novas versões do software com frequência e reflexões constantes do time sobre o projeto com o objetivo de ajustar e melhorar o desenvolvimento. Para cumpri-los foram feitas reuniões semanais com os seguintes objetivos:

validação das tarefas finalizadas na semana anterior, planejamento das tarefas futuras e reflexão do andamento do desenvolvimento.

A validação de tarefas finalizadas consistiu na revisão em pares dos *pull requests*, os quais continham todas as modificações propostas. A revisão em pares se apresenta de maneira mais natural para projetos de código aberto, é visto como um método eficaz de garantir a qualidade do *software* (RIGBY; BIRD, 2013). A revisão foi dividida em quatro etapas: envio da proposta de alteração, revisão por uma pessoa diferente da autora, modificações nas alterações até que sejam consideradas dentro dos padrões de qualidade do projeto e por fim a adição das alterações à base de código compartilhada.

Durante todo o desenvolvimento foi priorizado a manutenção contínua do código. Swanson e Chapin (1995) definem uma topologia para tipos de manutenção de um *software*, sendo elas: corretivas, adaptativas e perfectivas. Dessas o desenvolvimento teve um foco maior nas corretivas, responsáveis por corrigir defeitos já existentes. Também foi priorizado as manutenções preventivas cujo objetivo é evitar que problemas novos surjam ao decorrer do aumento de complexidade do *software* e seu envelhecimento (LIENTZ; SWANSON, 1980).

Um dos focos do desenvolvimento do código foi a criação de testes automatizados para acompanhar sua evolução. Adicionar essa etapa à integração contínua e resolver os problemas apresentados nela imediatamente foi uma maneira efetiva de se evitar problemas, tais como defeitos no *software* (BERNER; WEBER; KELLER, 2005). A maneira escolhida para implementar os testes foi utilizando o BDD (*Behaviour-Driven Development*), que evoluiu a partir do TDD (*Test-Driven Development*) para cobrir problemas nele (MOE, 2019).

Ferramentas

A ferramenta Git[1] foi utilizada para versionar o código. Ela oferece vantagens como a troca fácil de contexto a partir do uso de *branches* e a definição de um contexto (*branch*) que reproduz o estado da produção do *software*. A plataforma GitHub[2] foi usada para possuir uma versão compartilhada da base de código, além disso ela oferece as ferramentas de *pull requests* que são utilizadas durante as revisões em pares.

Quando um projeto possui integrações de novos códigos frequentemente, como no caso do cp-tools, automatizar parte dos testes para obter resultados automatizáveis o mais rápido possível reduz problemas e o tempo gasto por desenvolvedores, assim garantindo maior produtividade (FOWLER; FOEMMEL, 2006). Por essa razão foi utilizado o

[1] https://git-scm.com/
[2] https://github.com/

Travis CI[3] como ferramenta de integração contínua. Esta ferramenta apresenta também uma integração nativa com o GitHub. No contexto do cp-tools a integração contínua foi responsável por garantir a compilação e linkedição do *software*, além de uma etapa que verificava se o código seguia as formatações definidas pelo projeto.

A linguagem de programação utilizada para construção da ferramenta foi o C++17, pois mantê-la foi uma das restrições deste trabalho. Esta restrição era imposta pois seria necessário portar todo o código já desenvolvido anteriormente e o tempo planejado para este trabalho não seria suficiente nesse caso.

O `cp-tools` tem como objetivo ser multi-plataforma, abordando diversos sistemas operacionais, e isso restringe o código que pode ser escrito e utilizado. Outras linguagens como o Python[4] e o Golang[5] apresentam maneiras mais fáceis de lidar com o problema de múltiplas plataformas. Por exemplo, ao usar chamadas de sistema no C++ é necessário cautela pois cada sistema operacional possui as suas próprias. Linguagens de programação como as citadas abstraem essa parte do desenvolvimento. Essa restrição exigiu manutenções corretivas nos módulos responsáveis por manusear o sistema de arquivo do usuário já que usavam código específico para apenas um sistema operacional.

Uma das restrições do projeto é evitar o uso de dependências dinâmicas. Por esse motivo foi dado prioridade ao uso de bibliotecas *single-file header-only* de terceiros, constituídas de apenas um arquivo de cabeçalho. Esses foram utilizados para casos como conexão HTTPS, necessária para se comunicar com a API do Polygon, e manuseio de arquivos do tipo JSON, utilizados nos arquivos de configurações. Dependências dinâmicas evita que o usuário tenha mais esforço ao instalar o programa em sua máquina local, porém aumenta o tamanho do executável e seu tempo de compilação.

[3] https://travis-ci.com/
[4] https://www.python.org/
[5] https://golang.org/

3 Resultados

Este capítulo expõe os resultados deste trabalho explicando cada um deles, os problemas ocorridos e como foram solucionados.

3.1 Evolução e Manutenção do Código

Durante o trabalho foram feitos 16 *pull requests*, mais de 130 *commits* que inseriram mais de 52 mil linhas de código e removeram mais de 29 mil. O número alto de alterações é resultado, além da criação de novas funcionalidades, de diversas refatorações no código que serão explicadas nesta seção.

As evoluções e manutenções, que serão explicadas nas próximas subseções, são:

- criação de um protótipo de comunicação com o Polygon;
- criação de testes automatizados da integração com o Polygon;
- refatoração do sistema de arquivos;
- atualização da versão do C++ e do compilador;
- criação do pacote `cli`;
- melhorias na qualidade do código;
- implementação do comando `polygon` e do seu subcomando `pull`;
- implementação de tratamento de conflitos no comando `polygon pull`;
- evolução do pacote responsável por lidar com os arquivos de configuração;
- atualização das bibliotecas estáticas *single-file header-only*;
- evolução e manutenção da arquitetura do projeto.

3.1.1 Protótipo de Comunicação com o Polygon

Primeiramente foi construído um protótipo da conexão entre o cp-tools e o Polygon. O objetivo era demonstrar a viabilidade da comunicação entre os dois dada a complexidade de autenticação com a API do Polygon. Nessa etapa também houve a evolução dos arquivos de configuração para que eles possuíssem os dados de autenticação do Polygon, compostos por uma chave e um segredo únicos para cada usuário.

Apesar da ferramenta ter como uma meta a redução do uso de dependências dinâmicas, o que é um dos desafios do projeto, foi decidido utilizar a `libssl` como biblioteca para a criptografia necessária para comunicações HTTPS. O protótipo foi iniciado com diversas bibliotecas estáticas a fim de testes. Porém, ao final foi constatado que o uso da biblioteca dinâmica libssl seria a escolha mais efetiva dada também a complexidade de autenticação com a API do Polygon, que exige uso de criptografia além da utilizada no protocolo HTTPS. Entretanto, foi possível manter as funções para conexão HTTPS em uma dependência estática.

Durante o desenvolvimento foi priorizada a criação de testes automatizados. Os testes do protótipo de comunicação com o Polygon precisavam ser executados durante os processos da integração contínua. Para isso, a ferramenta Travis CI precisa ter conhecimento dos dados de autenticação de maneira segura. A própria ferramenta fornece variáveis de ambientes secretas, que podem ser acessadas apenas pelo dono do projeto, e essas foram usadas como solução. Também foi criado uma conta apenas para esses testes, mantendo a segurança de ser uma conta isolada e que não seria usada por ninguém.

O uso das variáveis de ambiente secretas do Travis CI trouxeram uma desvantagem. Até então os *pull requests* eram criados a partir de *forks* do projeto oficial, porém por questões de segurança essas variáveis não ficam expostas a eles, assim então durante a fase de testes da integração contínua era sempre resultava em falhas, pois a chave e o segredo de autenticação não estavam disponíveis. A solução foi interromper o uso de *forks* e começar a codificar em *branches* dentro do projeto oficial.

3.1.2 Refatoração do Sistema de Arquivos

Ao decorrer do desenvolvimento, foram feitas refatorações no código para torná-lo mais próximo do seu objetivo de ser multi-plataforma. Portanto, houve um extenso trabalho em substituir acessos de leitura e escritura ao sistema de arquivos, que antes eram feitos a partir de comandos Linux e que agora utilizam a biblioteca `filesystem` da linguagem C++. Foi criado um novo módulo no código com a responsabilidade única de manter funções que encapsulam as funções dessa biblioteca, facilitando a futura evolução e manutenção do código.

Diversos pacotes eram diretamente dependentes do pacote de sistema de arquivos, logo uma alteração no código do sistema de arquivos apresentou efeitos colaterais nos seus dependentes. As alterações causaram falhas em partes do programa que já eram consideradas estáveis, e esse resultado poderia ser evitado a partir do uso de testes automatizados. Eles evitariam esse problema, pois suas execuções identificariam os defeitos antes de dar como finalizado a refatoração do sistema de arquivos.

A refatoração do gerenciamento de arquivos para usar a biblioteca `filesystem`

levou às novas atualizações na integração contínua e requisitos do projeto. Para que a compilação ocorra sem erro foi necessário o uso de compiladores com versões mais recentes dos que eram utilizados antes. Foi efetuada a troca para a versão 9.3.0 do compilador G++ ocorresse uma troca da versão da máquina virtual que executava as etapas de compilação e linkedição.

Durante o desenvolvimento se tornou perceptível que diversas funcionalidades do C++20 poderiam ser usadas a fim de facilitar o desenvolvimento. Um dos exemplos utilizados durante o desenvolvimento do comando `pull` para o Polygon é a parte de filtros (*range views*[1]). Portanto, foi efetuado novamente uma atualização da versão do compilador e a versão atual do G++ no projeto é a 10.3.0. Os arquivos de documentação e bibliotecas tiveram que ser atualizadas para ter suporte à nova versão do C++.

3.1.3 Criação do Pacote `cli`

A principal interface de comunicação com o usuário atualmente é a interface de linha de comando (*CLI - command line interface*). Considerando isso foi criado um pacote chamado `cli` para facilitar a aplicação de boas práticas, melhorar a usabilidade da interface, e também para reduzir o acoplamento entre interfaces de comunicação com o usuário e outras partes do código.

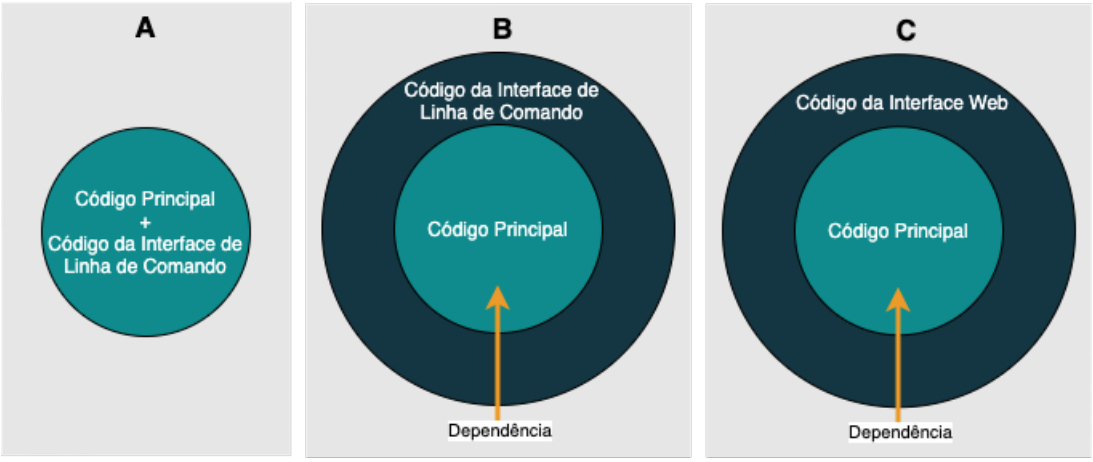

Figura 6 – Representação da dependência entre interfaces de comunicação e a parte principal do código

Na Figura 6 é representado na parte A como o código estava organizado no início deste trabalho. O código principal, responsável pelas principais operações como o comando `polygon pull`, e o código responsável por lidar com a interface de linha de comando está fortemente acoplado. Esta primeira refatoração causada pela criação do pacote `cli` foi a primeira evolução para alcançar a parte B, em que a dependência entre as partes do

[1] https://en.cppreference.com/w/cpp/ranges

código ganham uma direcionalidade e um sentido único: o mais externo depende do mais interno.

Na parte B da Figura 6 os códigos mais externos dependem dos mais internos, porém os internos não tem conhecimento dos externos, isso é representado pela direção da seta de dependência. A intenção é que a evolução e manutenção do código de interface se torne mais fácil e menos propenso a erros. Outra consequência é a facilidade de troca da interface para a de outro tipo sem afetar o código principal, isso é representado na parte C da figura.

Antes da criação do pacote `cli` as variáveis de *output stream*, usadas para enviar as mensagens de saída na interface de linha de comando, eram passadas por argumento desde a `main` até quem pretendia usá-la. O problema dessa abordagem é que sempre que alguma função precisar utilizar essas variáveis é necessário passar ela por múltiplas funções como parâmetro, inclusive por funções que não as utilizariam. Isso gerava código desnecessário e fazia com que funções tivessem que ter conhecimento de variáveis as quais não necessariamente fariam uso.

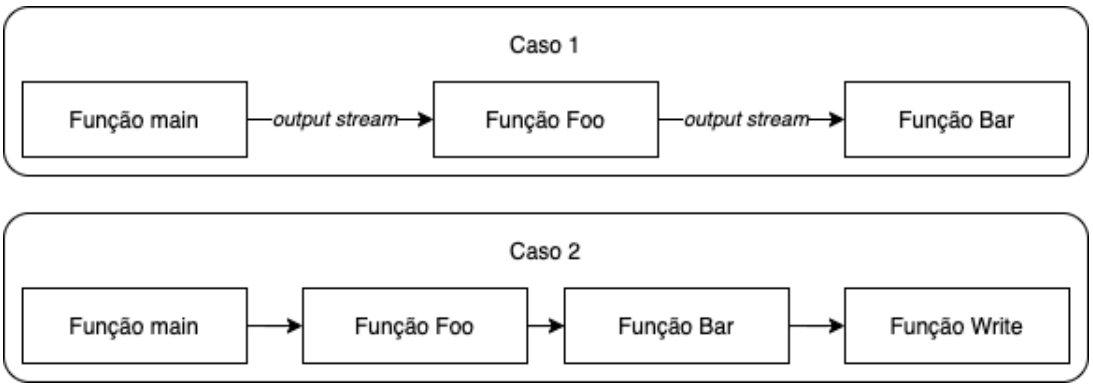

Figura 7 – Fluxos de chamadas de funções nos casos de haver ou não o pacote `cli`

No cenário hipotético da Figura 7 podemos visualizar o problema da passagem de parâmetros excessiva. Nesse caso a função Bar precisa usar a variável de *output stream* enquanto a Foo só recebe para poder passá-la para a Bar, mas não faz uso dela. Após a implementação do pacote `cli` nós entramos no segundo caso, tambem hipotético, em que a função Foo não precisa mais saber da existência dessa variável, e a função Bar pode chamar a função Write, responsável por manter uma referência às variáveis de *output stream*.

Na Figura 9 podemos ver um exemplo do uso do pacote `cli`. As mensagens apresentam diferentes formatações para facilitar o senso de importância de cada uma. Na mesma figura cumpre-se algumas das boas práticas recomendadas como avisar quando o programa altera seu estado e apresentar mensagens durante processos longos para evitar que o usuário fique em dúvida se o comando está funcionando corretamente. Outras

boas práticas já vinham sendo aplicadas desde antes deste trabalho, como por exemplo as mensagens de ajuda ao utilizar a opção `-h`.

3.1.4 Melhorias de Qualidade de Código

Diversas mudanças menores foram feitas simultaneamente com objetivo de melhorar a qualidade do código. Módulos do código foram divididos em outros submódulos para atingir a responsabilidade única deles, tal como o módulo de configuração. Regras de formatação do código baseadas no ClangFormat[2] foram inseridas, automatizadas e exigidas na etapa de integração contínua. A inserção de uma etapa responsável pela validação da formatação do código obrigava os desenvolvedores a mantê-las, já que a falha da integração contínua causava a rejeição do *pull request*.

Diversas partes do código foram alteradas para convergir com as boas práticas apresentadas no livro *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* (SUTTER; ALEXANDRESCU, 2004). Por exemplo, foram removidos os usos das diretivas `using` em cabeçalhos e priorizado o uso das estruturas de dados corretas de acordo com o caso de uso. Apesar da maioria dessas manutenções terem sido preventivas, elas também focavam em aumentar a legibilidade do código para facilitar contribuições ao projeto.

3.1.5 Criação do Subcomando `polygon pull`

Um dos principais resultados do trabalho foi a criação do subcomando `pull` do comando `polygon`. Ele é responsável por buscar o identificador do problema sendo trabalhado, autenticar o usuário à API do Polygon e atualizar a versão local do problema com a versão remota a partir da comunicação com a API da plataforma Polygon. Na Figura 8 é apresentado o fluxo seguido pelo *software* ao utilizar esse comando.

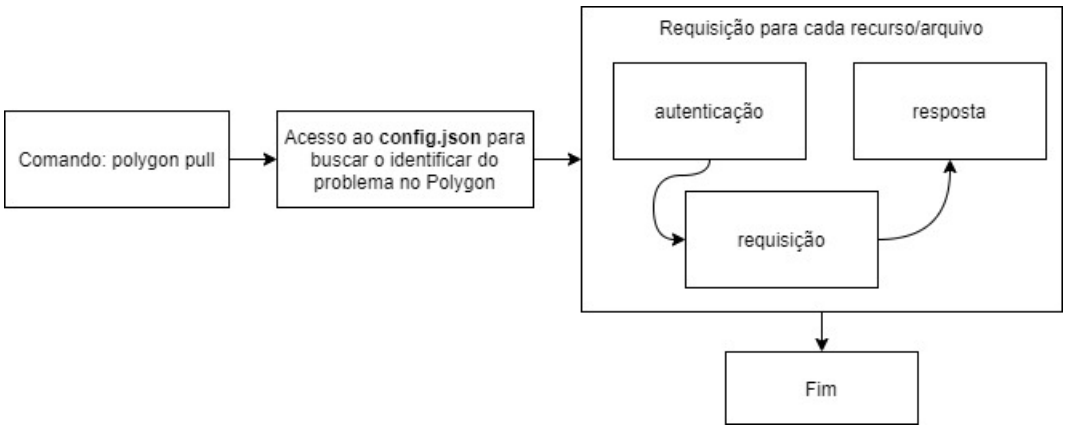

Figura 8 – Representação do fluxo ao utilizar o comando `polygon pull`

[2] https://clang.llvm.org/docs/ClangFormat.html

Nomes iguais	Conteúdos diferentes	Resultado
Sim	Sim	Conflito
Não	Não	Conflito parcial
X	X	Sem conflito

Tabela 1 – Casos de tratamento de conflitos entre arquivos

Atualmente, o comando `polygon pull` tem dois padrões de funcionamento: normal e forçado. Por padrão a execução do comando seguirá o funcionamento normal, mas caso a opção `-f` seja adicionada ele utilizará o modo forçado. Esse funcionamento foi baseado na ferramenta de versionamento de código Git e se comporta de maneira parecida.

Quando o comando `pull` é executado de maneira forçada ele ignora quaisquer conflitos que aconteçam, e o cp-tools irá sempre substituir os arquivos locais pelos presentes na versão remota, ignorando qualquer modificação que possa ter ocorrido localmente. Esse funcionamento foi implementado pensando em casos onde se quer ignorar a versão local, porém no caso do funcionamento normal existe um algoritmo que analisa os conflitos e é responsável por decidir o resultado final.

Para identificar a existência de conflito entre dois arquivos foram consideradas duas variáveis: se os nomes dos arquivos são iguais e se os seus conteúdos diferem. Para definir se os arquivos tem diferentes conteúdos é usado o algoritmo de *hash* SHA-512 em cada, e depois o resultado é comparado. A Tabela 1 expõe as possíveis junções do valor das variáveis e seus respectivos resultados que podem ser conflitos, conflitos parciais ou sem conflito.

Quando ambas variáveis tem resultado positivo, como representado pela primeira linha da Tabela 1, nós temos um conflito. O arquivo local e remoto possuem o mesmo nome, porém conteúdos diferentes. Para tratar isso é feito uma cópia do arquivo local adicionando a extensão `.old` ao final dele, e por fim o arquivo original é sobrescrito com o conteúdo do arquivo remoto.

Na ocasião em que os nomes são diferentes e o conteúdo de ambos são iguais, como representado pela segunda linha da Tabela 1, é considerado como se o arquivo apenas tivesse sido renomeado e por isso um conflito parcial. Nesse caso, a única ação necessária é reproduzir isso localmente, isto é, alterar o nome do arquivo local para condizer com a versão remota dele.

Em todos os outros casos é considerado que não houve conflito, por exemplo, quando os nomes dos arquivos são diferentes e os conteúdos também. Nesse caso é considerado que um arquivo novo foi adicionado e não que um existente foi alterado. Sempre serão mostradas mensagens aos usuários sobre como o cp-tools está lidando com o conflito. Um exemplo é exposto na Figura 9.

```
vscode ➜ /tmp/tmp.KlIxFaAmVA $ cp-tools polygon pull
Local config.json copied to config_2021-09-29_17:10:40.json.old
Pulling checker...
Warning! The file 'tools/checker.cpp' will be moved to 'tools/checker.cpp.old'
Pulling validator...
Pulling solutions...
Pulling tests...
Warning! The file 'tests/1' will be moved to 'tests/1.old'
Pulling problem title...
Pulling problem informations...
Pulling problem tags...
Ok! Pull completed
```

Figura 9 – Exemplo de mensagens expostas na interface da linha de comando após o comando `polygon pull`

Para evitar quaisquer perdas das configurações do problema presente no arquivo `config.json` é efetuado uma cópia de segurança dele toda vez que o comando `polygon pull` é feito. Esse arquivo de cópia tem, em seu nome, uma marca da data e hora que o comando foi executado e extensão `.old`. A primeira mensagem após a execução do comando na Figura 9 é relacionada à criação desse novo arquivo.

Para que o código responsável pela comunicação com a API do Polygon fosse de fácil manutenção e evolução, foi criado um módulo com essa responsabilidade única. Ele foi criado pensando também na escalabilidade, para que seja fácil criar módulos responsáveis pela comunicação com outras APIs de outras plataformas futuramente. Também foi criado o módulo de tipos, responsável por criar estruturas de dados que representem os objetos contidos nas respostas da API.

Além da necessidade de evolução nas partes responsáveis por acesso ao sistema de arquivos, houve uma intensa evolução no código responsável por manusear arquivos do tipo JSON. Assim, se torna possível não só buscar arquivos remotos, como alterar as configurações do problema local para estarem de acordo com as mudanças.

Na Figura 10 é demonstrado parte do processo do comando `pull`. Nela pode-se ver que ao usá-lo, quando um arquivo possui nomes diferentes no servidor do Polygon e no repositório local, há necessidade de alteração no arquivo de configuração. Diferentemente de quando o nome é igual, nesse caso apenas é feita a substituição do conteúdo do arquivo. Para manipulação dos arquivos do tipo JSON foi incluído uma biblioteca[3] estática do tipo *single-file header-only*.

3.1.6 Evolução da Arquitetura

Na Figura 11 podemos ver a arquitetura inicial do projeto, antes das contribuições deste trabalho. Dividido sem nenhuma hierarquia, ela apresentava 14 módulos, problemas com as responsabilidades de cada um e inexistência de uma organização apropriada.

[3] https://github.com/nlohmann/json

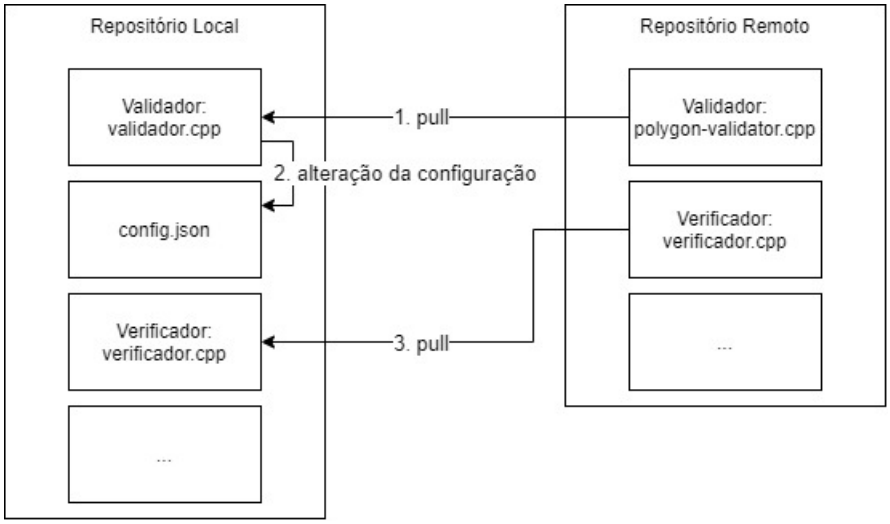

Figura 10 – Fluxo a partir do comando `pull` para o verificador e o validador

Ambas adversidades são um problema para um desenvolvedor recém-chegado, logo foi um objetivo solucioná-las.

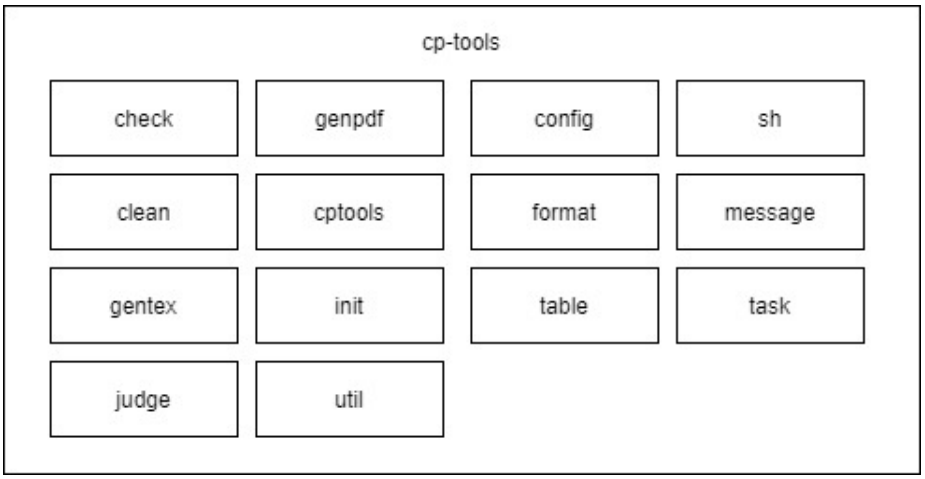

Figura 11 – Arquitetura inicial antes das contribuições

Já na Figura 12 é representado a arquitetura atual, após as contribuições originadas desse trabalho. A complexidade do projeto aumentou, porém a organização e a arquitetura passaram por melhorias para acompanhar. Os comandos foram separados em seus próprios módulos, e comandos que possuem subcomandos se tornaram módulos. Foi criado o módulo para o código responsável para cada API e para os tipos dos objetos que essas APIs usam para se comunicar.

3.1.7 Criação do subcomando `polygon push`

Apesar de fazer parte do escopo deste trabalho, por causa das restrições de tempo o comando `polygon push` não foi implementado. A prioridade foi dada ao `pull` já que o

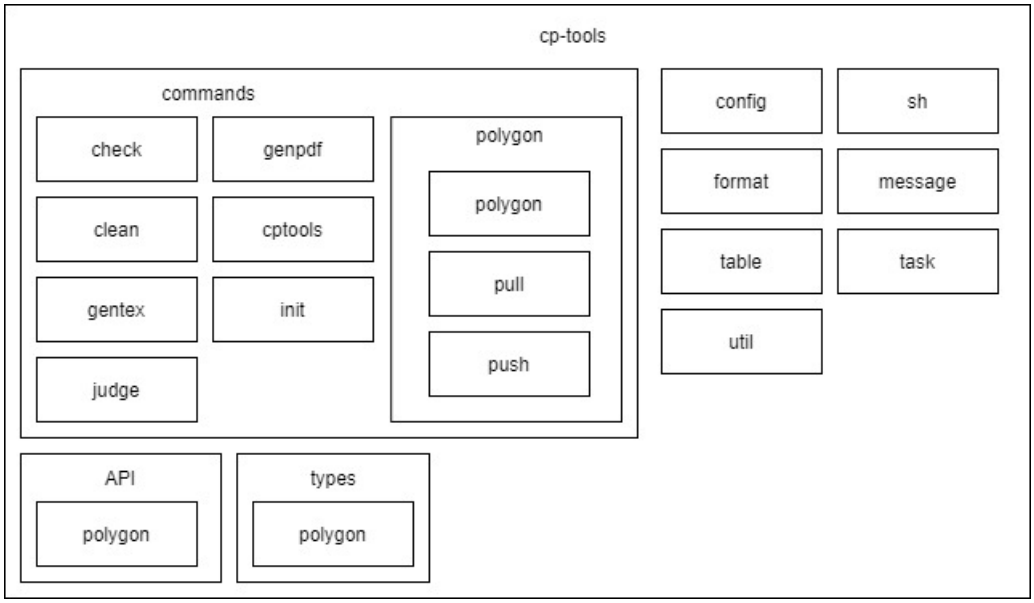

Figura 12 – Arquitetura final após as contribuições

`push` exige uma implementação eficaz e completa dele para funcionar corretamente. Na Figura 13 é apresentado o fluxo de quando o comando de `push` é executado. A comparação feita na segunda etapa entre o repositório local e remoto depende de métodos que façam leituras na API e que foram implementados durante o desenvolvimento do subcomando `pull`.

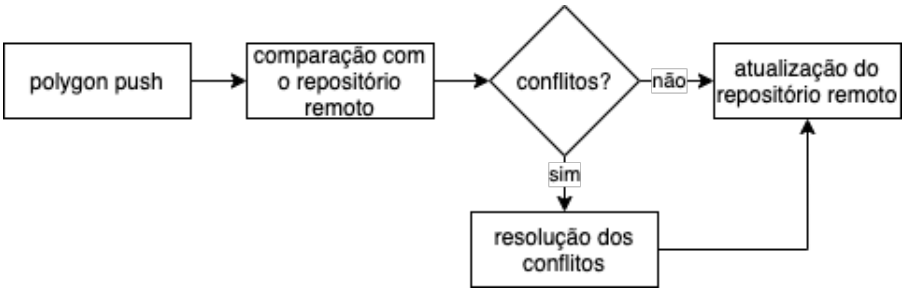

Figura 13 – Fluxo do `polygon push`

A partir da comparação da segunda etapa é possível verificar se há conflitos entre os dois repositórios. Isso pode acontecer quando alguma modificação foi feita no repositório remoto e que não foi adicionada ao local, então quando o `push` for efetuado, se não houver tratamento desses conflitos, essas mudanças podem ser sobrescritas acidentalmente. Análogo ao `pull`, é esperado que o `push` tenha uma opção para atualizar o repositório remoto forçadamente, sobrescrevendo quais alterações existentes nele.

4 Considerações Finais

Este trabalho fundamentou a existência e o uso das competições de programação, e também se propôs a desenvolver funcionalidades da ferramenta `cp-tools`. As competições de programação demonstraram serem efetivas quando usadas com objetivos educacionais, como por exemplo lecionar programação. Também são usadas por empresas e instituições acadêmicas como um esporte mental, em que apresentam prêmios atrativos para diversos participantes.

O objetivo da ferramenta `cp-tools` é facilitar a criação de competições em diversas plataformas ao criar uma interface comum entre elas e automatizando o máximo possível desse processo. Com isso em mente o principal objetivo deste trabalho foi a implementação da comunicação entre o `cp-tools` e a plataforma Polygon. Essa comunicação acontece a partir do uso dos comandos `polygon pull` e `polygon push`.

Apenas o `pull` foi implementado por restrições de tempo, porém diversas manutenções e evoluções foram feitas para aumentar a qualidade do código. A estrutura de arquivos foi alterada, a arquitetura foi melhorada, boas práticas de qualidade de código foram aplicadas, pacotes de código para lidar com algoritmos de conflito foram implementados, e refatorações foram feitas em pacotes já existentes.

Para que os resultados fossem atingido com êxito utilizou-se de métodos conhecidos na área de Engenharia de *Software* como o *Domain Driven Design*. Fluxos de desenvolvimento que utilizam as práticas ágeis foram aplicados junto com boas práticas, como a criação de testes automatizados e aplicações de manutenções constantes.

Os resultados provenientes deste trabalho aumentaram a qualidade da ferramenta `cp-tools`, também a aproximaram mais do seu objetivo final de ser uma interface comum entre diversos juízes *online*. Além disso, a ferramenta é de código aberto e a complexidade do código reduziu, o que torna contribuições externas mais fáceis.

Apesar de que diversas melhorias foram feitas, o `cp-tools` ainda possui espaço para muitas evoluções e aperfeiçoamentos. Atualmente a cobertura de testes ainda está baixa, o tempo de resposta da comunicação com a API do Polygon está lenta e a ferramenta ainda carece de melhores maneiras para lidar com os conflitos.

Durante 25 execuções de cada, o comando `polygon` apresentou média de tempo decorrido de 1 segundo, enquanto o comando `polygon pull` teve uma média de 15 segundos. De acordo com Hoxmeier e DiCesare (2000), o aumento do tempo de resposta é proporcional à insatisfação do usuário.

Esse alto tempo de resposta se dá às requisições feitas de maneiras sequencias à

API do Polygon. O cp-tools divide o comando em diversas partes representadas por cada tipo de arquivo a ser acessado e faz um acesso por vez, porém esses acessos costumam demorar por causa do tempo de resposta do Polygon.

Na documentação oficial do Polygon não é apresentado nenhum limite máximo de requisições por tempo, logo a paralelização dessas etapas do `pull` podem ser a solução para reduzir o tempo de resposta.

Atualmente o cp-tools possui diversos testes de ponta a ponta baseados no comportamento do programa, porém não são um conjunto de casos de testes exaustivo e que exploram por completo os comportamentos do cp-tools do ponto de vista do usuário. Isso por si só já se apresenta como um problema, pois a evolução de um *software* apoiada por uma cobertura de testes maior vai ser mais estável, ou seja, apresentando menos defeitos.

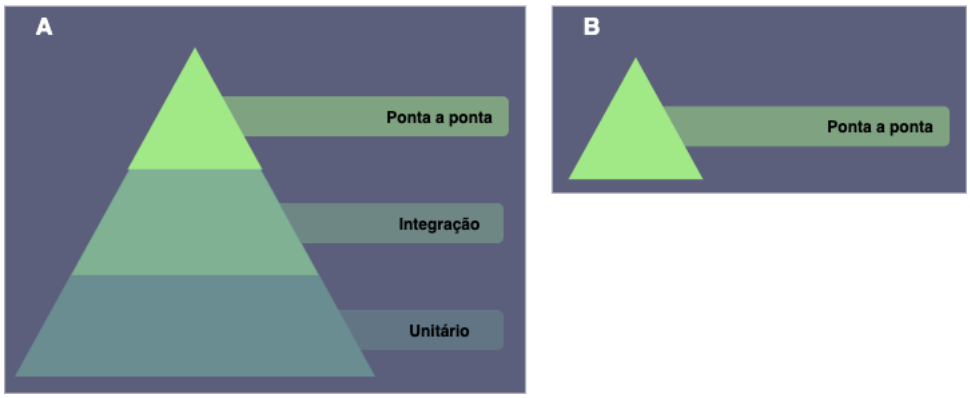

Figura 14 – Comparação das pirâmides de testes

Na seção A da Figura 14 é exposto a pirâmide de testes ideal para um *software*, que possui os três tipos de testes. Na seção B é representado o estado atual do cp-tools, em que possui apenas os testes de ponta a ponta. Algoritmos mais complexos como os responsáveis pelo tratamento de conflitos de arquivos devem possuir testes unitários para validá-los, enquanto a interação entre o cp-tools e plataformas como o Polygon necessitam de testes de integração para garantir que a comunicação entre ambas não possui defeitos.

Os problemas apresentados ao final da Subseção 3.1.1 podem ser resolvidos com

uma aplicação apropriada de testes de integração. Para isso seria necessário o uso de um *mock* para a API do Polygon, cujo objetivo é reproduzir o comportamento da plataforma sem necessitar da autenticação. Assim problemas de segurança como ter as chaves e segredos da API na integração contínua poderiam ser resolvidos e uso de *forks* poderiam retornar para manter o padrão de um projeto de código aberto.

Os comandos `pull` e `push` foram baseados na ferramenta Git, porem esse tem diversas maneiras de lidar com conflitos que ele chama de estratégias[1]. Enquanto essas se baseiam em maneiras mais complexas de detectar conflitos usando algoritmos como o Algoritmo de Diferença de Myers (*Myers Diff Algorithm*), o cp-tools se baseia em variáveis simples como a diferença geral do conteúdo ou do nome dos arquivos.

Implementações mais robustas são interessantes para melhorar a usabilidade e a facilidade de resolver os conflitos. Para isso é recomendado a implementação de uma interface que abstraia as possíveis estratégias e torne fácil a adição de novas. Dessa maneira será mais fácil estender as possíveis estratégias sem a necessidade da modificação de código já existente.

[1] https://git-scm.com/docs/merge-strategies

Referências

ABRAN, A. et al. Software engineering body of knowledge. *IEEE Computer Society, Angela Burgess*, 2004. Citado na página 28.

BECK, K. et al. Manifesto for agile software development. 2001. Citado na página 31.

BERNER, S.; WEBER, R.; KELLER, R. K. Observations and lessons learned from automated testing. In: *Proceedings of the 27th international conference on Software engineering*. [S.l.: s.n.], 2005. p. 571–579. Citado na página 32.

COHN, M. *Succeeding with agile: software development using Scrum*. [S.l.]: Pearson Education, 2010. Citado na página 28.

COLLOFELLO, J. S. Teaching practical software maintenance skills in a software engineering course. Association for Computing Machinery, v. 21, n. 1, p. 182–184, fev. 1989. ISSN 0097-8418. Disponível em: <https://doi.org/10.1145/65294.71211>. Citado na página 31.

COORE, D.; FOKUM, D. Facilitating course assessment with a competitive programming platform. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. [S.l.: s.n.], 2019. p. 449–455. Citado na página 20.

FOWLER, M.; FOEMMEL, M. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, v. 122, n. 14, p. 1–7, 2006. Citado na página 32.

GONZALEZ-ESCRIBANO, A. et al. Toward improving collaborative behaviour during competitive programming assignments. In: IEEE. *2019 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*. [S.l.], 2019. p. 68–74. Citado na página 20.

HALIM, S. et al. *Competitive programming 3*. [S.l.]: Citeseer, 2013. Citado 2 vezes nas páginas 15 e 19.

HOXMEIER, J. A.; DICESARE, C. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, p. 347, 2000. Citado na página 45.

LAAKSONEN, A. *Competitive Programmer's Handbook*. 2018. Disponível em: <https://cses.fi/book/book.pdf>. Citado na página 19.

LAWRENCE, R. Teaching data structures using competitive games. *IEEE Transactions on Education*, IEEE, v. 47, n. 4, p. 459–466, 2004. Citado na página 20.

LIENTZ, B. P.; SWANSON, E. B. *Software maintenance management: A study of the maintenance of computer application software in 487 data processing organizations*. [S.l.]: Addison-Wesley, 1980. Citado na página 32.

MOE, M. M. Comparative study of test-driven development (tdd), behavior-driven development (bdd) and acceptance test–driven development (atdd). *International Journal of Trend in Scientific Research and Development*, p. 231–234, 2019. Citado na página 32.

RIBEIRO, P.; GUERREIRO, P. Early introduction of competitive programming. *Olympiads in Informatics*, v. 2, p. 149–162, 2008. Citado na página 20.

RIGBY, P. C.; BIRD, C. Convergent contemporary software peer review practices. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2013. (ESEC/FSE 2013), p. 202–212. ISBN 9781450322379. Disponível em: <https://doi.org/10.1145/2491411.2491444>. Citado na página 32.

SUTTER, H.; ALEXANDRESCU, A. *C++ coding standards: 101 rules, guidelines, and best practices*. [S.l.]: Pearson Education, 2004. Citado na página 39.

SWANSON, E. B.; CHAPIN, N. Interview with e. burton swanson. *Journal of Software Maintenance: Research and Practice*, Wiley Online Library, v. 7, n. 5, p. 303–315, 1995. Citado na página 32.

VERNON, V. *Implementing domain-driven design*. [S.l.]: Addison-Wesley, 2013. Citado na página 31.

WASIK, S. et al. A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 51, n. 1, p. 1–34, 2018. Citado 3 vezes nas páginas 15, 20 e 23.