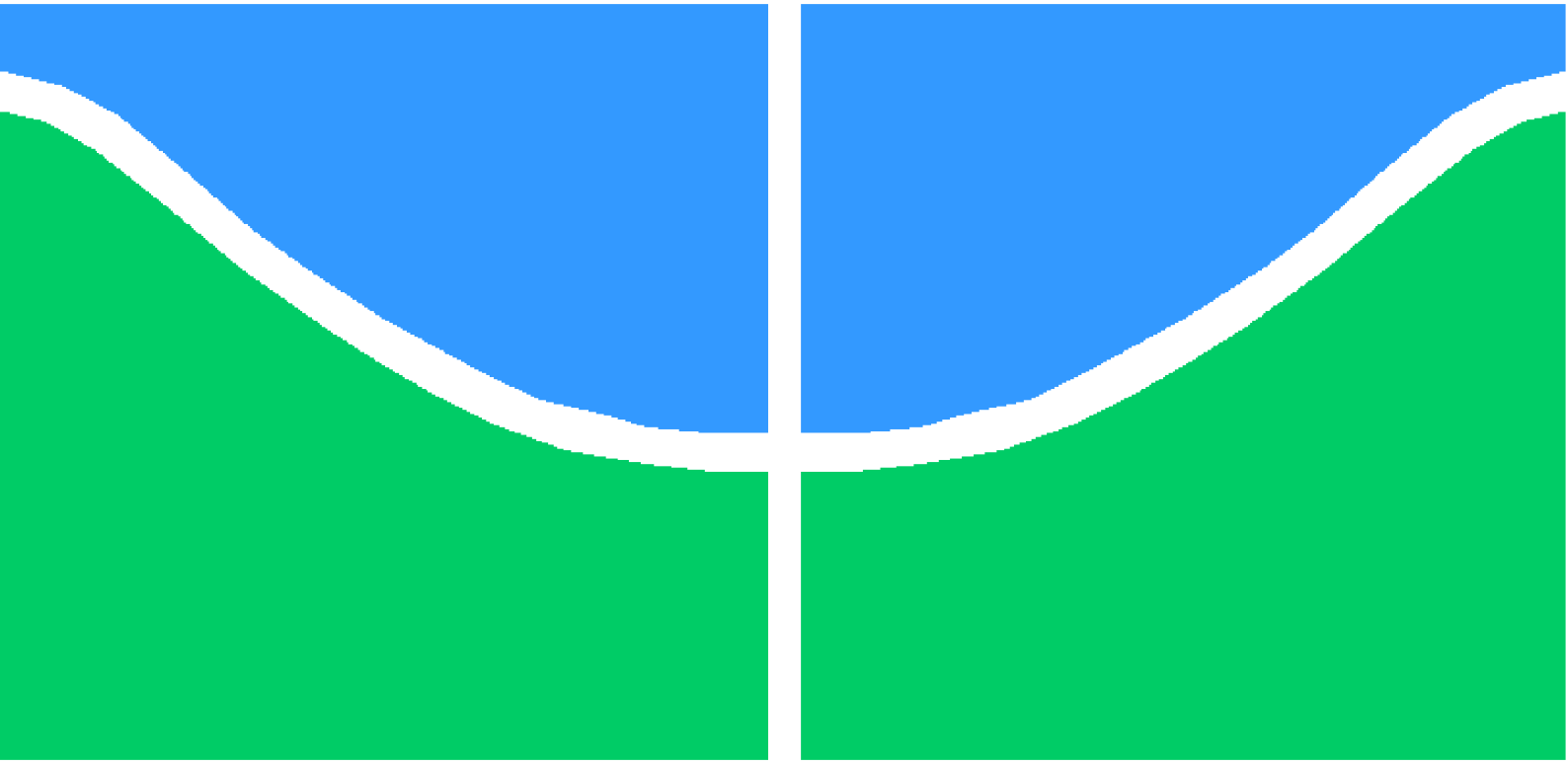

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Curso Engenharia de Software

Análise da interface de linha de comando da ferramenta *cp-tools*: um estudo de caso

Autor: Durval Carvalho de Souza

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2022

Durval Carvalho de Souza

Análise da interface de linha de comando da ferramenta *cp-tools*: um estudo de caso

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

2022

Durval Carvalho de Souza
Análise da interface de linha de comando da ferramenta *cp-tools*: um estudo de caso/ Durval Carvalho de Souza. – Brasília, DF, 2022-
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2022.

1. CLI. 2. boas práticas. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Análise da interface de linha de comando da ferramenta *cp-tools*: um estudo de caso

CDU 02:141:005.6

Durval Carvalho de Souza

Análise da interface de linha de comando da ferramenta *cp-tools*: um estudo de caso

Monografia submetida como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software da Faculdade UnB Gama - FGA, da Universidade de Brasília, em 06 de abril de 2022 apresentada e aprovada pela banca examinadora abaixo assinada:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Daniel Saad Nogueira Nunes
Membro convidado

Prof. Dr. John Lenon Cardoso Gardenghi
Membro convidado

Brasília, DF
2022

"Concentre-se nos pontos fortes, reconheça as fraquezas, agarre as oportunidades e proteja-se contra as ameaças."
(Sun Tzu)

Resumo

Após a popularização das interfaces gráficas, as interfaces de linha de comando (CLIs) deixaram de ser a principal maneira de interagir com computadores. Porém, mesmo com sua popularidade reduzida, softwares que possuem CLIs ainda são usados e continuam relevantes, principalmente para desenvolvedores e administradores de sistemas. Dessa forma, é importante seguir algumas boas práticas e recomendações durante o seu desenvolvimento, para que assim, potencialize suas chances de sucesso. Diante disso, este trabalho consistiu na análise do CLI do software *cp-tools*, visando identificar e corrigir os principais problemas encontrados em sua interface de linha de comando.

Palavras-chaves: CLI. *cp-tool*. boas práticas. interface de linha de comando.

Abstract

After the popularization of graphical interfaces, command line interfaces (CLIs) are no longer the main way of interacting with computers. However, even with its reduced popularity, software that has CLIs is still used and remains relevant, mainly for developers and system administrators. In this way, it is important to follow some good practices and recommendations during its development, in order to maximize the chances of success of the software. Therefore, this work consisted in the elaboration of the CLI analysis of the *cp-tools* tool, aiming to identify and correct the main problems found in its command line interface.

Key-words: CLI. *cp-tool*. best practices. command line interface.

Sumário

Introdução

Antes do surgimento das interfaces gráficas, para realizar qualquer atividade utilizando um computador era preciso interagir com as interfaces de linha de comandos (*Command Line Interfaces - CLIs*) (BLUM; BRESNAHAN, 2015). Durante esse período, a obtenção de ajuda na utilização ou na operação de um software era precária, pois a maioria dos usuários não tinham acesso à internet, e os que tinham dificilmente encontravam ajuda, uma vez que não existiam plataformas web como conhecemos hoje, por exemplo, o StackOverFlow[1], o GNU Org[2] e o Microsoft Docs[3] (PRASAD et al., 2018).

Diante dessa dificuldade de utilizar os computadores e suas aplicações, muitas experimentações foram realizadas, e algumas características se destacaram. Dessa forma, os desenvolvedores, visando maximizar o sucesso de sua próxima ferramenta, começaram a reproduzir as características com maior aceitação por parte dos usuários, tais como: o modo de receber argumentos e parâmetros; a forma de tratar sinais e caracteres de controle; o modo de fornecer a documentação e licença de uso do software; e outras características que se provaram bem-sucedidas. Diante disso, algumas iniciativas agruparam tais características, surgindo vários guias e manuais de boas práticas que buscavam ajudar os desenvolvedores a criarem softwares de melhor qualidade (PRASAD et al., 2018).

A popularização das interfaces gráficas de usuários (*Graphical User Interface* - GUI) teve como consequência a diminuição do uso dos programas de interface de linha de comandos (BLUM; BRESNAHAN, 2015). Para a maioria dos usuários é mais fácil aprender a utilizar um software por meio da visualização de elementos como botões, caixas de texto, imagens e vídeos, em vez da leitura e digitação de comandos em um terminal[4]. Porém, os programas com interfaces de linha de comando ainda são relevantes, possuindo diferenciais que dificilmente são implementados em programas de interfaces gráficas. Alguns exemplos de diferenciais são: o baixo requisito de memória, uma vez que não há necessidade de renderização; a capacidade de automação e de geração de scripts, uma vez que as sequências de comandos CLIs podem ser escritas em ordem para executar uma determinada tarefa; a capacidade de agendamento de execução de scripts; a capacidade de operação em conjunto, uma vez que comandos CLIs podem ser usados em cadeia, onde o resultado de um determinado comando pode ser utilizado como parâmetro do próximo comando, criando assim inúmeras possibilidades de combinações de tais ferramentas

[1] Site de perguntas e respostas para profissionais e entusiastas na área de software (<https://stackoverflow.com/>).

[2] Site que agrupa os vários manuais desenvolvido pela organização GNU (<https://www.gnu.org/org/>).

[3] Site que agrupa documentação técnica para usuários finais e profissionais de TI que trabalham com produtos da Microsoft (<https://docs.microsoft.com/>).

[4] Uma explicação detalhada sobre os terminais é dada no Apêndice A.

(PRASAD et al., 2018).

Sendo assim, o desenvolvimento de ferramentas de linha de comando continuam relevantes para o cenário de software, sendo os desenvolvedores e os administradores de sistemas os dois principais perfis de usuários que utilizam tais ferramentas diariamente. Um caso de uma nova ferramenta que possui um CLI é o software *cp-tools*, criado pelo Prof. Dr. Edson Alves da Costa Júnior, com o objetivo de facilitar o desenvolvimento e a portabilidade de questões de programação para plataformas de juízes online.

Esse CLI surgiu no ambiente acadêmico, no cenário de maratonas de programação. Os professores dos alunos interessados em programação competitiva precisam preparar cenários de competição com diversos problemas, de preferência inéditos. Porém, a formatação e empacotamento de problemas não é uma tarefa trivial, sendo necessário conhecer as particularidades de cada plataforma de juíz eletrônico. Essa falta de padronização cria uma barreira na colaboração entre professores, além de dificultar a portabilidade de uma questão de programação de uma plataforma para outra. Diante disso, o software *cp-tools* foi criado com o objetivo de unificar a formatação dos problemas e realizar o empacotamento de maneira automática, partindo da estrutura base proposta pelo *cp-tools*. Explicar os benefícios da utilização de programação competitiva na formação de estudantes de software foge do escopo deste estudo. Diante disso, para um maior detalhamento sobre os desafios na formação de estudantes de software e os benefícios na utilização de maratonas de programação para fins educacionais consulte o Apêndice B.

Nesse contexto, a proposta deste trabalho é analisar a ferramenta de formatação de problemas *cp-tools*, com a finalidade de encontrar as boas e más práticas sob a ótica do desenvolvimento de interfaces de linha de comando (CLIs), além de realizar uma priorização e correção das inconformidades.

Descrição do problema

A ferramenta *cp-tools* é um software que provê funcionalidades relacionadas à criação, importação e exportação de problemas de juízes eletrônicos. Atualmente, as funcionalidades do software estão disponíveis somente por meio da sua interface de linha de comando. Desse modo, buscando reduzir a curva de aprendizado e assim tornar a ferramenta mais acessível, será analisada a sua interface de linha de comando com a finalidade de visibilizar as convenções que foram ou não adotadas, assim como as boas e más práticas presentes no software.

Justificativa

Uma vez que a ferramenta *cp-tools* possui uma CLI é importante que ela siga uma série de convenções e boas práticas, pois dessa maneira o software terá maior aceitação pelos seus usuários. Isso posto, este trabalho é importante pois trará visibilidade para problemas existentes na interface da ferramenta, além de priorizar e corrigir os problemas mais impactantes.

Objetivos

O objetivo geral deste trabalho é analisar a interface de linha de comando da ferramenta *cp-tools*, com o intuito de dar visibilidade às boas e más práticas presentes que podem favorecer ou prejudicar a adoção da ferramenta.

A fim de atingir o objetivo geral deste trabalho, foram estabelecidos os seguintes objetivos específicos:

- conhecer as principais recomendações e boas práticas de desenvolvimento de CLI aplicáveis ao software *cp-tools*;
- analisar a CLI do software *cp-tools* visando identificar quais boas práticas e recomendações estão presente e quais não estão;
- realizar modificações no software *cp-tools* buscando corrigir as inconformidades encontradas;

Estrutura do Trabalho

Este trabalho está organizado em 4 capítulos. O primeiro capítulo, Fundamentação Teórica, apresenta todos os termos e conceitos necessários para compreensão do estudo de caso. O segundo capítulo, Metodologia, explica como o estudo de caso foi concebido e realizado, apresentando como foi realizado o levantamento do referencial teórico, a metodologia utilizada e o desenho do estudo de caso. O terceiro capítulo, Resultados, é explicado cada um dos artefatos gerados durante a elaboração do estudo de caso, além das ações corretivas feitas no código fonte do software para corrigir as inconformidades encontradas. Por fim, no último capítulo, Conclusão, é discutindo os objetivos que foram alcançados e os potenciais futuros trabalhos.

1 Fundamentação Teórica

Neste capítulo serão apresentados os conceitos relacionados às interfaces de linha de comando e ao estudo de caso que foi elaborado. Na primeira seção, Interfaces e Interação Humano-Computador, é introduzida a área do conhecimento que estuda as interações com os computadores, assim como as principais abordagens de avaliação de interfaces. Na segunda seção, Algoritmo Distância de Levenshtein, é explicado o algoritmo utilizado em uma das ações corretivas realizadas no software do estudo caso. E na última seção, Guias, manuais e recomendações de desenvolvimento de CLIs, é apresentado os principais documentos que abordam recomendações de desenvolvimento de softwares com interfaces de linha de comando.

1.1 Interfaces e Interação Humano-Computador

A busca por interfaces de fácil utilização sempre foi uma preocupação humana. O resultado dessa busca foi a constante melhoria das várias interfaces presentes no nosso dia a dia, visando a criação de produtos mais intuitivos e mais eficientes. No campo da computação não foi diferente, o estudo sobre interfaces é realizado há muitos anos, com o objetivo de facilitar a interação com sistemas computacionais, sendo a área do conhecimento denominada Interação Humano-Computador a mais conhecidas delas (NORMAN, 2002).

De acordo com Norman (2002), o aumento na qualidade de uso de sistemas interativos teve como consequências vários benefícios, diretos e indiretos. Alguns desses são:

- o aumento da produtividade dos usuários, visto que, se a interação for eficiente, os objetivos almejados serão obtidos mais rapidamente;
- a redução do número e da gravidade das falhas cometidas pelos usuários, uma vez que eles poderão prever as consequências de suas ações e somente executar ações perigosas quando estiverem certos disso;
- a redução do tempo de aprendizado, pois a interface intuitiva seguirá padrões já estabelecidos, além de ser capaz de ensinar a utilização a novos usuários durante o uso;
- o aumento da satisfação do usuário e consequentemente da popularidade do software, uma vez que usuários satisfeitos recomendam o sistema para mais pessoas.

A medição da qualidade de uso de uma interface pode ser aferida pela sua usabilidade. Considerando que a medição desse indicador é uma tarefa subjetiva, os pesquisadores Rogers, Sharp e Preece (2011) afirmam que a usabilidade de uma interface pode ser aferida por meio da medição de outros fatores, sendo os principais deles:

- facilidade de aprendizado: refere-se ao tempo e ao esforço necessário para aprender a utilizar uma determinada funcionalidade ou determinado sistema;
- facilidade de uso: refere-se ao esforço cognitivo para interagir com o sistema, descobrir novas funcionalidades e contornar erros durante a interação;
- eficácia de uso: refere-se à análise da execução do sistema, se o sistema faz bem aquilo que se propõe fazer;
- eficiência de uso: refere-se a análise da realização do objetivo do usuário, se esta ocorre de forma rápida e com nenhum ou com o mínimo de erros;
- satisfação do usuário: refere-se à avaliação subjetiva do usuário, durante e após a interação;
- flexibilidade: refere-se à capacidade do sistema de acomodar as diferentes abordagens que os usuários podem adotar durante a utilização do sistema;
- utilidade: refere-se à quantidade de funcionalidades que o sistema disponibiliza para os seus usuários;
- segurança de uso: refere-se à capacidade do sistema proteger os usuários e seus artefatos contra condições não intencionais.

1.1.1 Avaliação e técnicas de avaliação de usabilidade

Uma vez compreendida a definição de usabilidade no contexto de software, o primeiro passo para melhorar uma interface é analisar e avaliar a interface atual. De acordo com Barbosa e Silva (2010), a avaliação é a atividade fundamental em qualquer processo de desenvolvimento e melhoria de interfaces, pois é por meio dela que o avaliador irá fazer o julgamento de valor sobre a qualidade de uso da interface, além de identificar os potenciais problemas existentes.

De acordo com Cybis (2003), é possível utilizar três tipos de técnicas na avaliação da usabilidade de um software: técnicas preditivas, técnicas empíricas e técnicas prospectivas.

As técnicas preditivas não dependem da participação de usuários durante a avaliação. Essa técnica é baseada em verificações e inspeções, realizadas geralmente por especia-

listas em usabilidade. As avaliações que usam esse tipo de técnica podem ser classificadas como avaliações analísticas, avaliações heurísticas e inspeções por *checklist*.

Já as técnicas empíricas são aquelas que buscam constatar um problema a partir da observação do usuário utilizando o sistema. Normalmente as avaliações que utilizam essa técnica são realizadas em um ambiente controlado, onde os usuários são convidados a utilizar um sistema e são monitorados, visando identificar situações onde há falhas na usabilidade.

Por fim, as técnicas prospectivas buscam conhecer o grau de satisfação ou insatisfação de usuários que já utilizaram um determinado sistema interativo. As avaliações que utilizam essa técnica geralmente utilizam questionários e entrevistas.

A inspeção por *checklist* é um tipo de avaliação que utiliza a técnica preditiva, ou seja, não depende do uso prévio do software por parte dos usuários. O objetivo da avaliação é a identificação rápida de um conjunto de potenciais problemas que podem existir em uma interface. Uma inspeção por *checklist* pode ser dividida em duas partes: a elaboração do *checklist* e a inspeção guiada pelo *checklist*. A primeira parte geralmente é realizada por especialistas em usabilidade que definem um conjunto de características ou situações que devem ser procuradas na interface de um sistema. Já a segunda parte, a inspeção, normalmente é realizada por mais de um avaliador visando reduzir potenciais falhas de julgamento no momento da avaliação.

1.2 Guias, manuais e recomendações de desenvolvimento de CLIs

Antes do surgimento das interfaces gráficas, a principal maneira de interagir com sistemas computacionais era por meio de uma interface de linha de comando (*Command Line Interface* - CLI) disponibilizada pelo terminal do sistema. Esse terminal, na década 50 e 60, era um dispositivo físico integrado a um computador *mainframe*[1]. Com a revolução dos computadores pessoais, esse terminal passou a ser um hardware emulado em um software comumente chamado *shell*[2]. A explicação detalhada da evolução das interfaces dos sistemas computacionais foge do objetivo deste trabalho, sendo brevemente explanada no Apêndice A.

Após a popularização das interfaces gráficas de usuários (*Graphical User Interface* - GUI), a maioria dos usuários deixaram de utilizar as interfaces de linha de comando. No entanto, elas continuam relevantes, principalmente para desenvolvedores de softwares e administradores de sistemas (PRASAD et al., 2018). Esses dois perfis de usuários utilizam as CLIs de diversos programas buscando principalmente a automatização de tarefas e a

[1] Termo utilizado para referir ao gabinete principal que alojava a unidade central de processamento nos primeiros computadores

[2] Software interpretador interativo de comandos

integração de várias ferramentas por meio de suas CLIs.

Porém, antes do surgimento das interfaces gráficas vários softwares com CLIs foram construídos. Nesse cenário, os primeiros desenvolvedores implementavam suas interfaces com bastante experimentação, criando suas próprias rotinas de obtenção de parâmetros, seus próprios tratamentos de exceções e outras características comuns de toda interface de linha de comando. A consequência dessa falta de padronização foi a dificuldade no aprendizado dessas várias ferramentas e a incompatibilidade do uso conjunto para realização de uma determinada tarefa.

Diante desse cenário, várias iniciativas surgiram visando melhorar a usabilidade dos usuários e padronizar comportamentos comuns. Algumas das iniciativas que se destacaram foram o *Standards for Command Line Interfaces - GNU Coding Standards* (Free Software Foundation, 2021), *POSIX Utility Conventions* (IEEE and The Open Group, 2021), *IBM Command-line interface conventions* (IBM, 2021) e o *Command Line Interface Guidelines* (PRASAD et al., 2018).

1.2.1 *Standards for Command Line Interfaces - GNU Coding Standards*

Os padrões de codificação GNU (*GNU Coding Standards*) foram escritos por Richard Stallman e outros voluntários do projeto GNU[3]. O objetivo desse documento é tornar o sistema GNU consistente, fácil e compatível. Esse documento pode ser lido como um guia para escrever programas portáveis e confiáveis. O foco principal do guia está em programas escritos na linguagem C, mas muitos dos princípios e recomendações são úteis em qualquer linguagem de programação.

Esse guia abrange vários aspectos do ciclo de vida de um software, desde seu licenciamento, passando pelo design de interfaces e finalizando no processo de lançamento (*release*). Referente à seção de padrões para interfaces de linha de comando, o guia apresenta algumas diretrizes, sendo algumas delas:

- utilize a funcão `getopt_long()` para tratar opções de linha de comando;
- todos os CLIs devem suportar as opções `--version` e `--help`;
- a opção `--help` deve gerar uma breve documentação de como utilizar o programa;
- caso a opção `--help` esteja presente, nenhuma outra ação deve ser executada;
- ao término da mensagem de ajuda da opção `--help`, disponibilize um canal de comunicação onde os usuários poderão enviar *feedbacks* e relatos de *bugs*;

[3] O projeto GNU é uma iniciativa liderada por Richard Stallman cujo objetivo é dar aos usuário de software liberdade e controle no uso dos seus computadores e aplicações.

- a opção `--version` deve apresentar informações referente ao nome do programa, versão, status legal e autores.

1.2.2 *POSIX Utility Conventions - POSIX.1-2017*

O *POSIX.1-2017*, também conhecido como *IEEE Std 1003.1-2017*, é uma especificação de sistema que define características necessárias para que um sistema operacional seja da família POSIX. Algumas dessas características são:

- a presença de um conjunto de programas utilitários (`cat`, `ls`, `mv`, `rm`, dentre outros);
- a presença de ao menos um interpretador de comando (`bash`, `fish`, `zsh`, `sh`, dentre outros);
- a presença de um subconjunto de chamadas de sistemas com uma determinada interface.

Como é de se esperar de uma especificação de sistema, o *POSIX.1-2017* possui um longo escopo que envolve desde terminologias até definição de interfaces de chamadas de sistemas. Essa especificação possui um capítulo referente ao funcionamento de terminais em sistemas POSIX, a Seção 11 da especificação, *General Terminal Interface*[4], que define detalhadamente o comportamento esperado de um terminal, apresentando quais rotinas devem ser executadas em diferentes cenários de execução de um programa via terminal de comandos. A explicação do comportamento dos terminais foge do escopo deste trabalho. O Apêndice A traz um breve histórico da evolução das CLIs.

Referente às definições aplicáveis às interfaces de linha de comando, a especificação *POSIX.1-2017* apresenta a seção 12, *Utility Conventions*[5], que expõe a forma como as ferramentas utilitárias devem tratar seus argumentos. Nessa seção é apresentado pela primeira vez a notação comumente utilizada por diversos programas em suas CLIs. Essa notação pode ser vista no Código 1.

```
utility_name [-a][-b][-c option_argument] [-d|-e][-f[option_argument]][operand...]
```

Código 1: notação de tratamento de argumentos definidos pela especificação *POSIX.1-2017*.

A notação definida pela *POSIX.1-2017* determina que o nome do utilitário deve ser seguido primeiramente pelas opções. Para as opções que recebem argumentos o argumento

[4] Seção 11 do POSIX.1-2017 - (<https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap11.html>)

[5] Seção 12 do POSIX.1-2017 - (<https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html>)

deve ser apresentado logo em seguida da respectiva opção. E por fim, no final do comando deve ser apresentado os operandos, caso aplicável.

Após a definição da forma como os utilitários de sistemas POSIX devem receber e tratar seus argumentos e opções, a especificação apresenta algumas recomendações de nomenclatura, sendo algumas delas apresentadas a seguir:

- o nome dos utilitários devem ter entre dois a noves caracteres;
- o nome dos utilitários devem incluir somente letras minúsculas e dígitos do conjunto de caracteres portátil[6];
- todas as opções devem ser precedidas pelo carácter delimitador "-";
- quando uma única opção aceita vários argumentos de opção, tais argumentos devem ser separados pelo carácter vírgula (",") e não pelo caractere espaço (" ");
- a ordem relativa das opções não deve importar.

1.2.3 *Command Line Interface Guidelines*

O guia denominado *Command Line Interface Guidelines*[7], publicado em 2018, foi uma iniciativa de Aanand Prasad e Ben Firshman, ambos ex-colaboradores do software Docker Compose[8]. Esse guia agrupa diversas boas práticas e recomendações presentes em vários outros guias e manuais, além de relatar aprendizados obtidos durante o desenvolvimento da ferramenta *docker-compose*. No total são apresentadas 99 recomendações e boas práticas, exemplificadas e categorizadas em um dos seguintes 15 grupos:

- *the basics* - o básico;
- *help* - ajuda;
- *documentation* - documentação;
- *output* - saída;
- *errors* - erros;
- *arguments and flags* - argumentos e opções;
- *interactivity* - interatividade;

[6] Conjunto de caracteres portátil (*Portable Character Set*) é um conjunto de 103 caracteres que, de acordo com o padrão POSIX.1-2017, deve estar presente em qualquer conjunto de carácter (<https://en.wikipedia.org/wiki/Portable_character_set>).

[7] Site que agrupa diversas recomendações para o desenvolvimento de CLIs (<https://clig.dev/>).

[8] O Docker Compose é uma ferramenta para definir e executar aplicativos Docker de vários contêineres (<https://docs.docker.com/compose/>).

- *subcommands* - subcomandos;
- *robustness* - robustez;
- *future-proofing* - capacidade de evolução;
- *signals and control characters* - sinais e caracteres de controle;
- *configurations* - configuração;
- *environment variables* - variáveis de ambiente;
- *distribution* - distribuição;
- *analytics* - análise.

Dentre os demais guias apresentados, este é o que abrange o maior número de boas práticas e recomendações para o desenvolvimento de CLIs, além de apresentar diversas situações de exemplo de outros softwares que respeita tal recomendação.

1.3 Algoritmo Distância de Levenshtein

A Distância de Levenshtein foi um algoritmo criado em 1966 pelo cientista russo *Vadimir Levenshtein*, cujo propósito é calcular a quantidade mínima de operações para transformar uma string em outra, onde as operações permitidas são operações de substituição, remoção e inserção de caracteres.

O cálculo da distância de Levenshtein é baseada na observação de que, se reservarmos uma matriz para manter a distância de Levenshtein entre todos os prefixos de ambas as strings, será possível calcular os valores na matriz utilizando programação dinâmica, e assim calcular a distância das strings com a complexidade temporal de $O(mn)$, onde m e n são o tamanho das strings.

Para compreender o funcionamento deste algoritmo suponha que queremos saber a distância de Levenshtein das strings “asas” e “aguas”. A fórmula matemática que define o algoritmo é apresentada abaixo:

$$levenshtein_{a,b}(i,j) = \begin{cases} \max(i,j) & se \min(i,j) = 0 \\ min \begin{cases} levenshtein_{a,b}(i-1,j) + 1 \\ levenshtein_{a,b}(i,j-1) + 1 \\ levenshtein_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{caso contrário.} \end{cases}$$

Tabela 1 – Tabela inicial utilizada pelo algoritmo de Levenshtein

			a	g	u	a	s
		0	1	2	3	4	5
	0	0					
a	1						
s	2						
a	3						
s	4						

Tabela 2 – Tabela parcialmente completa utilizada pelo algoritmo de Levenshtein

			a	g	u	a	s
		0	1	2	3	4	5
	0	0	1	2	3	4	5
a	1	1					
s	2	2					
a	3	3					
s	4	4					

Na fórmula apresentada acima, a e b são as substrings comparadas, e i e j é a posição que delimita a substrings. Desta forma, para as strings “asas” e “aguas”, com a indexação começando em 1, quando dizemos $levenshtein_{a,b}(3,4)$ estamos calculando a distância de Levenshtein das substrings “asa” e “agua”, ou seja, os 3 primeiros caracteres da string a e os 4 primeiros caracteres da string b.

Nesta fórmula, a primeira parte da equação diz que quando uma das substrings for vazia, isto é, quando o seu delimitador é igual a zero, a distância de Levenshtein será a quantidade de caracteres da outra substring. Caso contrário, será preciso computar se será mais vantajoso adicionar, alterar ou remover um caractere de uma das substrings comparadas, isto levando em consideração os cálculos feitos anteriormente, sendo este o motivo da minimização utilizando os delimitadores com $(i-1,j)$, $(i,j-1)$ e $(i-1,j-1)$.

Para facilitar a compreensão de como a tabela utilizada pelo algoritmo é preenchida, foi ilustrado duas colunas e duas linhas adicionais a tabela, para visualizar o índice acessado e qual a substring que está sendo comparada, respectivamente. Conforme pode ser visto na Tabela 1, inicialmente sabemos que a distância entre duas substrings vazias é zero.

Quando o i e o j são incrementados são realizados sucessivas comparações entre as substrings. Quando comparado uma substring não vazia com uma substrings vazia, representado pela terceira linha da Tabela 2, a distância é simplesmente a quantidade de caracteres da substrings não vazia, pois para torná-las iguais basta realizar inserções na substrings vazia.

Tabela 3 – Tabela parcialmente completa utilizada pelo algoritmo de Levenshtein

			a	g	u	a	s
		0	1	2	3	4	5
	0	0	1	2	3	4	5
a	1	1	0	1	2	3	4
s	2	2	1	1	2	3	4
a	3	3	2	2	2	2	3
s	4	4	3	3	3	3	2

Quando ambas as substrings possuem caracteres é comparado qual das operações, sendo elas inserção, alteração e remoção, apresentará o menor custo. Uma vez que já foi computado a distância das substrings anteriores, basta somar 1 caso o caractere comparado seja diferente. Deste modo, para a $levenshtein_{a,b}(1,1)$ as substrings "a" e "a" serão comparadas, para isso será consultado na tabela o valor já calculado de $levenshtein_{a,b}(i-1,j)+1$, $levenshtein_{a,b}(i,j-1)+1$ e $levenshtein_{a,b}(i-1,j-1)+1$, com a diferença que este último somente será acrescentado 1 caso o caractere na posição i e j sejam diferentes. Deste modo, completando a tabela seguindo o raciocínio apresentado acima, iremos obter a Tabela 3.

Uma implementação do algoritmo Distância de Levenshtein na linguagem C++ é apresentada no Código 2.

```cpp
size_t levenshtein_distance(const std::string& s, const std::string& t) {
    size_t n = s.length() + 1;
    size_t m = t.length() + 1;

    std::vector<size_t> d(m*n, 0);

    for(size_t i=1, im=0; i<m; ++i, ++im)
    {
        for(size_t j=1, jn=0; j<n; ++j, ++jn)
        {
            if(s[jn] == t[im])
            {
                d[ (i*n)+j ] = d[((i-1)*n) + (j-1)];
            }
            else
            {
                d[(i * n) + j] = std::min({
                    d[(i - 1) * n + j] + 1,       // deletion
                    d[i * n + (j - 1)] + 1,       // insertion
                    d[(i - 1) * n + (j - 1)] + 1 // substitution
                });
            }
        }
    }

    size_t result = d[n * m - 1];

    return result;
}
```

Código 2: Implementação do algoritmo Distância de Levenshtein utilizada pelo software *cp-tools*

2 Metodologia

Este capítulo tem como objetivo apresentar os aspectos referentes à metodologia adotada neste estudo de caso. A Seção 2.1 apresenta como foi feito o levantamento do referencial teórico que orientou a elaboração deste trabalho. Já a Seção 2.2 apresenta quais ferramentas foram utilizadas durante a elaboração deste trabalho, assim como a metodologia de desenvolvimento utilizada. Por fim, a Seção 2.3 detalha quais foram as etapas realizadas durante toda a avaliação da interface de linha de comandos.

2.1 Levantamento do referencial teórico

Dado o objetivo deste trabalho, ou seja, analisar se a ferramenta *cp-tools* segue as principais boas práticas e recomendações de desenvolvimento de CLIs, foi necessário realizar um levantamento de referencial teórico, visando conhecer os principais guias, manuais e documentos que descrevem tais boas práticas e recomendações.

A busca foi feita utilizando os buscadores *Scholar Google*, *World Wide Science Org*, *Scielo*, *Biblioteca Digital de Teses e Dissertações da USP*, *Biblioteca Digital Brasileira de Teses e Dissertações* e *Google*. Nestes buscadores foram feitas buscas utilizando os termos *command line interface*, *command line interface best practices*, *command line interface recommendations*, *CLI best practices* e *CLI recommendations*.

A maioria das obras encontradas não tinham foco no desenvolvimento de interfaces de linha de comando. Foram encontradas obras que apresentavam um determinado software que possuía um CLI, mas sem apresentar o processo de desenvolvimento de sua interface de linha de comando. Também foram encontradas obras que se propunham a instruir os leitores a utilizar um determinado software, apresentando as funcionalidades e as possibilidades de sua CLI, mas sem o foco no processo e nas boas práticas implementadas. Deste modo, excluindo estas obras que não focam no processo de desenvolvimento de CLIs foram encontradas as obras que estão listadas na Tabela 4.

2.2 Materiais e métodos

Para a avaliação e codificação do software *cp-tools* foi imprescindível a sua compilação e instalação local. Para isto foi necessário a instalação de certas dependências, sendo elas: o pacote LaTex `pdflatex`, o compilador `GCC` na versão 10.3 e o kit de ferramentas `libssl-dev` na versão 1.1.1. O sistema operacional utilizado foi o `Ubuntu` na versão 18.04.6 TLS. As características do notebook utilizado para avaliar e codificar a

Tabela 4 – Referencial teórico encontrado durante a fase de pesquisa.

Tipo de obra	**Título da obra**	**Autores**
Artigo	*"Usability improvements for products that mandate use of command-line interface: Best Practices"*	Samrat Dutta
Artigo	*"Bionitio: demonstrating and facilitating best practices for bioinformatics command-line software"*	Peter Georgeson et al.
Livro	*"PHP CLI: Create Command Line Interface Scripts with PHP."*	Rob Aley
Livro	*"The CLI Book: Writing Successful Command Line Interfaces with Node.js."*	Robert Kowalski
Livro	*"Command-Line Rust."*	Ken Youens-Clark
Livro	*"Linux Command Line and Shell Scripting Bible."*	Richard Blum, Christine Bresnahan
Página web	*"Best Practices Building a CLI Tool for Your Service"*	Adam DuVander
Página web	*"How To Make Your CLI More Intuitive"*	Kristopher Sandoval
Página web	*"Command Line Interface Guidelines"*	Aanand Prasad e Ben Firshman
Página web	*"Standards for Command Line Interfaces - GNU Coding Standards"*	
Página web	*"POSIX Utility Conventions"*	

Tabela 5 – Características do notebook utilizado para avaliar e codificar o software *cp-tools*.

Modelo	HP Pavilion 14-V067BR
Processador	Intel Core i5-4210U CPU @ 1.70GHz × 4
Placa de Vídeo	NVIDIA GeForce 830M/PCIe/SSE2
Disco	SSD Crucial MX500 de 500GB
Memória	12GB DDR3 SODIMM (8GB + 4GB)

ferramenta são apresentadas na Tabela 5.

Durante o correção das inconformidades encontradas no software, foram realizados encontros semanais com o mantenedor do projeto, Prof. Dr. Edson Alves da Costa Júnior, visando revisar as mudanças implementadas e realizar potenciais melhorias. Durante estes encontros cuja duração variava entre 1 e 2 horas, eram revisados os avanços feitos durante a semana e com o restante do tempo eram realizados pareamentos de desenvolvimento.

2.3 Desenho do estudo de caso

Para a realização deste estudo de caso foi escolhida a avaliação utilizando inspeção por *checklist*. Conforme é explicado na Seção 1.1.1, esta é uma técnica preditiva, ou seja, não depende do uso prévio por parte dos usuários do software.

Para elaborar o *checklist* foram primeiramente selecionadas boas práticas e recomendações de 3 principais fontes:

- *Standards for Command Line Interfaces - GNU Coding Standards* (Free Software Foundation, 2021);
- *POSIX Utility Conventions* (IEEE and The Open Group, 2021);
- *Command Line Interface Guidelines* (PRASAD et al., 2018).

Utilizando tais guias e manuais foi selecionado um subconjunto de recomendações e boas práticas que eram aplicáveis ao contexto do software *cp-tools*. Esse subconjunto foi organizado na forma de manual, onde é explicada a essência da recomendação, assim como exemplo de aplicações. Esse manual está disponível no Apêndice C.

Com base nos tópicos do manual criado, foi elaborado um *checklist* com o objetivo de guiar a inspeção dos avaliadores da interface. Esse *checklist* está disponível no Apêndice D.

Após inspeção do avaliador na interface do software, foi elaborado um relatório que evidencia quais práticas foram ou não atendidas, assim como a devida justificativa. Esse relatório está disponível no Apêndice F.

Uma vez evidenciados os problemas na CLI do software, foi realizada uma priorização de quais problemas poderiam ser corrigidos, e foram feitas modificações no software, visando corrigir as inconformidades.

3 Resultados

Neste capítulo, serão mostrados todos os resultados obtidos durante a realização deste trabalho. O capítulo está organizado na sequência temporal das ações realizadas para avaliar a interface do software *cp-tools*. Deste modo, na Seção 3.1 são apresentados os referenciais teóricos utilizados para elaborar o "Manual de Recomendação de Boas Práticas para o Software *cp-tools*" utilizado durante a avaliação. Já na Seção 3.2 é apresentado o *checklist* criado para auxiliar na realização da inspeção. Na Seção 3.3 é apresentado o relatório da inspeção. Na Seção 3.4 são apresentadas as inconformidades encontradas e sua respectiva prioridade de correção. Por fim, na Seção 3.5 são apresentadas as ações corretivas realizadas para sanar tais inconformidades.

3.1 Manual de recomendações e boas práticas para o software *cp-tools*

Para definir o conjunto de boas práticas e recomendações que seriam analisadas durante a avaliação da interface de linha de comando do software *cp-tools*, foi necessário o estudo de diversos documento, guias e manuais, sendo os três de maior influência para este trabalho os seguintes documentos:

- *Standards for Command Line Interfaces - GNU Coding Standards* (Free Software Foundation, 2021);
- *POSIX Utility Conventions* (IEEE and The Open Group, 2021);
- *Command Line Interface Guidelines* (PRASAD et al., 2018).

Com base nestes documentos, foi criado um manual específico para o software *cp-tools*, onde são detalhadas e exemplificadas as recomendações e boas práticas selecionadas. Neste manual, também são apresentadas as consequências quando há inconformidades com as recomendações, buscando exemplificar tais situações. A estrutura deste documento consiste em 6 tópicos, onde cada tópico apresenta as recomendações de acordo com uma categoria de boa prática, sendo elas listadas a seguir:

- boas práticas gerais;
- boas práticas de ajuda ao usuário;
- boas práticas de tratamento de erros;

- boas práticas na utilização de argumentos e sinalizadores;
- boas práticas na interatividade;
- boas práticas no uso de subcomandos.

O manual de recomendações do software *cp-tools* está disponível no Apêndice C.

3.2 *Checklist*

Uma vez definido o conjunto de boas práticas e recomendações que serão avaliadas, foi criado o *checklist* que auxiliou durante a inspeção da interface. Este *checklist* foi criado com base nos tópicos do manual de boas práticas do *cp-tools*, visando facilitar a referência de qual boa prática foi ou não evidenciada. Além dos tópicos, o *checklist* possui duas colunas adicionais, as colunas "Análise" e "Comentário", cujas funções são marcar se determinado tópico foi ou não evidenciado, e anotar comentários feitos pelo avaliador durante a inspeção, respectivamente. O *checklist* criado está disponível no Apêndice D.

Após a realização da inspeção, o *checklist* foi preenchido por completo com as análises do avaliador, e assim foram evidenciadas quais boas práticas estavam ou não presentes na interface do software. O documento *checklist* preenchido está disponível no Apêndice E.

3.3 Relatório da avaliação da CLI da ferramenta *cp-tools*

Uma vez que o propósito do *checklist* era o de auxiliar o avaliador durante a inspeção, foi necessário a elaboração de um relatório detalhado com os problemas encontrados. Deste modo, foi criado um novo documento com a mesma estrutura do manual de boas práticas do software *cp-tools*, onde cada tópico apresenta as evidências da implementação ou da ausência de uma boa prática. A estrutura deste relatório foi propositalmente mantida igual ao do manual de boas práticas, visando assim facilitar a referência cruzadas entre os documentos. Este relatório está disponível no Apêndice F.

3.4 Inconformidades e priorização de correções

Após a realização do relatório, foi possível compreender quais casos de uso da ferramenta *cp-tools* que evidenciavam a ausência de uma boa prática, possibilitando a realização de ações de melhoria. Porém, antes de efetuar modificações no código fonte da ferramenta, foi elaborada uma priorização de quais inconformidades devem ser corrigidas primeiro. Esta priorização foi desenvolvida pelo principal mantenedor do software, o Prof. Dr. Edson Alves da Costa Júnior.

Tabela 6 – Priorização das boas práticas não estão presentes no software *cp-tools*

Priorização	Boas práticas não implementadas
1	Use uma biblioteca de análise de argumento de linha de comando
2	Se o usuário fez algo errado, tente adivinhar o que ele quis dizer
3	Disponibilize um texto de ajuda quando os sinalizadores `-h` e `--help` forem passados
4	Trate os erros e emita uma mensagem significativa
5	Tenha suporte a coringas *wildcards*
6	Não leia credenciais diretamente dos argumentos
7	Seja responsível

Das 25 boas práticas analisadas, 18 foram identificadas na interface da ferramenta, e 7 foram evidenciadas situações onde a boa prática poderia estar aplicada ou situações onde havia inconformidade com a recomendação. As recomendações não atendidas e sua respectiva priorização são apresentadas na Tabela 6.

3.5 Ações corretivas

3.5.1 Use uma biblioteca de análise de argumento de linha de comando

Para atender à esta recomendação foi desenvolvida uma biblioteca de análise de linha de argumentos específica para o software *cp-tools*. A decisão de não utilizar uma biblioteca existente se deve a um dos requisitos de projeto, que busca minimizar as dependências externas. Sendo assim, foi elaborado um *microframework* de desenvolvimento de ferramentas CLIs, que visa facilitar a evolução da ferramenta respeitando o manual de boas práticas e recomendações desenvolvidas para a ferramenta *cp-tools*.

3.5.1.1 Biblioteca desenvolvida

A biblioteca desenvolvida foi construída utilizando uma arquitetura orientada a *plugins*, onde cada comando do CLI é um artefato de software que não precisa ser compilado juntamente com o *cp-tools*. Essa característica aumenta a extensibilidade da ferramenta *cp-tools* uma vez que os plugins são carregados dinamicamente, em tempo de execução. A Figura 1 ilustra o diagrama de classe da biblioteca desenvolvida.

Esta arquitetura utiliza um gerenciador de *plugins*, a classe `PluginManager`, que é responsável por: localizar os módulos dos *plugins*, instanciá-los, executá-los e desalocá-los no final da execução.

A classe `PluginManager` implementa o padrão de projetos *Singleton*, onde somente

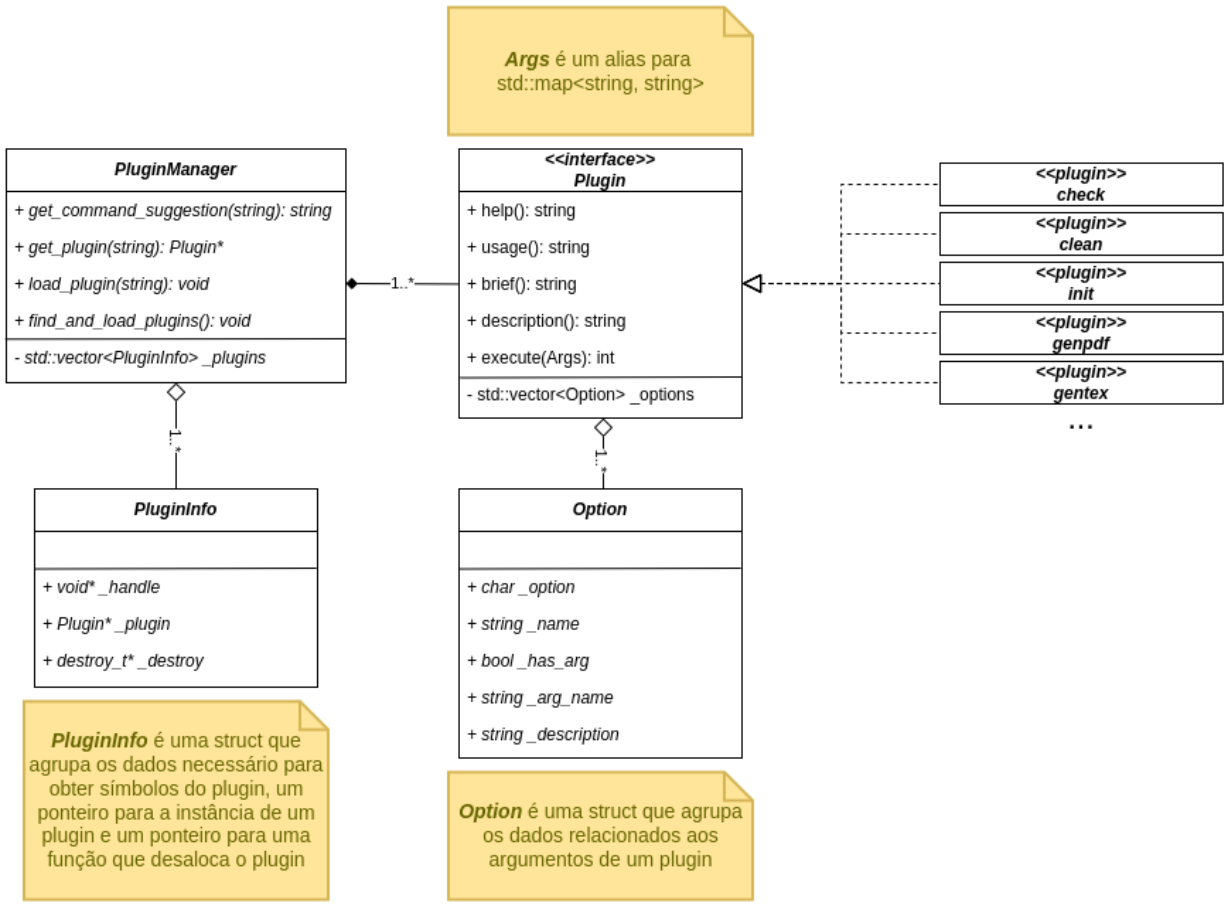

Figura 1 – Diagrama de classe de biblioteca de análise de argumentos de linha de comando desenvolvida para o *cp-tools*

é possível uma única instância durante toda execução da ferramenta. Este padrão foi necessário para evitar o constante carregamento dos módulos de *plugins*.

Na linguagem C++ existem diversas maneiras de implementar o padrão *Singleton*, porém, nem todas elas são seguras para ambientes com *threads*. Por mais que atualmente nenhum comando utilize *threads*, foi tomada a precaução de utilizar a implementação *thread-safe* de *singletons* proposta por Scott Meyers. O padrão proposto por Meyers (2005) baseia-se na seguintes características da linguagem C++:

- os objetos de uma função estática são inicializados quando o fluxo de controle atinge a função pela primeira vez;
- o tempo de vida das variáveis estáticas de uma função inicia quando o fluxo atinge a determinada função e termina no fim da execução;
- se dois fluxos de controle concorrentes atingem a função pela primeira vez, somente um dos fluxos irá inicializar as variáveis estáticas, bloqueando todos os demais fluxos

concorrentes;

A partir destas características da linguagem C++, o padrão proposto por Meyers (2005) baseia-se na utilização de construtores e destrutores privados, para impedir o acesso às novas instâncias, e disponibiliza um método público que retorna sempre a mesma instância da classe, tornando assim a classe um *Singleton*. Esta implementação na ferramenta *cp-tools* pode ser vista no Apêndice H, onde o construtor e destrutor da classe possui mensagens de *logs* que são impressas uma única vez, mesmo quando a classe é usada em rotinas concorrentes.

Já os *plugins* são classes que implementam a interface definida pela classe *Plugin*. Esta interface define um conjunto de métodos que devem ser implementados por todos os *plugin* concretos, como por exemplo os métodos `help()`, `usage()`, `brief()` e `execute()`. Desta forma, é estabelecido um contrato de interface onde o gerenciador de *plugins* sempre poderá executar determinados métodos, independente do *plugin* existente. Um exemplo de uso desta interface é a impressão do texto de ajuda da ferramenta (`cp-tools --help`), onde o gerenciador itera sobre os *plugins* carregados imprimindo o nome e a descrição dos *plugins* existentes.

3.5.2 Disponibilize um texto de ajuda quando os sinalizadores `-h` e `--help` forem passados

A biblioteca criada para o *cp-tools* permitiu a correção desta inconformidade de maneira simples. Uma vez que o gerenciador de *plugin* é o responsável pela execução de um determinado comando, é possível verificar previamente a presença dos sinalizadores `-h` e `--help`, e desta forma, forçar a execução da rotina de impressão do texto de ajuda.

Deste modo, o Código 3 apresenta o trecho de código que certifica que esta boa prática estará presente em todos os *plugins*, pois independente da implementação do método `execute()`, a rotina `help()` sempre será a única ação executada.

3.5.3 Se o usuário fez algo errado, tente adivinhar o que ele quis dizer

A biblioteca criada para o *cp-tools* permitiu a correção desta inconformidade de maneira simples. Uma vez que há um gerenciador de *plugin* que conhece o nome de todos os comandos presentes na ferramenta, foi possível implementar uma rotina de sugestão de comandos para os casos onde o comando digitado pelo usuário não existe mas se assemelha a um comando existente.

Para implementar esta funcionalidade foi utilizado o algoritmo da Distância de Levenshtein (LEVENSHTEIN, 1965). Desta forma, sempre que a rotina de busca pelo *plugin* falha é realizado uma busca pelos comandos com nomes parecidos e caso exista é

```
int main(int argc, char *argv[])
{
    auto& manager = PluginManager::get_instance();
    manager.find_and_load_plugins();

    auto subcommand = get_subcommand(argc, argv);
    auto plugin = manager.get_plugin(subcommand);

    auto args = parse_args(argc, argv, plugin->options());

    if(args.count("help"))
        std::cout << plugin->help() << '\n';
    else
        plugin->execute(args);
}
```

Código 3: Trecho de código que força a execução da impressão do texto de ajuda em casos onde os sinalizadores de ajuda estão presentes

exibida uma mensagem de sugestão para o usuário. A codificação desta funcionalidade é apresentada no Código 4, e uma situação onde a funcionalidade é executada pode ser vista no Código 5.

```
Plugin* PluginManager::get_plugin(const std::string& plugin_name) {
    // Busca pelo plugin com o nome especificado pelo usuário
    for(const auto& [_, plugin, __] : _plugins) {
        if(plugin->command() == plugin_name)
            return plugin;
    }

    // Caso não exista é impresso uma mensagem de alerta
    std::cerr << "cp-tools: '" << plugin_name
              << "' is not a cp-tools command. See 'cp-tools --help'.\n";

    // Busca pelo comando com o nome mais próximo do digitado pelo usuário
    auto [command_suggestion, proximity] = get_command_suggestion(plugin_name);

    // Caso a distância de levenshtein seja menor que 3 é sugerido o comando
    if(proximity < 3) {
        std::cerr << "\nThe most similar command is \n\t"
                  << command_suggestion << "\n";
    }

    exit(-1);
}
```

Código 4: Rotina de sugestão de comandos do software *cp-tools*

3.5.4 Trate os erros e emita uma mensagem significativa

Para corrigir esta inconformidade foi alterada a estratégia de tratamento de erros utilizada até então. Anteriormente as exceções geradas eram propagadas entre camadas até encontrar um tratador genérico, onde uma mensagem genérica era apresentada para o

```
$ cp-tools clear
cp-tools: "clear" is not a cp-tools command. See "cp-tools --help".

The most similar command is
        clean

$ cp-tools genlatex
cp-tools: "genlatex" is not a cp-tools command. See "cp-tools --help".

The most similar command is
        gentex
```

Código 5: Mensagem emitida quando o usuário digita algo errado durante a utilização da ferramenta *cp-tools*

usuário. Uma vez que os erros que podem ocorrer durante a execução não são contornáveis, ou seja, não há recuperação de erros, foi tomada a decisão de abortar o programa o mais rápido possível, simplificando assim o código e seus fluxos de execução. Outra vantagem desta abordagem é a possibilidade de apresentar uma mensagem mais significativa ao usuário, pois cada execução sujeita à exceção irá definir suas próprias mensagens, de acordo com a ação que estava em execução.

Deste modo, com esta mudança de estratégia, foram realizadas diversas mudanças no código do software, buscando modificar as rotinas de tratamento de erros e criar mensagens significativa atreladas. Um exemplo de como esta mudança de estratégia melhorou a base de código pode ser vista nos Códigos 6 e 7, onde é apresentada a rotina de remoção de arquivos e diretórios antes e depois da refatoração. Observe que antes a rotina de remoção retornava um objeto `Result` que abstrai o resultado da operação. Deste modo, o código cliente da rotina de remoção sempre precisa verificar se o resultado apresentou ou não algum erro. Já com a refatoração, o código cliente pode simplesmente executar a função sem tratamento de erro, pois o código cliente só irá continuar se a execução da rotina de remoção for bem sucedida, uma vez que a rotina de remoção aborta a execução em cenários de erros.

Por mais que vários comandos do software *cp-tools* tenham sido modificados para corrigir esta inconformidade, não foi possível aplicar todas as mudanças necessárias em toda a base de código. Desta forma, esta boa prática, ao término deste trabalho, foi parcialmente implementada, sendo assim uma das recomendações propostas para trabalhos futuros.

```
const Result remove(const std::string &path)
{
    if (not fs::exists(path).ok) return make_result(true);

    bool removed = false;

    try {
        removed = std::filesystem::remove_all(path);
    }
    catch (const std::filesystem::filesystem_error &err) {
        return make_result(
            false,
            CP_TOOLS_ERROR_CPP_FILESYSTEM_REMOVE,
            "",
            err.what()
        );
    }

    if (removed)
        return make_result(removed);
    else
        return make_result(
            removed,
            CP_TOOLS_ERROR_CPP_FILESYSTEM_REMOVE,
            "",
            "Impossible to remove " + path
        );
}
```

Código 6: Rotina de remoção do software *cp-tools* antes da refatoração

```
void remove(const std::string &path)
{
    if (not fs::exists(path)) return;

    try {
        std::filesystem::remove_all(path);
    }
    catch (const std::filesystem::filesystem_error &err) {
        std::cout << cptools::message::error(err.what()) << '\n';
        exit(CP_TOOLS_ERROR_FS_REMOVE);
    }
}
```

Código 7: Rotina de remoção do software *cp-tools* depois da refatoração

4 Conclusão

Neste capítulo são discutidos os resultados obtidos após execução deste trabalho. Na Seção 4.1 são discutido os resultados deste trabalho e na Seção 4.2 são apresentadas recomendações para trabalhos futuros relacionadas com os temas abordados neste trabalho.

4.1 Discussão dos resultados

Este trabalho se propôs a avaliar a interface de linha de comando do software *cp-tools*. Para isso foi necessário o estudo de diversos guias, manuais e artigos que descrevem as melhores práticas e recomendações para o desenvolvimento de CLIs flexíveis e intuitivas. Deste modo, o primeiro resultado deste trabalho foi o agrupamento dos principais documentos que abordam desenvolvimento de CLIs, facilitando assim as pesquisas futuras sobre este tema.

O segundo resultado deste trabalho foi a criação de um manual de desenvolvimento de CLIs específico para o software *cp-tools*, que poderá auxiliar futuros mantenedores do projeto durante o desenvolvimento da interface de linha de comando. Além deste manual, também foi apresentada uma metodologia de avaliação que poderá ser utilizada para verificar se as recomendações e práticas selecionadas estão ou não sendo seguidas.

Outro resultado deste trabalho foi trazer visibilidade aos problemas existentes na interface. Além de listar os problemas, este trabalho priorizou e corrigiu parte dos problemas encontrados. Das 7 inconformidades encontradas, 3 foram completamente corrigidas, 1 parcialmente corrigida e 3 não foram corrigidas. Por mais que todos os problemas não tenham sido corrigidos, este trabalho produziu um relatório que descreve com detalhes as situações onde as inconformidades se encontram, deste modo, futuros desenvolvedores do projeto poderão utilizar este relatório para corrigir tais inconformidades.

Desta forma, dos 3 objetivos inicialmente propostos, 2 deles foram completamente atingidos e 1 foi parcialmente atingindo. O objetivo, “Realizar modificações no software buscando corrigir as inconformidades encontradas”, foi considerado parcialmente atingido uma vez que não foi possível corrigir todas as inconformidades encontradas. As situações dos objetivos deste estudo de caso são apresentadas na Tabela 7.

Tabela 7 – Situação dos objetivos definidos para este estudo de caso

Objetivos	**Status**
Conhecer as principais recomendações e boas práticas de desenvolvimento de CLI aplicáveis ao software *cp-tools*	Completamente atingido
Analisar a CLI do software *cp-tools* visando identificar quais boas práticas e recomendações estão presente e quais não estão	Completamente atingido
Realizar modificações no software *cp-tools* buscando corrigir as inconformidades encontradas	Parcialmente atingido

4.2 Recomendações para futuros trabalhos

A primeira recomendação para futuros trabalhos é a pesquisa por outros guias e manuais de desenvolvimento de CLIs. Conforme foi listado na Tabela 4, há pouco referencial teórico para este tema, e nenhum deles está disponível em língua portuguesa. Deste modo, a segunda recomendação é a criação de um manual ou guia de desenvolvimento, específico ou genérico, em língua portuguesa, de CLIs.

Outra recomendação, agora específica para o software objeto de estudo, é a evolução do manual e da metodologia de avaliação desenvolvida. Com a evolução do software, novos critérios de avaliação poderão ser aplicáveis, deste modo, é preciso atualizar o manual de recomendações para que a inspeção da interface continue relevante. Outra recomendação específica para o software *cp-tools* é a realização das correções não realizadas neste trabalho. Conforme foi explicado na Seção 4.1, ainda é possível realizar ações corretivas para 4 das 7 inconformidades encontradas.

Por fim, a última recomendação é aplicação desta metodologia de avaliação para outros softwares que possuam uma interface de linha de comando.

Referências

BARAUD, P.-J. *TTY: under the hood*. 2016. Disponível em: <https://www.yabage.me/2016/07/08/tty-under-the-hood/>. Acesso em: Oct, 06. 2021. Citado 7 vezes nas páginas 49, 50, 51, 52, 53, 54 e 55.

BARBOSA, S.; SILVA, B. *Interação Humano-Computador*. [S.l.]: Elsevier Brasil, 2010, 2010. ISBN 8535211209, 9788535211207. Citado na página 20.

BARELL, J. *Problem-Based Learning: An Inquiry Approach*. 2nd. ed. 2455 Teller Rd, Newbury Park, CA 91320, Estados Unidos: Corwin Publishers, 2006. ISBN 141295004X, 978-1412950046. Citado na página 58.

BLUM, R.; BRESNAHAN, C. *Linux Command Line and Shell Scripting Bible*. 3rd. ed. 111 River St, Hoboken, NJ 07030, Estados Unidos: Wiley, 2015. ISBN 111898384X, 978-1118983843. Citado 2 vezes nas páginas 15 e 50.

CHAZELAS, S. *What are the responsibilities of each Pseudo-Terminal (PTY) component?* 2014. Disponível em: <https://unix.stackexchange.com/questions/117981/what-are-the-responsibilities-of-each-pseudo-terminal-pty-component-software#answer-120071>. Acesso em: Mar, 17. 2014. Citado na página 55.

CODEFORCES. *Website hosting competitive programming contests*. 2021. <https://codeforces.com/>. Accessed: 2021-10-06. Citado na página 99.

COOPER, M. *Advanced Bash Scripting Guide - Volume 2: An in-depth exploration of the art of shell scripting*. 2nd. ed. [S.l.]: Independently Published, 2019. ISBN 1707048916, 978-1707048915. Citado 2 vezes nas páginas 62 e 63.

CORMEN, T. H. et al. *Introduction to Algorithms*. 2nd. ed. The MIT Press, 2001. ISBN 0262032937. Disponível em: <http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262032937%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262032937>. Citado na página 57.

COSTA, E. A. da. *cp-tools*. 2021. Disponível em: <https://github.com/edsomjr/competitive-problems-tools>. Acesso em: Oct, 06. 2021. Citado na página 59.

CYBIS, W. de A. *Ergonomia e Usabilidade: Conhecimentos, Métodos e Aplicações*. [S.l.]: Laboratório de Utilizabilidade de Informática, 2014, 2003. Citado na página 20.

FASSBINDER, A. G. O.; PAULA, L. C.; ARAúJO, J. C. D. Experiências no estímulo à prática de programação através do desenvolvimento de atividades extracurriculares relacionadas com as competições de conhecimentos. *Anais do XXXII Congresso da Sociedade Brasileira de Computação, XX WEI – Workshop sobre Educação em Computação, Curitiba.*, 2012. Disponível em: <http://www.imago.ufpr.br/csbc2012/anais_csbc/eventos/wei/artigos/Experiencias%20no%20estimulo%20a%20pratica%20de%20Programacao%20atraves%20do%20desenvolvimento%20de%

20atividades%20extracurriculares%20relacionadas%20com%20as%20competicoes%20de%20conhecimentos.pdf>. Citado na página 57.

FRASSON, A. et al. Disciplinaridades e a aprendizagem baseada em problemas. In: . [S.l.: s.n.], 2017. Citado na página 58.

Free Software Foundation. *GNU Coding Standards*. 2021. Disponível em: <https://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html>. Acesso em: Oct, 08. 2021. Citado 3 vezes nas páginas 22, 31 e 33.

HIRAMA, K. K. *Engenharia de Software: Qualidade e produtividade com tecnologia*. 1st. ed. Rua Conselheiro Nébias, 1.384 - Centro Histórico de São Paulo, São Paulo - SP, 01203-904: GEN LTC;, 2012. ISBN 853524882X, 978-8535248821. Citado na página 57.

IBM. *IBM Command-line interface conventions*. 2021. Disponível em: <https://www.ibm.com/docs/en/csm/6.2.11?topic=csmcli-command-line-interface-conventions>. Acesso em: Oct, 08. 2021. Citado na página 22.

IEEE and The Open Group. *Utility Conventions - POSIX Conventions*. 2021. Disponível em: <https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html>. Acesso em: Oct, 08. 2021. Citado 3 vezes nas páginas 22, 31 e 33.

INEP. *Censo da Educação do Ensino Superior*. 2019. Disponível em: <https://www.gov.br/inep/pt-br/areas-de-atuacao/pesquisas-estatisticas-e-indicadores/censo-da-educacao-superior/resultados>. Citado na página 57.

LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 1965. Citado 2 vezes nas páginas 37 e 66.

MEYERS, S. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. [S.l.]: Addison-Wesley Professional, 2005. ISBN 0321334876. Citado 3 vezes nas páginas 36, 37 e 107.

NORMAN, D. A. *The design of everyday things*. [New York]: Basic Books, 2002. ISBN 0465067107 9780465067107. Disponível em: <http://www.amazon.de/The-Design-Everyday-Things-Norman/dp/0465067107/ref=wl_it_dp_o_pC_S_nC?ie=UTF8&colid=151193SNGKJT9&coliid=I262V9ZRW8HR2C>. Citado na página 19.

NUNES, D. S. N. *ds-contest-tools*. 2020. Disponível em: <https://github.com/danielsaad/ds-contest-tools>. Acesso em: Oct, 06. 2021. Citado na página 59.

PRASAD, A. et al. *Command Line Interface Guidelines*. 2018. <https://clig.dev/>. Accessed: 2021-10-05. Citado 9 vezes nas páginas 15, 16, 21, 22, 31, 33, 61, 63 e 79.

PRESSMAN, R. S. *Engenharia de software*. 9th. ed. [S.l.]: AMGH, 2021. ISBN 6558040107, 978-6558040101. Citado na página 57.

ROGERS, Y.; SHARP, H.; PREECE, J. *Interaction Design: Beyond Human - Computer Interaction*. [S.l.]: John Wiley & Sons, Ltd., 2011. v. 6. ISBN 0470665769. Citado na página 20.

SOMMERVILLE, I. *Software Engineering*. 9. ed. Harlow, England: Addison-Wesley, 2010. ISBN 978-0-13-703515-1. Citado na página 57.

WU, J.; CHEN, S.; YANG, R. Development and application of online judge system. *2012 International Symposium on Information Technologies in Medicine and Education*, v. 1, p. 83–86, 2012. Citado na página 58.

ZHIGANG, S. et al. Moodle plugins for highly efficient programming courses. In: . [S.l.: s.n.], 2001. Citado na página 58.

ÅKESSON, L. *The TTY demystified*. 2008. Disponível em: <http://www.linusakesson.net/programming/tty/>. Acesso em: Aug, 29. 2018. Citado 2 vezes nas páginas 52 e 55.

Apêndices

APÊNDICE A – Console, Terminal e TTY

Este apêndice apresenta a terminologia comumente utilizada quando é abordado o tema interface de linha de comando. Além da terminologia, também é explicada a cronologia da evolução dos dispositivos atuantes na interação com as CLIs.

A.1 Console, Terminal e TTY

Para compreender o fluxo de execução de uma CLI no sistema operacional, é preciso primeiramente entender e diferenciar os termos *Console*, *Terminal* e *TTY*.

Console, *terminal* e *TTY* são termos que atualmente estão intimamente ligados, mas nem sempre foi assim. Originalmente, esses termos eram usados para designar equipamentos dedicados a interagir com computadores *mainframes* (BARAUD, 2016).

Na década de 60, os *mainframes* eram grandes computadores utilizados por vários usuários simultâneos, por meio de dispositivos conectados por cabos. Os dispositivos que estavam no outro extremo dessas conexões eram chamados de terminais. No começo, esses equipamentos eram dispositivos eletromecânicos chamados de teletipo (em inglês, *teletypewriter*), cuja abreviação é *TTY* (BARAUD, 2016). A Figura 2 mostra um exemplo desse dispositivo eletromecânico.

Já o termo *console* era utilizado para descrever o terminal que se situava no ambiente do computador *mainframe*. Em alguns *mainframes*, esse *console* possuía uma tela e um teclado, e o operador podia realizar operações diretas no computador, conforme mostra na Figura 3.

Porém, esses termos perderam seus significados originais com a popularização dos computadores pessoais, onde o equipamento que o usuário interagia e o que realizava o processamento era o mesmo. Os sistemas operacionais desses computadores pessoais, buscando manter as analogias, continuaram utilizando os termos *console* e *terminal*, mas agora para denominar aplicações de softwares que o usuário utilizava para interagir com o sistema operacional.

Dessa forma, *terminal*, *TTY* e *console* são termos que originalmente descreviam *hardwares* que estavam conectados com um computador *mainframe*. Rapidamente os usuários desses equipamentos eletromecânicos sentiram falta de funcionalidades como a automação de execução de certas sequências de comandos, visualização e recuperação de histórico de comandos, definição de apelidos para comandos (*aliasing*) e recursos interativos como autocompletar e deleção de caracteres já digitados. Para resolver essas demandas,

Figura 2 – Teletipo “Fernschreiber 100” fabricado pela Siemens na década de 1960

Fonte: Iron Bishop, 2006.
Disponível em: <https://commons.wikimedia.org/wiki/File:Fernscheiber_01.jpg>

foi desenvolvida uma aplicação intermediária de software denominada *shell* (BARAUD, 2016).

A.2 Shell

O *shell* é um utilitário disponível em distribuições UNIX que possibilita aos usuários iniciarem programas, gerenciarem arquivos de sistema, gerenciarem processos em execução, redirecionar fluxos de dados e outras tarefas (BLUM; BRESNAHAN, 2015).

Por meio do *shell* é possível executar comandos, que podem ser nativos do sistema operacional ou programas de terceiros. A execução da aplicação *shell* pode ser compreendida como um *loop* de leitura-avaliação-impressão (*Read-Eval-Print Loop – REPL*). Por meio desse *loop*, a aplicação realiza a leitura dos textos digitados pelo usuário, executa-os, e imprime os resultados na tela. A leitura dos dados ocorre por meio da parte interativa do *shell*, que é comumente chamada de *prompt* de comando (BLUM; BRESNAHAN, 2015).

O *shell* é um software executado em um *terminal*, que antigamente era um dispositivo eletromecânico, mas hoje é um software que emula o antigo equipamento de hardware. Dessa forma, o *shell* não interage diretamente com o sistema operacional, e sim com o *terminal*, sendo esse o responsável por executar as rotinas de chamada de sistema para executar os comandos digitados pelo usuário (BLUM; BRESNAHAN, 2015).

Figura 3 – Console conectado no *mainframe* "IBM 360/85" utilizado na NSA durante a década de 70

Fonte: Agente não identificado do governo dos EUA, 1971.
Disponível em:
<https://commons.wikimedia.org/wiki/File:Supercomputer_NSA-IBM360_85.jpg>

Mesmo após a mudança de paradigma que os computadores pessoais conduziram para a computação, onde toda a capacidade de processamento se encontra em um mesmo dispositivo, a arquitetura interna dos sistemas operacionais UNIX ainda remonta a era dos *mainframes*. Uma comprovação disso é o módulo responsável pela integração do terminal com o sistema operacional, o subsistema *TTY* (*TTY subsystem*), onde o termo *TTY* ainda remete aos dispositivos de teletipo (BARAUD, 2016).

Os sistemas operacionais modernos disponibilizam o *shell* por meio de um terminal emulado por software (*Software Emulated Teletypes*). Quando o terminal é emulado no espaço do usuário, geralmente com uma interface gráfica, é chamado de *pseudo-teletypes*, cuja sigla abreviada em inglês é PTS. Quando o terminal é emulado diretamente pelo *kernel*, é chamado de terminal emulado *TTY* (BARAUD, 2016).

A.3 Fluxo de Execução de uma CLI

Conforme explicado na Seção A.2, a arquitetura e os módulos do subsistema de TTY dos sistemas operacionais modernos ainda remetem aos primeiros sistemas UNIX. Dessa forma, é de grande valia a compreensão da estrutura original do subsistema de TTY, para assim entender a sua estrutura atual.

A.3.1 Subsistema TTY nos *mainframes*

O fluxo de execução nos antigos *mainframes* iniciava com eventos de teclas nos terminais físicos, muitas vezes equipamentos eletromecânicos denominados teletipos. Esse terminal era conectado por um par de cabos a um equipamento de hardware denominado *UART* (*universal asynchronous receiver-transmitter*). O equipamento de hardware *UART* interage com o sistema operacional do *mainframe* por meio do *driver* de *UART*, que gerenciará a transmissão física de *bytes*, incluindo verificações de paridade e controle de fluxo (ÅKESSON, 2008).

Em sistemas mais simples, o *driver* de *UART* entregaria o fluxo de *bytes* direto para alguma aplicação, porém, essa abordagem carecia de algumas funcionalidades fundamentais, por exemplo, a edição de textos já digitados e tratamento de sequências de teclas especiais. Dessa forma, após o *driver* de *UART*, o fluxo passa pelo módulo denominado *line discipline*, cuja responsabilidade é tratar a entrada de caracteres especiais (`ctrl+c`, `ctrl+\`, `ctrl+d`, `backspace`, por exemplo), ecoar os caracteres digitados no dispositivo de saída (*local echoing*), gerenciamento do *buffer* de entrada e outros. Após o tratamento pelo módulo *line discipline*, o fluxo de dados é passado para o módulo *TTY Driver* quando o usuário aperta a tecla `enter` (BARAUD, 2016).

Uma vez que o fluxo de informação chegou ao módulo *tty driver*, são executadas uma série de rotinas internas do *kernel* para iniciar a execução do comando informado pelo usuário. É responsabilidade do *tty driver* executar as rotinas necessárias para encontrar e executar o binário do comando digitado pelo usuário, além de transmitir seus parâmetros de execução (BARAUD, 2016). Esse fluxo pode ser visualizado na Figura 4.

A.3.2 Subsistema TTY nos computadores pessoais

Com o advento dos computadores pessoais, vários dispositivos eletromecânicos utilizados nos computadores *mainframes* foram virtualizados, para assim aproveitar os softwares existentes (BARAUD, 2016).

Os dispositivos de teletipo, que antes funcionavam como um terminal que dependiam de um hardware de *UART* e seu respectivo *driver*, foram substituídos por monitores, teclados e seus respectivos *drivers*. O terminal que antes era um dispositivo físico, agora é um componente de software que emula um teletipo, possibilitando que em um mesmo computador existam várias sessões de terminais em paralelo.

A conexão bidirecional que antes era feita por cabos, em sistemas operacionais modernos do tipo UNIX, são feitas por meio de arquivos, de modo que para cada terminal

[1] Tradução própria do texto da Figura 4: (1) “O teletipo é conectado a uma das várias portas seriais disponíveis”. (2) “O controlador *UART* desserializa a mensagem elétrica recebida para ser lida pelo sistema operacional”. (3) “A Line Discipline ecoa algumas informações dependendo da configuração e do contexto”. (4) “O driver TTY roteia a entrada/saída de/para a sessão anexada a este TTY”.

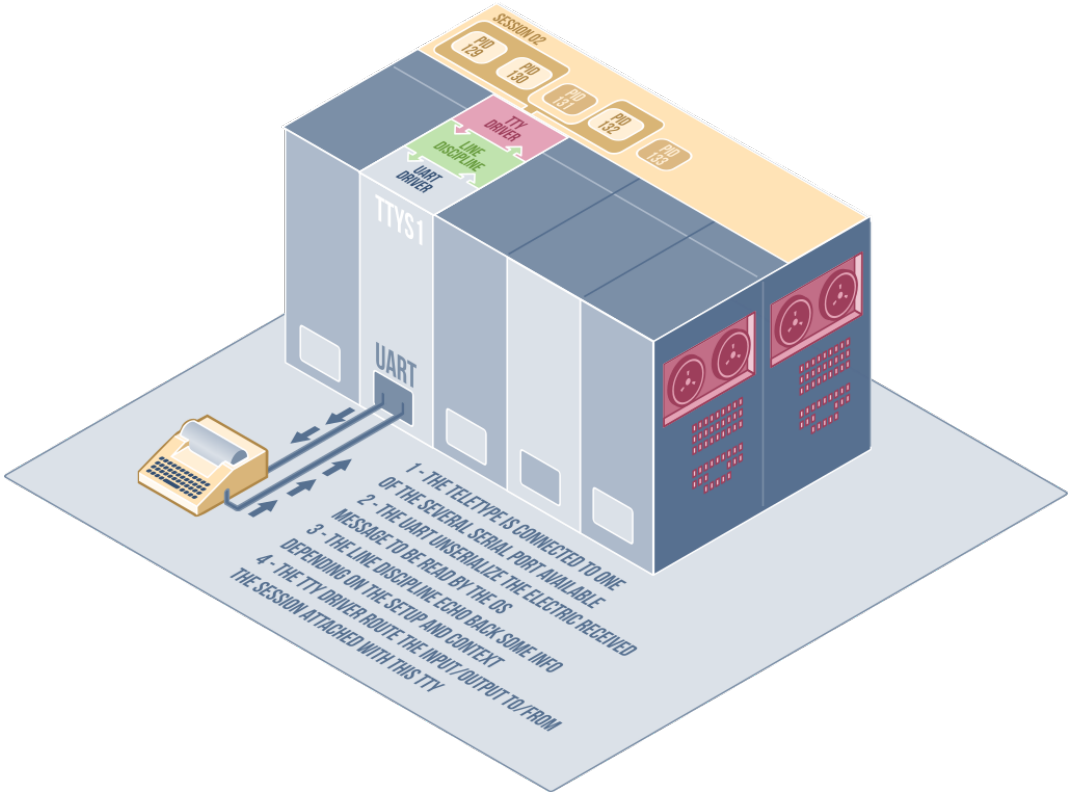

Figura 4 – Ilustração de como os antigos teletipos interagiam com os computadores *mainframe*[1]

Fonte: (BARAUD, 2016)

aberto existe um arquivo no caminho `/dev/pts/<number>`.

Antes os dispositivos de entrada e saída (o teclado e monitor) estavam sempre conectados a somente uma sessão de terminal por vez, e agora, em uma mesma tela, podem existir diversos terminais com fluxos de dados independentes, transferindo dados simultaneamente. Desse modo, para lidar com essa complexidade dos terminais emulados, o sistema operacional disponibiliza o arquivo `/dev/ptmx`, que, quando aberto, executa uma série de rotinas internas do *kernel* que resultam na criação de um pseudo terminal escravo (*pts*). Sendo assim, o programa em execução irá interagir com um *pts* associado a um determinado arquivo `/dev/pts/<number>` e toda escrita e leitura irá ecoar no pseudo terminal mestre que possui o descritor de arquivo `/dev/ptmx`, possibilitando assim fluxos de dados independentes, mantendo a mesma interface. Esse fluxo de informação e seus componentes podem ser visualizados na Figura 5.

A Figura 6 apresenta a visão macro do fluxo de dados e dos principais módulos atuantes na execução de um programa via um terminal de comandos. A interação começa

[2] Tradução própria do texto da Figura 5: (1) “O emulador de terminal "pede"ao escravo mestre para alocar um novo PTS para cada nova janela.” (2) “A parte master gera novos PTS e os anexa ao emulador de terminal solicitado.” (3) “Os processos de cada janela são vinculados ao seu PTS assim como o mecanismo tty.” (4) “O acesso de leitura/gravação no PTS é concedido para o usuário atual logado.”

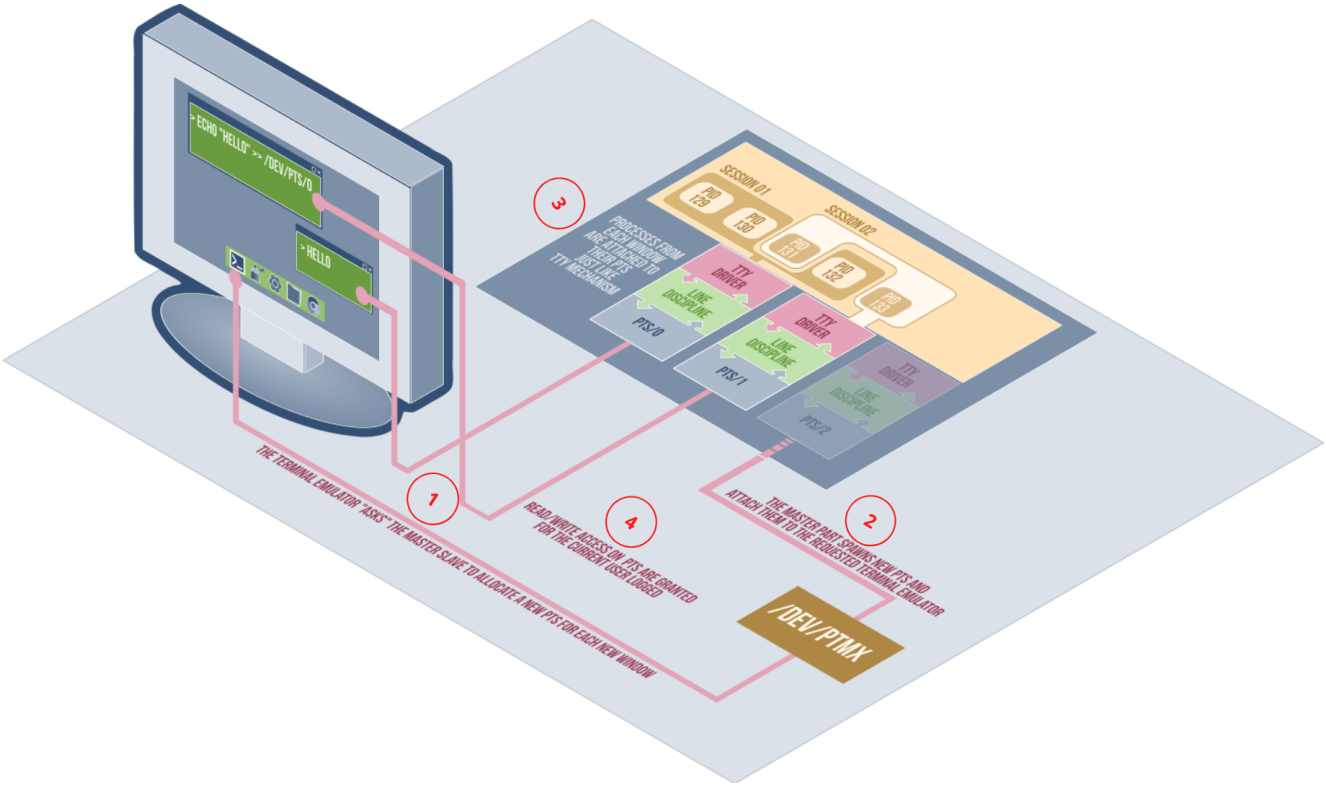

Figura 5 – Diagrama com o fluxo de execução em ambiente com terminais virtuais[2]

Fonte: (BARAUD, 2016)

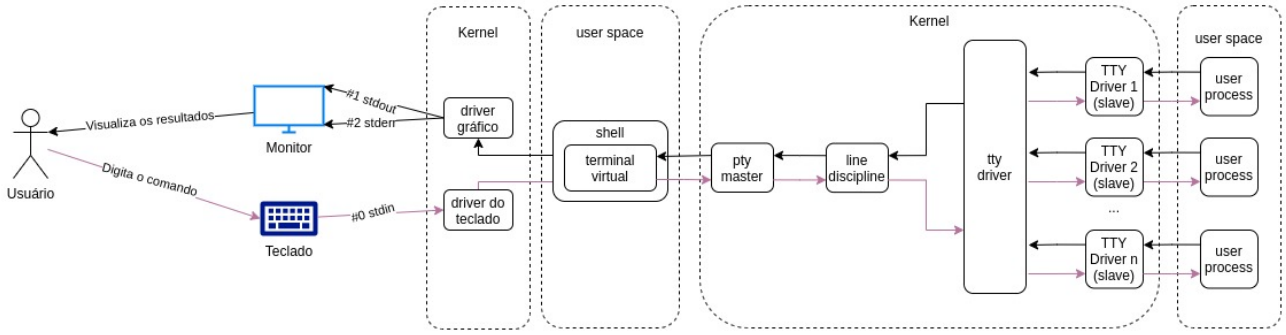

Figura 6 – Fluxos de dados entre os componentes atuantes na execução de um comando no *shell*

por iniciativa do usuário, utilizando um dispositivo de entrada, representada nesse fluxo por um teclado.

Uma vez que os sinais de teclas são emitidos pelo teclado, o seu respectivo *driver* irá interceptar e converter tais sinais de tecla para um formato que possa ser captado pelo sistema operacional e os programas em seu computador. No fluxo o usuário está interagindo com o software do *shell*, que por sua vez está associado ao software de um terminal emulado.

Dada essa organização entre os módulos de software, os caracteres passam pelo *shell*, que por sua vez utiliza a API do terminal e executa a rotina responsável por abrir e escrever no arquivo `/dev/ptmx`. O fluxo de dados no arquivo simula uma transmissão serial do software do terminal para o software chamado *line discipline*, cuja responsabilidade é implementar o editor de linha interno do dispositivo de entrada, além de tratar a entrada de sinais de teclas especiais. Uma vez que a tecla de fim de comando (geralmente a tecla *enter*) é digitada, o módulo *line discipline* realiza a comunicação com o *driver tty*.

Por fim, o *driver tty* realiza os procedimentos internos do sistema operacional para localizar os binários e executáveis associados com o comando digitado, cria uma nova instância de um pseudo terminal escravo (pts) e o associa ao arquivo `/dev/pts/<number>` criado durante a abertura do arquivo `/dev/ptmx`, criando assim o fluxo de retorno para os dados emitidos pelo pts (BARAUD, 2016) (CHAZELAS, 2014) (ÅKESSON, 2008).

APÊNDICE B – Formação de profissionais de Software e o *cp-tools*

O mundo moderno não poderia existir sem software (SOMMERVILLE, 2010). Os sistemas computacionais estão presentes em praticamente todas as áreas do mundo moderno: sistemas elétricos, sistemas de transporte de carga, pregões na bolsa de valores e até mesmo os sistemas eleitorais dependem de software (HIRAMA, 2012).

Uma característica de sistemas de software é sua intangibilidade, ou seja, tais produtos não estão sujeitos às propriedades materiais como localidade e temporariedade. Essa característica, ao mesmo tempo que aumenta o potencial de um sistema, também torna-o mais difícil de se produzir. É comum que produtos de softwares sejam criados sem o devido planejamento e tornem-os sistemas extremamente complexos e caros de se manter e adotar (PRESSMAN, 2021).

Uma vez que foi compreendida a importância e a complexidade dos sistemas de software, estudos foram realizados em várias áreas de conhecimento buscando compreender o ciclo de vida desses produtos, sendo a Engenharia de Software uma delas (SOMMERVILLE, 2010).

O principal objetivo da Engenharia de Software está nos processos que envolvem o ciclo de vida desses produtos, desde sua concepção até sua manutenção, quando o sistema já está em uso (PRESSMAN, 2021), porém, compreender o ciclo de vida não é suficiente. Outras áreas de conhecimento, como a Análise de Algoritmos, dedicaram-se ao estudo de sequências finitas de ações executáveis que visam obter uma solução para um determinado tipo de problema (CORMEN et al., 2001). Dessa forma, um profissional precisa ter conhecimentos tanto de engenharia de software, quanto de algoritmos, pois sem tais conhecimentos os produtos desenvolvidos podem estar sujeitos aos atrasos nas entregas, problemas de desempenho, requisitos não atendidos, desperdícios de recursos computacionais, problemas de manutenibilidade e outras situações que podem tornar o projeto mais caro ou até mesmo inviável.

Assim, fica evidente que a formação de um profissional de software é uma tarefa complexa. O ensino das disciplinas que envolvem sistemas computacionais é uma grande dificuldade para as instituições de ensino, e a falta de uma abordagem sistemática contribui para o alto índice de evasão dos cursos que se propõem a formar profissionais da área (INEP, 2019). De acordo com Fassbinder, Paula e Araújo (2012), um dos fatores que dificulta a formação de tais profissionais é a falta de aplicabilidade prática no processo de ensino. Muitas vezes, conceitos complexos são apresentados sem uma devida contex-

tualização e aplicabilidade, o que dificulta o entendimento dos conceitos por parte dos alunos.

Levando isso em consideração, uma abordagem que algumas instituições de ensino adotaram foi a metodologia *PBL* (*Problem-Based Learning*), que em português significa Aprendizado Baseado em Problemas (ou Projetos) (BARELL, 2006). Essa abordagem é uma metodologia voltada para a aquisição de conhecimento por meio da resolução de problemas relacionados com os tópicos de ensino (FRASSON et al., 2017). Porém, uma grande dificuldade do aprendizado baseado em problemas é sua escalabilidade quando aplicada para muitas pessoas por poucos instrutores. É comum que os estudantes produzam uma grande quantidade de artefatos e os instrutores não tenham tempo hábil para correção e *feedback*.

Uma solução para esse problema de escalabilidade da metodologia PBL é a utilização de juízes eletrônicos para a correção automatizada dos artefatos produzidos pelos alunos. Juízes eletrônicos são ferramentas de automatização encarregadas pelo gerenciamento e pelas correções dos artefatos enviados pelos alunos.

A utilização de um juiz eletrônico depende da configuração prévia por parte do instrutor, sendo necessário a elaboração de uma situação problema, juntamente com um conjunto de rotinas de testes, em uma estrutura de diretório específica que o juiz eletrônico impõe, muitas vezes chamada de pacote do problema. Após o preparo e importação do pacote do problema para o juiz eletrônico, os alunos poderão submeter os artefatos produzidos, que por sua vez serão confrontados com as rotinas de testes do problema, com a finalidade de verificar se a solução do aluno realmente resolve a situação problema (WU; CHEN; YANG, 2012).

Quando falamos de juízes no contexto de problemas de programação podemos separá-los em dois tipos: os juízes eletrônicos e os juízes online. Os juízes eletrônicos é um tipo de software capaz de compilar e verificar automaticamente o código-fonte submetido pelos alunos com base em um conjunto predefinido de arquivos de entrada e saída (ZHIGANG et al., 2001). Já os juízes online, além de serem juízes eletrônicos, possuem outras funcionalidades. Normalmente, os juízes online possuem uma base de dados que contém diversos problemas categorizados, ranking de usuários com melhor desempenho, estatística de desempenho ao decorrer do tempo, área com fóruns de dúvidas.

Alguns dos juízes mais conhecidos são: Codeforces[1], Beecrowd (Antigo URI Online Judge)[2], Online Judge (antigo Uva Online Judge)[3] e BOCA Online Contest Administra-

[1] Juíz online desenvolvido pela universidade russa *Saratov State University*. Disponível em: <https://codeforces.com/>

[2] Beecrowd é o novo nome do antigo URI Online Judge, projeto desenvolvido pelo Departamento de Ciência da Computação da Universidade URI (Universidade Regional Integrada - Campus de Erechim). Disponível em: <https://www.beecrowd.com.br/>

[3] Juíz online desenvolvido na universidade espanhola *University of Valladolid*. Disponível em: <https:

tor[4].

Geralmente, cada juiz eletrônico possui sua própria formatação de problemas, fato esse que resulta em diversas dificuldades. A primeira delas é a dificuldade de portabilidade entre plataformas, sendo necessário reformatar o pacote de problema para realizar as importações. Outro problema é a dificuldade de colaboração entre os elaboradores de problemas, uma vez que cada elaborador está acostumado com a formatação de uma determinada plataforma.

Uma vez que os instrutores adotem a utilização de juízes eletrônicos em sua metodologia de ensino, a dificuldade que antes era de corrigir várias soluções dos alunos, agora é a de formatar problemas para os vários juízes eletrônico existentes. Para solucionar essa nova dificuldade, diversas iniciativas foram feitas, sendo uma delas a criação de ferramentas que auxiliam na formatação de problemas para as diversas plataformas. Algumas dessas ferramentas são: *cp-tools* (COSTA, 2021), *ds-contest-tools* (NUNES, 2020) e outros.

//onlinejudge.org/>

[4] Juíz eletrônico desenvolvidos na universidade USP. Disponível em: <https://www.ime.usp.br/~cassio/boca/>

APÊNDICE C – Manual de boas práticas e recomendações do software *cp-tools*

Esse manual reúne as boas práticas e recomendações de diversos guias e manuais de desenvolvimento de CLIs. Além da listagem das práticas e recomendações, esse guia apresenta exemplos de outras ferramentas com CLIs que atendem tais práticas e recomendações.

As boas práticas e recomendações selecionadas foram agrupadas de acordo com os grupos definidos pelo manual *Command Line Interface Guidelines* (PRASAD et al., 2018). Os grupos presentes nesse manual são:

- boas práticas gerais;
- boas práticas de ajuda ao usuário;
- boas práticas de tratamento de erros;
- boas práticas na utilização de argumentos e sinalizadores;
- boas práticas na interatividade;
- boas práticas no uso de subcomandos.

C.1 Boas práticas gerais

As boas práticas gerais é o agrupamento de todas as recomendações que são aplicáveis à grande maioria dos CLIs, independentemente de seu contexto.

C.1.1 Use uma biblioteca de análise de argumento de linha de comando

A primeira tarefa executada por um programa acionado pela sua CLI é a análise de argumentos de linha de comando (*command-line argument parsing*). Essa é uma tarefa base, onde os argumentos e opções passados pelo usuário são extraídos e pré-processados para um formato específico, buscando interpretar a intenção do comando e executá-lo (PRASAD et al., 2018).

É recomendado a utilização de uma biblioteca de análise de argumentos de linha de comando por dois principais motivos. Primeiro, a tarefa de extrair dados de um determinado recurso da linguagem de programação, no caso da linguagem C e C++, as variáveis

Tabela 8 – Principais bibliotecas de análise de argumento de linha de comando e suas respectivas linguagens de programação

Bibliotecas	**Suporte linguagem de programação**	**Stars no Github**
docopt	Multiplataforma	7.5k
argbash	Bash	927
Cobra	Go	23.3k
CLI	Go	16.6k
optparse-applicative	Haskell	746
picocli	Java	3.2k
oclif	Node	6.5k
console	PHP	8.9k
CLImate	PHP	1.8k
Argparse	Python	-
Click	Python	11.4k
Typer	Python	6.3k
TTY	Ruby	2.3k
clap	Rust	6.6k
structopt	Rust	2.2k
swift-argument-parser	Swift	2.3k
cxxopts	C++	2.7k

argc e *argv*, é praticamente idêntica para vários programas, sendo assim desnecessário a sua reimplementação. E segundo, essas bibliotecas lidam com complexidades como análise de erros de digitação, suporte aos argumentos e *flags* sem ordem definida, gerador de documentação e outros. Desse modo, por meio da utilização de uma biblioteca de análise de argumentos de linha de comando, os desenvolvedores da CLI evitam repetições de códigos, além de tirar proveito de funcionalidades que não são facilmente implementadas, como o gerador dinâmico de documentação e a sugestão de comandos após detecção de erros de digitação.

A Tabela 8 apresenta algumas das principais bibliotecas de análise de argumentos que são utilizadas, assim como sua popularidade na plataforma de hospedagem de código-fonte GitHub.

C.1.2 Retorne zero em caso de sucesso

Todo programa UNIX, ao finalizar sua execução, emite um código de saída. Em algumas linguagens de programação, como C e C++, se a rotina de emissão desse código de saída for omitida, o código zero é emitido. Dessa forma, é padronizado que o código de retorno zero sinaliza uma rotina de execução bem sucedida, e qualquer outro valor de retorno pode ser interpretado como um problema na execução (COOPER, 2019).

Uma vez que comandos CLIs podem estar encadeados em uma sequência de ações, onde a ação posterior depende da execução correta do comando anterior, é altamente

recomendado que seja retornado zero em casos de sucesso e um valor diferente de zero em casos de erro, pois caso contrário, tornaria mais complexa a utilização da CLI em situações com vários comandos encadeados (COOPER, 2019) (PRASAD et al., 2018).

C.1.3 Utilize por padrão os *streams* de dados `stdout` e `stderr`

A saída padrão do programa deve ser direcionada para o `stdout`. Qualquer informação que possa ser necessária para outro programa, em caso de encadeamento de execução, também deve ir para o `stdout`, pois é para onde o encadeamento de fluxo padrão (*pipes*) do UNIX envia os dados de saída.

Já as mensagens de erros, *logs* e outras informações devem ser enviadas para o `stderr`. Essa abordagem possibilita que a informação chegue no usuário mesmo quando uma sequência de comandos está encadeada, pois por padrão o fluxo de dados `stderr` está associado ao terminal em execução.

C.1.4 Disponibilize a versão e a licença do software quando o sinalizador `--version` for utilizado

O CLI deve implementar os sinalizadores comumente utilizados para obter informações sobre a versão e a licença do software. É uma boa prática utilizar o sinalizador `--version`, pois esse é comumente utilizado em diversas CLIs.

Além da versão e a licença do software, é recomendado que na mensagem impressa ao usuário também sejam informados os autores envolvidos na elaboração da ferramenta e seus contatos.

C.1.5 O nome do utilitário e seus subcomandos devem possuir entre 2 e 9 caracteres

É recomendado que se utilize nomes curtos nos utilitários e seus subcomandos, pois nomes muitos longos tornam a digitação mais demorada e mais suscetível a erros de digitação. A recomendação do manual do *POSIX Utility Conventions - POSIX.1-2017*, sugere que o nome do utilitário deve possuir entre 2 e 9 caracteres.

C.1.6 O nome do utilitário e seus subcomandos devem ser escritos em minúsculo e deve conter somente dígitos do conjunto de caracteres portátil

De acordo com o manual do *POSIX Utility Conventions - POSIX.1-2017*, o nome dos utilitários e dos subcomandos devem ser compostos por caracteres minúsculos do conjunto portátil. Essa recomendação é baseada na especificação do *POSIX.1-2017*, onde é definido que todo conjunto de carácter de sistemas POSIX devem conter os caracteres do

conjunto portátil. Dessa maneira, os CLIs que seguem esta boa prática são mais portáveis entre sistemas POSIX. A tabela com todos os caracteres do conjunto portátil está disponível no Apêndice G.

C.1.7 A ordem relativa dos sinalizadores não devem importar

De acordo com o manual do *POSIX Utility Conventions - POSIX.1-2017*, por motivos de usabilidade, a ordem que os usuários digitam os diversos sinalizadores durante a utilização de um determinado recurso da CLI não deve importar.

Um exemplo de ferramenta que segue esta boa prática é o CLI *docker-compose*, que permite a definição de diversos sinalizadores independentes de sua ordem no comando. Tal comportamento pode ser visto no Código 8.

```
# 3 comandos com sinalizadores em ordem diferentes que executam a mesma tarefa
$ docker-compose up --no-color --no-build --no-start worker-container
$ docker-compose up --no-build --no-color --no-start worker-container
$ docker-compose up --no-start --no-build --no-color  worker-container
```

Código 8: Exemplo de situação onde a ordem dos sinalizadores não impacta na execução do comando

C.2 Boas práticas de ajuda ao usuário

As boas práticas de ajuda ao usuário é o agrupamento de todas as recomendações e boas práticas que ajudam o usuário a atingir um determinado objetivo.

C.2.1 Disponibilize um texto de ajuda quando os sinalizadores `-h` e `--help` forem passados

A CLI deve implementar os sinalizadores `-h` e `--help`, disponibilizando uma documentação resumida quando tais sinalizadores estiverem presentes.

Além da implementação desses dois sinalizadores, a CLI deve suportar a utilização de tais parâmetros em qualquer parte do comando, não executando qualquer outra ação além da impressão da documentação.

O CLI do *git* é um exemplo de software que adota esta boa prática, conforme mostra o Código 9. Como pode ser visto, é possível utilizar o sinalizador `--help` em qualquer parte do comando, e mesmo assim nenhuma ação é executada além da apresentação da documentação resumida do comando em questão.

```
# Mostra a documentação resumida do comando "git add"
$ git add . --help
usage: git add [<options>] [--] <pathspec>...

    -n, --dry-run         dry run
    -v, --verbose         be verbose
     ...

# Mostra a documentação resumida do comando "git add"
$ git add --help .
GIT-ADD

NAME
       git-add - Add file contents to the index

SYNOPSIS
       git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i]
                 [--patch | -p] [--edit | -e] [--[no-]all | --[no-]ignore-removal |
                 [--update | -u]] [--intent-to-add | -N] [--refresh] [--ignore-errors]
                 [--ignore-missing] [--renormalize] [--chmod=(+|-)x]
                 [--pathspec-from-file=<file> [--pathspec-file-nul]] [--]
                 [<pathspec>...]
       ...

# Mostra a documentação resumida da ferramenta "git"
$ git --help add .
GIT-ADD

NAME
       git-add - Add file contents to the index

SYNOPSIS
       git add [--verbose | -v] [--dry-run | -n] [--force | -f] [--interactive | -i]
                 [--patch | -p] [--edit | -e] [--[no-]all | --[no-]ignore-removal |
                 [--update | -u]] [--intent-to-add | -N] [--refresh] [--ignore-errors]
                 [--ignore-missing] [--renormalize] [--chmod=(+|-)x]
                 [--pathspec-from-file=<file> [--pathspec-file-nul]] [--]
                 [<pathspec>...]
       ...
```

Código 9: Comandos que abrem a página de ajuda do comando `git add`

C.2.2 Na documentação resumida mostre somente as *flags* e comandos mais usados

É comum que a CLI de ferramentas muito complexas possuam dezenas de *flags* e comandos que podem ser agrupados e resultarem em incontáveis combinações. Porém, é fortemente recomendado que na documentação resumida somente sejam apresentados os comandos e as *flags* mais utilizadas, pois, caso contrário, as informações apresentadas irão desnortear o usuário, invés de ajudá-lo.

Um exemplo da adoção desta boa prática é a CLI do `git`, que em sua documentação mostra somente os comandos mais utilizados e os argumentos e *flags* de cada subcomando

quando o usuário solicita explicitamente, conforme mostra o Código 10.

```
$ git --help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path]
           [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager]
           [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>]
           [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone             Clone a repository into a new directory
   init              Create an empty Git repository or
                     reinitialize an existing one

work on the current change (see also: git help everyday)
   add               Add file contents to the index
   mv                Move or rename a file, a directory, or a
                     symlink
   restore           Restore working tree files
   rm                Remove files from the working tree and
                     from the index
   sparse-checkout   Initialize and modify the sparse-checkout
```

Código 10: Resultado do comando `git --help`

C.2.3 Se o usuário fez algo errado, tente adivinhar o que ele quis dizer

É comum que usuários durante a utilização das CLIs cometam erros de digitação. Dessa forma, uma maneira de melhorar a experiência de uso da ferramenta é a sugestão de comandos parecidos.

Existem várias maneiras de implementar essa funcionalidade. Uma das maneiras é por meio de algoritmos de comparação de distâncias entre *strings* (LEVENSHTEIN, 1965), onde é possível encontrar o comando válido mais semelhante ao que o usuário digitou, e assim sugeri-lo.

Geralmente, as bibliotecas de análise de argumento de linha de comando já possuem essa funcionalidade, sendo assim mais um motivo para utilizá-las.

Um exemplo de utilização desta boa prática é a ferramenta `git`, conforme mostra o Código 11.

```
$ git stat
git: 'stat' is not a git command. See 'git --help'.

The most similar commands are
        status
        stage
        stash


$ git add --ignor
error: ambiguous option: ignor (could be --ignore-errors
       or --ignore-missing)
```

Código 11: Sugestões da ferramenta `git` quando o usuário comete um erro

C.3 Boas práticas de tratamento de erros

As boas práticas de tratamento de erros é o agrupamento de todas as recomendações e boas práticas relacionadas com cenários onde ocorre um erro e é preciso realizar uma determinada rotina para recuperar o estado anterior ou sinalizar o erro.

C.3.1 Trate os erros e emita uma mensagem significativa

É esperado que os usuários utilizem a ferramenta de modo incorreto. Dessa forma, para melhorar a usabilidade e reduzir a curva de aprendizado da ferramenta, é importante que seja emitida uma mensagem significativa quando um erro de execução acontece.

Uma situação de exemplo onde esta boa prática pode ser aplicada é em cenários onde envolvem operações em arquivos. Uma vez que existe a possibilidade de falha por conta das permissões de acesso ao arquivo, ou por conta da inexistência do arquivo, a ferramenta deve evitar a impressão de uma mensagem de erro genérica do tipo: *[Errno 2] No such file or directory* ou *[Errno 3] Permission denied*, conforme é exemplificado no Código 12. Invés disso, a ferramenta deve tratar o erro e emitir uma mensagem mais significativa para o usuário, e potencialmente uma ação que resolveria o problema, conforme mostra o Código 13.

```
$ fake-cli --output=unauthorized-file.txt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
io.UnsupportedOperation: not writable
```

Código 12: Exemplo de erro não tratado

C.3.2 Disponibilize um canal para envio de relatórios de *bugs*

É esperado que durante a utilização do software os usuários se deparem com problemas na ferramenta. Dessa forma, é uma boa prática disponibilizar um canal onde os

```
$ fake-cli --output=unauthorized-file.txt
ERROR: Could not write to `unauthorized-file.txt` file.
You may need to make it writable by running `chmod +w unauthorized-file.txt`
```

Código 13: Exemplo de tratamento de erro

usuários possam comunicar tais situações, para que assim os problemas sejam evidenciados e corrigidos nas próximas versões do software.

Geralmente tais canais de comunicação são divulgados no final da página principal do manual da ferramenta, mas há casos onde durante o tratamento de uma exceção é apresentado um canal de envio de relatos de bug. O Código 14 apresenta o canal de relato de bugs do manual da ferramenta *git*.

```
$ man git
...
REPORTING BUGS
       Report bugs to the Git mailing list <git@vger.kernel.org[6]> where the
       development and maintenance is primarily done. You do not have to be
       subscribed to the list to send a message there. See the list archive
       at https://lore.kernel.org/git for previous bug reports and
       other discussions.

       Issues which are security relevant should be disclosed privately to
       the Git Security mailing list <git-security@googlegroups.com[7]>.
...
```

Código 14: Canal de relato de bugs da ferramenta *git*

C.4 Boas práticas na utilização de argumentos e sinalizadores

As boas práticas na utilização de argumentos e sinalizadores é o agrupamento de todas as recomendações relacionadas a obtenção e tratamento de parâmetros. Quando falamos de CLIs, argumentos são os parâmetros posicionais passados para um comando. Enquanto que sinalizadores, também chamados de *flags*, são os parâmetros passados com prefixo de um ou dois hifens. Além do termo cujo prefixo contém hifens, os sinalizadores podem ou não ter um valor associado.

C.4.1 Prefira sinalizadores a argumentos

No desenvolvimento de CLIs que provavelmente serão utilizados em script de automação, é recomendado que os parâmetros sejam recebidos via sinalizadores diante dos argumentos. Essa abordagem, por mais que demande mais caracteres, deixa explícita a intenção do comando, facilitando assim a compreensão do *script* como um todo.

Para casos onde a ferramenta pode ser de uso cotidiano, é recomendado que os parâmetros possam ser obtidos por argumentos e sinalizadores, pois dessa maneira é possível atender aos usuários que visam produtividade na utilização da ferramenta, e aos usuários que buscam auto-documentação nos *scripts*.

Um exemplo de ferramenta que utiliza essa abordagem é a CLI do Apache Kafka[1], onde todos os parâmetros são sinalizadores, conforme mostra o Código 15.

```
$ kafka-topics --describle --topic usage-events \
               --bootstrap-server localhost:9092

$ kafka-console-consumer --topic usage-events \
                         --from-beginning \
                         --bootstrap-server localhost:9092
```

Código 15: Exemplo de CLI que utiliza sinalizadores para todos os parâmetros

C.4.2 Tenha versões curtas e longas de todos os sinalizadores

Uma boa prática na utilização de sinalizadores é sempre disponibilizar duas versões para o mesmo sinalizador, uma versão longa com somente um caractere e outra longa com um termo autoexplicativo.

Essa prática permite que usuários mais experientes possam utilizar a CLI mais eficientemente, com os sinalizadores curtos, e também possibilita a documentação dos parâmetros e suas intenções em scripts bash, por meio dos sinalizadores longos.

Um exemplo de ferramenta que respeita essa recomendação em quase todos os sinalizadores é o utilitário *cp* dos sistemas da família POSIX. Por meio desse utilitário é possível copiar arquivos e diretórios no sistema de arquivos do sistema operacional, conforme mostra o Código 16.

C.4.3 Tenha suporte a coringas (*wildcards*)

Caso a CLI possua alguma funcionalidade que possa ser utilizada em um contexto repetitivo, é recomendado o suporte a argumentos com *wildcards*.

No contexto de sistemas de software, um caractere *wildcards*, ou um caractere coringa, é um tipo de caractere que pode ser interpretado como uma sequência de zero ou muitos caracteres.

Um exemplo de ferramenta com suporte a *wildcards* é o utilitário *cp* que possibilita a cópia de arquivos de diretórios, conforme mostra no Código 17.

[1] Apache Kafka é uma plataforma open-source de processamento de *streams*. O projeto tem como objetivo fornecer uma plataforma unificada, de alta capacidade e baixa latência para tratamento de dados em tempo real.

```
$ cp --help
Usage: cp [OPTION]... [-T] SOURCE DEST
  or:  cp [OPTION]... SOURCE... DIRECTORY
  or:  cp [OPTION]... -t DIRECTORY SOURCE...
Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.
  -a, --archive
  -f, --force
  -l, --link
  -L, --dereference
  -n, --no-clobber
  -P, --no-dereference
  -R, -r, --recursive
...
```

Código 16: Mensagem de ajuda do utilitário *cp*

```
$ cp ./*.txt ./all-text
```

Código 17: Exemplo de comando CLI com suporte a *wildcards*

C.4.4 Se aplicável, utilize os sinalizadores padrões

É comum que as interfaces de determinadas ferramentas com CLIs possuam semelhanças. Assim, visando uma melhor usabilidade, é recomendado que esses recursos semelhantes tenham interfaces semelhantes, e se possível idêntica. Alguns exemplos de recursos comuns em diversas ferramentas com CLIs são: a impressão do texto de ajuda; o direcionamento da saída para um determinado arquivo; a execução forçada de uma determinada ação; a execução verbosa; e outros. Dessa forma, sempre que aplicável utilize os mesmos sinalizadores para situações onde a semântica do comando é semelhante, pois dessa maneira a utilização da ferramenta se torna mais intuitiva.

Alguns sinalizadores comumente utilizados, e suas intenções, são apresentados na Tabela 9.

C.4.5 Toda entrada via *prompt* deve ter um argumento ou sinalizador associado

Para comandos que durante a execução precisa de alguma interação do usuário, é recomendado a disponibilização de sinalizadores que suprimam a entrada via *prompt*. Essa boa prática simplifica a utilização da CLI em scripts bash.

Um exemplo de CLI que precisa da interação do usuário é o comando *apt-get dist-upgrade*. Por meio desse comando é possível realizar a recuperação de pacotes, a obtenção de informações de fontes autenticadas, além de realizar instalações, atualizações e remoções de pacotes desatualizados, junto com suas dependências. Como pode ser visto no

Tabela 9 – Sinalizadores e seus significados mais comumente usados em CLIs

Sinalizador curto	Sinalizdor longo	Significado associado
-a	--all	Tudo
-d	--debug	Mostra a saída de depuração
-f	--force	Força uma ação
	--json	Exibe a saída no formato JSON
-h	--help	Exibe o texto de ajuda
	--no-input	Desabilita *prompts*
-o	--output	Define o arquivo de saída
-p	--port	Define a porta de rede
-q	--quiet	Reduz ou desliga a saída
	--verbose	Exibe todos os *logs* de execução
-u	--user	Referencia um usuário
-v	--version	Exibe a versão da ferramenta
-y		Confirmação de uma ação

Código 18, o comando exige a interação do usuário durante sua execução, sendo necessário confirmar a ação de atualização de pacote. Esse utilitário segue esta boa prática, uma vez que disponibiliza o sinalizador *-y* que suprime a interação via *prompt* e realiza a ação do comando sem a reafirmação do usuário, conforme pode ser visto no Código 19.

```
$ apt-get dist-upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages will be upgraded:
  google-chrome-stable
1 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 90,3 MB of archives.
Do you want to continue? [Y/n]
```

Código 18: Exemplo de comando CLI que precisa de dupla confirmação

```
$ apt-get dist-upgrade -y
Get:1 http://dl.google.com/linux/chrome/deb stable/main amd64
      google-chrome-stable amd64 94.0.4606.71-1 [90,3 MB]
      Fetched 90,3 MB in 150s (6.036 kB/s)
Preparing to unpack
      .../11-google-chrome-stable_94.0.4606.71-1_amd64.deb ...
Unpacking
      google-chrome-stable (94.0.4606.71-1) over
      (94.0.4606.61-1) ...
Setting up
      google-chrome-stable (94.0.4606.71-1) ...
...
```

Código 19: Exemplo de dupla confirmação por sinalizador

C.4.6 Confirme antes de fazer qualquer ação perigosa

Uma boa prática de ferramentas CLI que realiza ações irreversíveis é a dupla confirmação de intenção. Essa prática consiste em perguntar ao usuário se ele realmente deseja realizar determinada ação, antes da execução do comando. Geralmente essa confirmação é realizada via *prompt*, mas também é possível realizá-la via sinalizadores como *-f* e o *–force*.

A definição de qual ação é ou não perigosa é subjetiva, e cabe ao avaliador definir quais ações devem ser duplamente confirmadas.

Leve em consideração que existe maneiras não óbvias de modificar e destruir recursos acidentalmente. Um exemplo é a modificação de um arquivo de configuração de *deploy*, como o *docker-compose.yml*[2], em que é possível alterar o número de réplicas de uma aplicação de 10 para 1. A simples mudança no arquivo não realiza ação, mas a próxima execução da ferramenta *docker-compose* realizará a leitura do arquivo e resultará na exclusão de 9 instâncias de uma aplicação.

Um exemplo de ferramenta que utiliza esta boa prática é o `git push` , em que é necessário uma dupla confirmação de intenção quando detectada uma ação que resultará em perda irreversível de dados, conforme pode ser visto nos Códigos 20 e 21.

```
$ git push origin master
To github.com:durvalcarvalho/testing-github-codespaces.git
 ! [rejected]        main -> origin/main (non-fast-forward)
error: failed to push some refs to
'git@github.com:durvalcarvalho/testing-github-codespaces.git'
hint: Updates were rejected because the tip of your current
      branch is behind hint: its remote counterpart.
Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in
      'git push --help' for details.
```

Código 20: Exemplo de comando que recusa a ação perigosa do usuário

C.4.7 Não leia credenciais diretamente dos argumentos

Por motivos de segurança, é recomendado que credenciais e informações confidenciais não sejam obtidos por meio de argumentos e sinalizadores. Por mais que essa recomendação contraponha a boa prática C.4.5, o seu não cumprimento pode resultar em falhas graves de segurança.

[2] O arquivo *docker-compose.yml* é o arquivo de configuração padrão da ferramenta *docker-compose*. Por sua vez, *docker-compose* é uma aplicação CLI utilizada para definir e executar aplicações *multi-contêiner* baseadas em Docker. Já o *Docker*, é uma aplicação que utiliza virtualização no nível do sistema operacional para fornecer softwares em pacotes, chamados contêiners

```
$ git push origin master --force
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 269 bytes | 269.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1),
        completed with 1 local object.
To github.com:durvalcarvalho/testing-github-codespaces.git
 + cdcb890...4764e63 master -> origin/master (forced update)
```

Código 21: Exemplo de uso do sinalizador *--force* para confirmar uma ação perigosa

Essa recomendação é baseada no fato de que sistemas POSIX armazenam informações associadas ao histórico de comandos executados e de comandos em execução. Dessa forma, caso algum dado confidencial faça parte do comando, tal informação ficará armazenada e poderá ser consultada posteriormente.

Para compreender tal recomendação, suponha a existência de um utilitário chamado *fake-cli*, onde é possível realizar login passando como parâmetro as credenciais de um usuário, tal situação pode ser vista no Código 22.

Uma vez que esse suposto utilitário obteve informações confidenciais via argumentos, um usuário malicioso poderá obter os dados confidenciais utilizando a ferramenta *history* dos sistemas POSIX, que por sua vez lista o histórico de comandos executados naquela sessão do terminal, conforme mostra o Código 23.

Outra maneira de obter o dado confidencial é por meio do utilitário *ps* que é capaz de listar informações sobre os processos em execução, inclusive o comando que iniciou o processo. Conforme pode ser visto no Código 24, é possível obter as credenciais do usuário.

```
$ python fake-cli.py login --user=admin, --password=bitcoin
```

Código 22: Exemplo de CLI que ler credenciais pelos argumentos

```
$ history
 2000  cd /tmp
 2001  touch fake-cli.py
 2002  nano fake-cli.py
 2002  python3 fake.cli.py login --user=admin --password=bitcoin
 2003  history
```

Código 23: Credencial salva no histórico de comandos

Dessa forma, para evitar vazamento de dados críticos, esta boa prática recomenda que informações confidenciais sejam obtidas por meio de arquivos, variáveis de ambiente ou por *prompts*.

```
$ ps -af
UID      PID      PPID    C   STIME   TTY      TIME       CMD
durval   123349   115597  0   09:06   pts/2    00:00:00   python3 fake-cli.py
                                                          login --user=admin
                                                          --password=bitcoin
durval   124260   34739   0   09:07   pts/0    00:00:00   ps -af
```

Código 24: Credencial visível na descrição dos processos em execução

Um exemplo de ferramenta que implementa esta boa prática é a CLI do PostgreSQL, o psql. Utilizando esta ferramenta é possível realizar *login* passando as credenciais por *prompt*, por arquivo ou por meio de uma variável de ambiente, conforme é mostrado nos Códigos 25, 26 e 27, respectivamente.

```
# Opção 1: Prompt
# O sinalizador --password força o prompt de senha,
# que por sua vez desabilita o echo
$  psql --username=postgres --password
Password:
psql (13.4 (Debian 13.4-1.pgdg100+1))
Type "help" for help.

postgres=#
```

Código 25: Exemplo de entrada de credenciais via prompt

```
# Opção 2: Variáveis de ambiente

# A ferramenta de `echo` mostra no terminal o valor
# associado à variável de ambiente POSTGRES_PASSWORD
$ echo $POSTGRES_PASSWORD
bitcoin

# Usar o comando `psql` sem o sinalizador --password
# utiliza como senha o valor definido na variável de
# ambiente POSTGRES_PASSWORD
$ psql --username=postgres
psql (13.4 (Debian 13.4-1.pgdg100+1))
Type "help" for help.

postgres=#
```

Código 26: Exemplo de entrada de credenciais via variável ambiente

C.5 Boas práticas na interatividade

As boas práticas na interatividade é o agrupamento de todas as recomendações e boas práticas relacionadas a rotinas onde o usuário interage com o programa em execução.

```
# Opção 3: Utiliza um arquivo

# catA ferramenta `tool` mostra no terminal conteúdo
# do arquivo localizado em /root/.pgpass
$ cat /root/.pgpass
localhost:5432:postgres:admin:bitcoin

# O CLI `echo` mostra no terminal o valor associado
# à variável de ambiente `PGPASSFILE`
$ echo $PGPASSFILE
/root/.pgpass

# A execução do comando `psql` sem nenhum argumento
# utiliza por padrão o conteúdo do arquivo localizado
# no caminho definido pela variável de ambiente `PGPASSFILE`
$  psql
psql (13.4 (Debian 13.4-1.pgdg100+1))
Type "help" for help.

postgres=#
```

Código 27: Exemplo de entrada de credenciais via arquivo

C.5.1 Somente habilite o *prompt* se `stdin` estiver associado a um terminal interativo

Conforme foi explicado na Seção A.2, é possível utilizar os *shell scripts* para realizar uma série de comandos CLIs. E conforme explicado na Seção C.1.2 e C.1.3, é esperado que as ferramentas CLIs em ambiente UNIX sejam utilizadas em conjunto, na qual a saída de um comando é a entrada no próximo comando. Sendo assim, a presença de *prompts* nessa cadeia de execução pode prejudicar a utilização em conjunto das ferramentas.

Dessa forma, uma boa prática no desenvolvimento de CLIs é a utilização de *prompt* somente caso a entrada padrão, o `stdin`, esteja associado a um terminal.

Essa verificação pode ser feita facilmente por meio de funções que a maioria das linguagens de programação disponibilizam, conforme mostram os Códigos 28 e 29.

C.5.2 Desligue o feedback (echo) de digitação enquanto o usuário digita dados confidenciais

Conforme explicado na Seção C.4.7, é recomendado o uso de *prompt* durante a leitura de dados confidenciais. Porém, o comportamento padrão do *line discipline* associado ao *shell* e ao terminal é ecoar todos os caracteres digitados.

Essa boa prática recomenda que durante o tratamento das entradas de dados confidenciais, esse comportamento padrão seja alterado para não mostrar o que está sendo digitado, e ao término, o comportamento padrão seja retornado.

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  if (isatty(fileno(stdin)))
    printf("stdin is connected to a terminal");
  else
    printf("stdin is NOT connected to a terminal");
  return 0;
}
```

Código 28: Exemplo em C de como detectar se a entrada padrão está associada a um terminal interativo

```python
import sys

if sys.stdin.isatty():
    printf("stdin is connected to a terminal");
else:
    printf("stdin is NOT connected to a terminal");
```

Código 29: Exemplo em Python de como detectar se a entrada padrão está associada a um terminal interativo

A maioria das linguagens de programação possuem funções auxiliares para realizar esse procedimento de obter uma entrada de usuário sem feedback de digitação, conforme mostram os Códigos 30 e 31.

```python
from getpass import getpass

password = getpass('Digite a senha: ')
```

Código 30: Exemplo em Python de obtenção de credenciais sem ecoamento no terminal

```c
#include <unistd.h>

char *password = getpass("Digite a senha: ");
```

Código 31: Exemplo em C de obtenção de credenciais sem ecoamento no terminal

C.5.3 Permita o usuário interromper a execução

Conforme explicado na Seção A.3.2, o módulo de *line discipline* realiza rotinas especiais quando determinadas sequências de caracteres são acionadas. Uma das rotinas mais importantes quando tratamos de CLIs é a emissão de sinais quando determinadas teclas são digitadas em conjunto.

Tabela 10 – Principais sinais, seus comportamentos e as teclas associadas

Sinal de sistema	**Comportamento padrão associado**	**Sequência de teclas associadas**
`SIGINT`	Abortar a execução do programa	Ctrl + C
SIGQUIT	Despejo imediato da memória alocada (*core dump*)	Ctrl + \
SIGSTOP	Suspensão da execução	Ctrl + Z
SIGINFO	Mostra informação sobre o comando em execução	Ctrl + T

A Tabela 10 apresenta os principais sinais, seus comportamentos padrões associados e as sequências de teclas que acionam sua emissão.

É fortemente recomendado que não se altere o comportamento padrão do tratamento destes sinais. Um exemplo de CLI que não segue essa recomendação e que frequentemente causa confusão em novos usuários é a ferramenta de edição de arquivos *vim*, que sobrescreve a sequência de teclas *CTRL + C* comumente usada para abortar a execução de programas, causando confusão em usuários que não conseguem encerrar o utilitário.

Uma forma de atender a esta boa prática e ao mesmo tempo sobrescrever o comportamento padrão do tratamento de sinais, é utilizar rotinas de saídas graciosas (*graceful shutdown*), em que o programa é notificado que deve finalizar sua execução assim que atingir um estágio de saída segura.

Um exemplo de ferramenta que utiliza essa abordagem do *graceful shutdown* é a CLI do *docker-compose*. Quando um sinal do tipo `SIGINT` é emitido, a CLI trata o sinal e realiza as operações necessárias para desligar o contêiner sem perda de dados, possibilitando ao usuário emitir um novo sinal do tipo `SIGINT` para saída imediata, conforme mostram os Códigos 32 e 33.

```
$ docker-compose up workers
workers is up-to-date
Starting workers ... done
Attaching to workers
workers     | Processing events [1 / 200]
workers     | Processing events [2 / 200]
workers     | Processing events [3 / 200]

# teclas ctrl+c pressionadas
$ ^C
Gracefully stopping... (press Ctrl+C again to force)
Stopping workers ... done
```

Código 32: Exemplo de CLI que realiza uma rotina de execução quando recebe o sinal `SIGINT`

```
$ docker-compose up workers
workers is up-to-date
Starting workers ... done
Attaching to workers
workers     | Processing events [1 / 200]
workers     | Processing events [2 / 200]
workers     | Processing events [3 / 200]

# teclas ctrl+c pressionadas
$ ^C

Gracefully stopping... (press Ctrl+C again to force)
Stopping workers ...

# teclas ctrl+c pressionadas novamente
$ ^C
Killing workers  ... done
```

Código 33: Exemplo de CLI que possibilita o encerramento forçado ao enviar dois sinais do tipo `SIGINT`

C.6 Boas práticas no uso de subcomandos

Este tópico é o agrupamento de todas as recomendações e boas práticas relacionadas a implementação de subcomandos. Em um CLI é comum que existam cenários em que um conjunto de ferramentas realize tarefas correlacionadas. Dessa forma, uma maneira de facilitar a usabilidade dos usuários é o agrupamento de tais ferramentas por meio de subcomandos. Um exemplo de uso dessa abordagem é a CLI da ferramenta *git*, onde diversas ferramentas como o *git-add*, *git-remove*, *git-stash*, *git-commit* são agrupadas sob o utilitário *git* na forma de subcomandos.

Além da melhoria de usabilidade, essa abordagem é útil para utilização compartilhada de definições globais, textos de ajuda e configurações.

C.6.1 Seja consistente em todos os subcomandos

Se aplicável, utilize o mesmo conjunto, ou subconjunto de sinalizadores e argumentos, e tenha a formatação de saída semelhante.

Um exemplo de ferramenta que possui uma interface semelhante entre os subcomandos é a CLI do *docker-compose*, onde o primeiro argumento dos subcomandos *start*, *stop* e *pause* é sempre o nome do recurso, conforme pode ser visto no Código 34.

```
$ docker-compose start workers_service
$ docker-compose stop workers_service
$ docker-compose pause workers_service
```

Código 34: Exemplo de subcomandos com interface parecida

C.6.2 Use nomes consistentes para vários níveis de subcomando

Em um software complexo é comum que existam várias operações e recursos que podem ser realizados e modificados, dessa forma, a interface de linha de comando tende a aumentar sua complexidade, diante da presença de vários subcomandos e parâmetros que podem ser utilizados. Diante disso, é recomendado que a interface desses vários subcomandos e os parâmetros utilizados sejam consistentes entre si, pois dessa maneira a usabilidade do software se torna mais intuitiva.

Um exemplo de padronização que deve ser adotada entre subcomandos é o padrão *verbo-ação* e *nome-objeto*, onde o subcomando recebe tais argumentos em uma determinada ordem. Por exemplo, o CLI *docker-compose* utiliza o padrão onde o *verbo-ação* sempre precede o *nome-objeto*, como por exemplo no comando `docker-compose stop worker-container`.

Independentemente do padrão que seja escolhido para o contexto da ferramenta, é recomendado que tal padrão seja seguido em todos os subcomandos.

C.6.3 Seja responsível

Se o comando executado pelo usuário for uma operação demorada, é recomendado que se realize escritas no terminal apresentando os resultados parciais, pois caso contrário, o usuário poderá ter a impressão que a ferramenta parou de funcionar.

Dessa forma, Prasad et al. (2018) recomenda que operações muito longas atualizem o usuário sobre o processamento em intervalos máximos de 100 milissegundos.

APÊNDICE D – *Checklist*

Este apêndice apresenta o *checklist* que foi elaborado com base nos tópicos explicados no manual de boas práticas aplicáveis ao software *cp-tools*, disponível no Apêndice C.

Checklist de boas práticas de Convenções para Interfaces de Linha de Comando				
Software Analisado: cp-tools (https://github.com/edsomjr/competitive-problems-tools)				
Agrupamento	**Item**	**Nome**	**Análise**	**Comentário**
Boas Práticas Gerais	1.1	Use uma biblioteca de análise de argumento de linha de comando		
	1.2	Retorne zero em caso de sucesso		
	1.3	Utilize por padrão os *streams* de dados `stdout` e `stderr`		
	1.4	Disponibilize a versão e a licença do software quando o sinalizador `--version` for utilizado		
	1.5	O nome do utilitário e seus subcomandos devem possuir entre 2 e 9 caracteres		
	1.6	O nome do utilitário e seus subcomandos devem ser escritos em minúsculo e deve conter somente dígitos do conjunto de caracteres portátil		
	1.7	A ordem relativa dos sinalizadores não devem importar		

Boas práticas de ajuda ao usuário	2.1	Disponibilize um texto de ajuda quando os sinalizadores -h e --help forem passados		
	2.2	Na documentação resumida mostre somente as flags e comandos mais usadas		
	2.3	Se o usuário fez algo errado, tente adivinhar o que ele quis dizer		
Boas práticas de tratamento de erros	3.1	Trate os erros e emita uma mensagem significativa		
	3.2	Disponibilize um canal para envio de relatórios de *bugs*		
Boas práticas na utilização de argumentos e sinalizadores	4.1	Prefira sinalizadores a argumentos		
	4.2	Tenha versões curtas e longas de todos os sinalizadores		
	4.3	Tenha suporte a coringas (*wildcards*)		
	4.4	Se aplicável, utilize os sinalizadores padrões		
	4.5	Toda entrada via prompt deve ter um argumento ou sinalizador associado		
	4.6	Confirme antes de fazer qualquer ação perigosa		
	4.7	Não leia credenciais diretamente dos argumentos		
Interatividade	5.1	Somente habilite o *prompt* se `stdin` estiver associado a um terminal interativo		

	5.2	Desligue o feedback (echo) de digitação enquanto o usuário digita dados confidenciais		
	5.3	Permita o usuário interromper a execução		
Interatividade	6.1	Seja consistente em todos os subcomandos		
	6.2	Use nomes consistentes para vários níveis de subcomando		
	6.3	Seja Responsível		

APÊNDICE E – *Checklist* após inspeção

Este apêndice apresenta o *checklist* preenchido com as anotações do avaliador da interface de linha de comando do software *cp-tools*.

Checklist de boas práticas de Convenções para Interfaces de Linha de Comando				
Software Analisado: cp-tools (https://github.com/edsomjr/competitive-problems-tools)				
Agrupamento	**Item**	**Nome**	**Análise**	**Comentário**
Boas Práticas Gerais	1.1	Use uma biblioteca de análise de argumento de linha de comando	Não Atendido	É utilizada a biblioteca 'getopt', porém, há partes do código que realizam modificações diretas nas variáveis argc e argv
	1.2	Retorne zero em caso de sucesso	Atendido	
	1.3	Utilize por padrão os *streams* de dados stdout e stderr	Atendido	
	1.4	Disponibilize a versão e a licença do software quando o sinalizador --version for utilizado	Atendido	
	1.5	O nome do utilitário e seus subcomandos devem possuir entre 2 e 9 caracteres	Atendido	
	1.6	O nome do utilitário e seus subcomandos devem ser escritos em minúsculo e conter somente dígitos do conjunto de caracteres portátil	Atendido	

	1.7	A ordem relativa dos sinalizadores não devem importar	Atendido	
Boas práticas de ajuda ao usuário	2.1	Disponibilize um texto de ajuda quando os sinalizadores -h e --help forem passados	Não Atendido	Há situações onde os sinalizadores de ajuda não tem maior prioridade de execução
	2.2	Na documentação resumida mostre somente as flags e comandos mais usadas	Atendido	
	2.3	Se o usuário fez algo errado, tente adivinhar o que ele quis dizer	Não Atendido	Não há sugestões para situações de erros
Boas práticas de tratamento de erros	3.1	Trate os erros e emita uma mensagem significativa	Não Atendido	As mensagens de erros são muito técnicas
	3.2	Disponibilize um canal para envio de relatórios de *bugs*	Não Atendido	Não foi encontrado nenhum canal para envio de relatos de *bugs*
Boas práticas na utilização de argumentos e sinalizadores	4.1	Prefira sinalizadores a argumentos	Atendido	
	4.2	Tenha versões curtas e longas de todos os sinalizadores	Não Atendido	Há sinalizadores que possuem somente versões longas
	4.3	Tenha suporte a coringas (*wildcards*)	Não Atendido	Não há sugestões para situações de erros
	4.4	Se aplicável, utilize os sinalizadores padrões	Atendido	
	4.5	Toda entrada via prompt deve ter um argumento ou sinalizador associado	Atendido	
	4.6	Confirme antes de fazer qualquer ação perigosa	Atendido	

	4.7	Não leia credenciais diretamente dos argumentos	Não Atendido	Há situações onde os sinalizadores de ajuda não tem maior prioridade de execução
Interatividade	5.1	Somente habilite o *prompt* se `stdin` estiver associado a um terminal interativo	Atendido	
	5.2	Desligue o feedback (echo) de digitação enquanto o usuário digita dados confidenciais	Atendido	
	5.3	Permita o usuário interromper a execução	Atendido	Não há sugestões para situações de erros
Interatividade	6.1	Seja consistente em todos os subcomandos	Atendido	
	6.2	Use nomes consistentes para vários níveis de subcomando	Atendido	
	6.3	Seja Responsível	Não Atendido	

APÊNDICE F – Relatório de inspeção da interface

Este apêndice contém o relatório detalhado da avaliação da interface de linha de comando do software *cp-tools*.

F.1 Boas práticas gerais

F.1.1 Use uma biblioteca de análise de argumento de linha de comando

O software *cp-tools* utiliza a função nativa da linguagem C `getopt_long` disponível na biblioteca `getopt.h`. Essa função automatiza algumas das tarefas envolvidas na leitura de opções e sinalizadores, mas carece de várias funcionalidades avançadas que geralmente são obtidas utilizando uma biblioteca de análise de argumentos de linha de comando, como por exemplo, a geração de documentação automática, sugestão para cenários de erros de digitação e estilização das mensagens impressas no terminal.

Dessa forma, este tópico do *checklist* é considerado não atendido, uma vez que o CLI analisado carece de recursos que seriam facilmente obtidos caso se utilizasse uma das biblioteca sugeridas na Tabela 8

F.1.2 Retorne zero em caso de sucesso

A análise deste tópico do *checklist* consistiu na execução de cada um dos comandos do software *cp-tools* e na análise do código retornado. Alguns dos resultados obtidos podem ser vistos no Código 35.

Foi evidenciado que o código de retorno das rotinas bem sucedidas sempre era o valor zero, enquanto que as rotinas, onde ocorria um erro ou havia uma má utilização do comando, era retornado um código diferente de zero. Dessa forma, este tópico do *checklist* é considerado atendido.

F.1.3 Utilize por padrão os `streams` de dados `stdout` e `stderr`

Conforme é mostrado no Código 35, foi evidenciado que os comandos utilizam por padrão os *stream* de dados *stdout* e *stderr*. Além disso, alguns comandos disponibilizam sinalizadores para direcionar o fluxo de dados para um arquivo específico. Dessa forma, este tópico do *checklist* é considerado atendido.

```
$ cp-tools init
Initializing directory '.' ...
Ok!

$ echo $?
0

$ cp-tools clean
Cleaning autogenerated files...
Warning! No autogenerated files found

$ echo $?
0

$ cp-tools genboca
Ok! File 'problem.pdf' generated.
Failed! filesystem error: cannot copy: Is a directory
        [problem.pdf] [.cp-build/boca/description/]

$ echo $?
211

$ cp-tools judge
Usage: cp-tools problem judge solution.[cpp|c|java|py]

echo $?
255
```

Código 35: Alguns dos resultados da análise do tópico "Retorne zero em caso de sucesso" do *checklist*

```
$ wc file.pdf
  wc: file.pdf: No such file or directory

$ cp-tools gentex --output=file.tex

$ wc file.pdf
  308  1122 40766 file.pdf
```

Código 36: Direcionamento de fluxo de dados para um arquivo

F.1.4 Disponibilize a versão e a licença do software quando o sinalizador `--version` for utilizado

Conforme é mostrado no Código 37, o software *cp-tools* apresenta a versão e a licença da ferramenta quando é utilizado o sinalizador `--version`. Desse modo, esse tópico do *checklist* é considerado atendido.

```
$ cp-tools --version
cp-tools 0.6.0-r7
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Edson Alves.
```

Código 37: Mensagem contendo a versão e a licença do software `cp-tools`

Tabela 13 – Comandos e subcomandos presentes na ferramenta *cp-tools*

Comando	Quantidade de caracteres
init	4
check	5
clean	5
judge	5
gentex	6
genpdf	6
genboca	7
polygon	7

F.1.5 O nome do utilitário e seus subcomandos devem possuir entre 2 e 9 caracteres

Conforme é mostrado na Tabela 13, nenhum dos comandos e subcomandos presentes na ferramenta *cp-tools* possui a quantidade de caracteres fora do intervalo de 2 a 9 caracteres. Desse modo, esse tópico do *checklist* é considerado atendido.

F.1.6 O nome do utilitário e seus subcomandos devem ser escritos em minúsculo e deve conter somente dígitos do conjunto de caracteres portátil

Conforme é mostrado na Tabela 13, nenhum dos comandos e subcomandos da ferramenta *cp-tools* possui caracteres que não pertencem ao conjunto portátil de caracteres. Desse modo, esse tópico do *checklist* é considerado atendido.

F.1.7 A ordem relativa dos sinalizadores não deve importar

Os comandos que possuem mais de um sinalizador que podem ser usados em conjunto são: `cp-tools gentex`, `cp-tools genpdf` e `cp-tools genboca`. Estes comandos possuem a mesma interface, onde são disponibilizados sinalizadores que alteram os documentos que são gerados.

Para analisar se a ferramenta atende a esta boa prática foram escolhidos os sina-

lizadores `-b`, `-g` e `-t`, que conforme mostra o Código 38, são responsáveis por modificar o rótulo, a linguagem e o template do documento que será gerado.

```
$ cp-tools gentex --help
Usage: cp-tools gentex [-h] [-o outfile] [-c doc_class] [-l list] [-g lang]
                       [-b label] [--no_author] [--no_contest]
Generate a LaTeX file from the problem description. The options are:

    -h              Displays this help.
    --help

    -o              Output file. If omitted, the output will be written on stdout.
    --output

    -b              Defines the problem label. The default value is 'A'.
    --label

    -c              Document class that will be used. The default value
    --class         is 'cp_modern'.

    -g              Problem language. The default value is 'pt_BR'.
    --lang

    -l              Lists all available document classes.
    --list

    -t              Generates LaTeX file for problem's tutorial.
    --tutorial

    --no_author     Omits problem's author.

    --no_contest    Omits problem's contest.
```

Código 38: Mensagem de ajuda impressa quando executado o comando `cp-tools gentex -help`

A rotina de verificação desta boa prática consistiu em executar as 6 possíveis variações de posições dos 3 sinalizadores escolhidos e comparar se o arquivo resultante se manteve o mesmo. Conforme é possível observar no Código 39, nenhuma variação do comando alterou o resultado final. Dessa forma, este tópico do *checklist* é considerado atendido.

```
$ cp-tools gentex -b A          -c cp_classic -g en_US      > outputs/output1.tex
$ cp-tools gentex -b A          -g en_US      -c cp_classic > outputs/output2.tex
$ cp-tools gentex -g en_US      -b A          -c cp_classic > outputs/output3.tex
$ cp-tools gentex -g en_US      -c cp_classic -b A          > outputs/output4.tex
$ cp-tools gentex -c cp_classic -g en_US      -b A          > outputs/output5.tex
$ cp-tools gentex -c cp_classic -b A          -g en_US      > outputs/output6.tex

$ diff --from-file=outputs/output1.tex outputs/*.tex
# A ausência de output do comando diff indica que todos os arquivos são iguais
```

Código 39: Rotina de verificação da boa prática da ordem relativa dos sinalizadores

F.2 Boas práticas de ajuda ao usuário

F.2.1 Disponibilize um texto de ajuda quando os sinalizadores `-h` e `--help` forem passados

Foi evidenciado que os comandos da ferramenta *cp-tools* implementam os sinalizadores `-h` e `--help`, onde é emitida a mensagem de ajuda do respectivo comando. Porém, existem comandos que são executados mesmo quando os sinalizadores de ajuda estão presentes. Conforme mostra o Código 40, a execução do comando `cp-tools check -all -help` executa a rotina do comando `cp-tools check -all`, invés de emitir a mensagem de ajuda ao usuário. Dessa forma, este tópico do *checklist* é considerado não atendido.

```
# Era esperado que o texto de ajuda fosse emitido
$ cp-tools check --all --help
Creating directory .cp-build
Testing the checker (3 tests) ...
Ok!
Creating directory .cp-build
Validating the input files (9 tests) ...
Ok!
Creating directory .cp-build
Testing the validator (2 tests) ...
Ok!
```

Código 40: Exemplo de comando que não deveria ter sido executado

F.2.2 Na documentação resumida mostre somente as *flags* e comandos mais usados

Ficou evidenciado que a documentação impressa quando utilizado os sinalizadores de ajuda é uma versão resumida da documentação existente no manual da ferramenta. Conforme pode ser visto no Código 41, a documentação impressa pelo comando `cp-tools -help` apresenta somente os comandos mais utilizados e uma breve descrição, enquanto que o manual impresso pelo comando `man cp-tools` contém outras informações relevantes para a ferramenta. Dessa forma, este tópico do *checklist* é considerado não atendido.

F.2.3 Se o usuário fez algo errado, tente adivinhar o que ele quis dizer

Essa boa prática não é seguida pela ferramenta *cp-tools*, pois não há tratamento de entradas incorretas, conforme mostra o Código 42. Dessa forma este tópico do *checklist* não é atendido.

```
# Manual da ferramenta
$ man cp-tools
CP-TOOLS(1)               General Commands Manual               CP-TOOLS(1)

NAME
       cp-tools - Format, test and pack competitive programming problems.

SYNOPSIS
       cp-tools [-h] [-v] [init] [check] [genpdf]
                [clean] [judge] [gentex]

DESCRIPTION
       ...

OPTIONS
       ...

BUGS
       ...

AUTHORS
       ...

                         07 November 2020                  CP-TOOLS(1)


# Documentação resumida
$ cp-tools --help
Usage: cp-tools [-h] [-v] action
Format, test and pack competitive programming problems.

    Action              Description

    init                Generates template files on current directory.
    check               Verifies problem files and tools.
    clean               Removes autogenerated files.
    genpdf              Generates a PDF file from the problem description.
    gentex              Generates a LaTeX file from the problem description.
    genboca             Pack the marathon problem for the online judge BOCA.
    judge               Runs a solution against all tests sets.
    polygon             Connects and synchronize with a Polygon account.
```

Código 41: Manual e documentação resumida da ferramenta *cp-tools*

F.3 Boas práticas de tratamento de erros

F.3.1 Trate os erros e emita uma mensagem significativa

Foi evidenciado que os erros emitidos pela ferramenta não possuem mensagem significativa para os usuários finais. Conforme pode ser visto no Código 44, as mensagens de erros emitidas estão relacionadas às rotinas internas do comando e não ao comando em si. Dessa forma, para o usuário final, a mensagem emitida não tem valor significativo e não sugere uma possível correção.

```
# A ferramenta poderia sugerir o comando 'cp-tools check'
$ cp-tools chec
Failed! Invalid action 'chec'

# A ferramenta poderia sugerir o comando 'cp-tools gentex'
$ cp-tools genlatex
Failed! Invalid action 'genlatex'
```

Código 42: Situação onde a ferramenta *cp-tools* poderia realizar uma sugestão

No Código 43 é apresentada uma situação de emissão de uma mensagem de erro associada a uma rotina interna do comando, onde o usuário tenta executar o comando `cp-tools init` em um diretório protegido. Uma possível solução para este cenário de erro, seria a impressão de uma mensagem de maior valor para o usuário, juntamente com uma sugestão de correção, conforme é mostrado no Código 44.

```
$ cp-tools init
Initializing directory '.' ...
Failed! filesystem error: cannot copy:
        Permission denied
        [/usr/local/lib/cp-tools/templates/
         problem-template/] [.]
```

Código 43: Mensagem de erro emitida quando ocorre uma falha na execução do comando `cp-tools init`

```
$ cp-tools init
Initializing directory '.' ...
Failed! Unable to start the project in the current directory.
        Check if your user has write permission in the
        chosen directory.
```

Código 44: Mensagem de erro significativa que poderia ser emitida

Sendo assim, este tópico do *checklist* não é atendido.

F.3.2 Disponibilize um canal para envio de relatórios de *bugs*

Conforme mostra o Código 45, o manual da ferramenta *cp-tools* apresenta o link do repositório do software, e informa que os potenciais *bugs* devem ser relatados via *issues*[1]. Sendo assim, este tópico do *checklist* é atendido.

[1] No contexto do site GitHub, *issues* são anotações, semelhantes a blocos de Post-it®, que são frequentemente usados para anotar funcionalidade desejadas, bugs encontrados e funcionalidade sugeridas

```
# Manual da ferramenta
$ man cp-tools
CP-TOOLS(1)               General Commands Manual               CP-TOOLS(1)
...

BUGS
    Please report any bugs using the Github issue
    tracker: https://github.com/edsomjr/competitive-problems-tools/issues?state=open
...


                          07 November 2020                  CP-TOOLS(1)
```

Código 45: Seção do manual onde é informado o modo de relatar *bugs*

F.4 Boas práticas na utilização de argumentos e sinalizadores

F.4.1 Prefira sinalizadores a argumentos

Atualmente, o *cp-tools* possui 8 comandos em sua CLI, conforme mostra o Código 46. Após analisar a API de cada um dos 8 subcomandos, foi identificado que 7 deles utilizam sinalizadores ao invés de argumentos na entrada de parâmetros. Dessa forma, é possível afirmar que esse tópico do *checklist* é atendido.

```
$ cp-tools -h
Usage: cp-tools [-h] [-v] action
Format, test and pack competitive programming problems.

    Action              Description

    init                Generates template files on current directory.
    check               Verifies problem files and tools.
    clean               Removes autogenerated files.
    genpdf              Generates a PDF file from the problem description.
    gentex              Generates a LaTeX file from the problem description.
    genboca             Pack the marathon problem for the online judge BOCA.
    judge               Runs a solution against all tests sets.
    polygon             Connects and synchronize with a Polygon account.
```

Código 46: Comandos disponíveis no CLI da ferramenta cp-tools

F.4.2 Tenha versões curtas e longas de todos os sinalizadores

A maioria dos sinalizadores implementados pela ferramenta *cp-tools* disponibilizam a versão longa e a versão curta implementada. Dos 26 sinalizadores disponíveis, 18 possuem versões curtas e longas e somente 8 possuem apenas versões longas, conforme é mostrado no Quadro F.1.

Dessa forma, este tópico do *checklist* é considerado atendido, uma vez que a maioria dos sinalizadores possuem ambas as versões e os sinalizadores que possuem somente a

Quadro F.1: Todos os sinalizadores da ferramenta *cp-tools* e suas respectivas versões longas e curtas

Sinalizador	**`Versão longa`**	**`Versão curta`**
all	`--all`	`-a`
checker	`--checker`	`-c`
solutions	`--solutions`	`-s`
tests	`--tests`	`-t`
validator	`--validator`	`-v`
working-dir	`--working-dir`	`-w`
version	`--version`	`-v`
help	`--help`	`-h`
label	`--label`	`-b`
lang	`--lang`	`-g`
list	`--list`	`-l`
class	`--class`	`-c`
output	`--output`	`-o`
tutorial	`--tutorial`	`-t`
key	`--key`	`-k`
secret	`--secret`	`-s`
creds	`--creds`	`-c`
force	`--force`	`-f`
no_author	`--no_author`	
no_contest	`--no_contest`	
c-time-limit	`--c-time-limit`	
cpp-time-limit	`--cpp-time-limit`	
java-time-	`--java-time-limit`	
kt-time-limit	`--kt-time-limit`	
py2-time-limit	`--py2-time-limit`	
py3-time-limit	`--py3-time-limit`	

versão longa não é de uso recorrente.

Durante deste tópico do *checklist*, foi observado que existem dois sinalizadores (*--no_autor*, *--no_contest*) que utilizam o símbolo *underscore (_)* invés do símbolo *dash (-)*. É desejável que esses dois sinalizadores sejam alterados para assim seguir uma única convenção de nomenclatura.

F.4.3 Tenha suporte a coringas (*wildcards*)

Após a análise dos comandos presentes na ferramenta foi evidenciado que esta boa prática somente é aplicável ao comando `cp-tools judge`. Conforme mostra o Código 47, este comando é responsável por compilar, executar e analisar os resultados de uma solução de exemplo do problema. Considerando que um determinado problema de maratona pode apresentar mais de um arquivo de solução, é desejável que o comando `cp-tools judge` suporte *wildcards*, pois, dessa maneira, seria possível analisar vários arquivos em uma única execução. Uma vez que essa funcionalidade é possível e não está presente, este

tópico do *checklist* é considerado não atendido.

```
$ cp-tools judge solutions/solution.cpp
Judging solution 'solutions/solution.cpp'...
-------------------------------------------------------------------------
|    # | Verdict                          |     Time (s) |  Memory (MB) |
-------------------------------------------------------------------------
|    1 | Accepted                         |     0.005256 |        3.254 |
-------------------------------------------------------------------------
|    2 | Accepted                         |     0.004692 |        3.355 |
-------------------------------------------------------------------------
|    3 | Accepted                         |     0.005271 |        3.355 |
-------------------------------------------------------------------------
|    4 | Accepted                         |     0.005468 |        3.062 |
-------------------------------------------------------------------------
|    5 | Accepted                         |     0.005046 |        3.023 |
-------------------------------------------------------------------------
|    6 | Accepted                         |     0.004750 |        3.246 |
-------------------------------------------------------------------------
|    7 | Undefined Error                  |     0.005190 |        3.238 |
-------------------------------------------------------------------------
|    8 | Undefined Error                  |     0.005280 |        3.066 |
-------------------------------------------------------------------------
|    9 | Undefined Error                  |     0.004599 |        3.242 |
-------------------------------------------------------------------------

Verdict:    Undefined Error
Passed:     6
Max time:   0.005468
Max memory: 3.355

# É desjável que este comando suporte vários arquivos usando wildcards
# $ cp-tools judge solutions/*
```

Código 47: Resultado da execução do comando `cp-tools judge`

F.4.4 Se aplicável, utilize os sinalizadores padrões

Conforme foi explicado no Seção C.4.4, existe um conjunto de sinalizadores que são usados com frequência em vários CLIs.

Da lista de sinalizadores frequentes, 5 sinalizadores de versão curta estão presentes na ferramenta *cp-tools*, e todos eles são usados com a semântica sugerida pelos guias de boa prática. Estes 5 sinalizadores são *(-h, --help)*, *(-o, --output)*, *(-a, --all)*, *(-v, --version)* e *(-f, --force)*.

Dessa forma, este tópico do *checklist* é considerado atendido.

F.4.5 Toda entrada via *prompt* deve ter um argumento ou sinalizador associado

Não foi evidenciada rotina onde são utilizadas entradas via *prompt* de comando, dessa forma, não é possível afirmar que a ferramenta não atende a este tópico do *checklist*. Sendo assim, foi considerado que este tópico é atendido.

F.4.6 Confirme antes de fazer qualquer ação perigosa

A ferramenta *cp-tools* utiliza esta boa prática durante a execução do comando `cp-tools polygon pull`. Esse comando realiza a integração com a API do site Codeforces (CODEFORCES, 2021) e realiza o *download* de um determinado problema. Desse modo, durante essa sincronia de um problema é possível que arquivos locais sejam sobrescritos e ocorra perda de dados. Assim, por padrão, a ferramenta *cp-tools* não sobrescreve arquivos, exceto se o usuário confirmar explicitamente com o sinalizador *--force* ou *-f*. Diante disso, esse tópico do *checklist* é considerado atendido.

F.4.7 Não leia credenciais diretamente dos argumentos

A ferramenta *cp-tools* não utiliza esta boa prática, pois permite que credenciais sejam passadas por argumentos. Na atual versão da ferramenta, é possível informar a chave de acesso e a chave secreta da API do site Codeforces por meio dos sinalizadores *--key* e *--secret*. Desse modo, esse tópico do *checklist* não é atendido.

F.5 Boas práticas na interatividade

F.5.1 Somente habilite o *prompt* se `stdin` estiver associado à um terminal interativo

Não foi evidenciado rotina onde são utilizadas entradas via *prompt* de comando, dessa forma não há situações onde a entrada via *prompt* seja habilitada ou desabilitada. Desse modo, é considerado que este tópico do *checklist* é atendido.

F.5.2 Desligue o feedback (echo) de digitação enquanto o usuário digita dados confidenciais

Não foi evidenciada rotina onde são utilizadas entradas via *prompt* de comando, dessa forma, não há situações onde são obtidas credenciais via *prompt* de comando. Desse modo, é considerado que este tópico do *checklist* é atendido.

F.5.3 Permita o usuário interromper a execução

Não foi evidenciada sobrescrita de sinais na ferramenta *cp-tools*. Todos as sequências de teclas apresentadas na Tabela 10 foram testadas e o comportamento esperado foi obtido. Dessa forma, este tópico do *checklist* é considerado atendido.

F.6 Boas práticas no uso de subcomandos

F.6.1 Seja consistente em todos os subcomandos

Foi evidenciado que a ferramenta *cp-tools* segue esta boa prática, uma vez que comandos semelhantes possuem uma interface semelhante. Um exemplo disso são os comandos `cp-tools genpdf` e `cp-tools gentex`, comandos de geração de arquivos no formato PDF e LaTex, respectivamente. Ambos os comandos utilizam o mesmo conjunto de sinalizadores, conforme mostram os Códigos 49 e 48.

```
$ cp-tools genpdf --help
Usage: cp-tools gentex [-h] [-o outfile] [-b label]
                       [-c doc_class] [-g lang] [-l]
                       [-t] [--no-author] [--no-contest]
Generate a PDF file from the problem description.
The options are:

    -b              Defines the problem label. The default value
    --label         is 'A'.

    -c              Document class that will be used.
    --class         The default value is 'cp_modern'.

    -g              Problem language. The default value
    --lang          is 'pt_BR'.

    -h              Displays this help.
    --help

    -l              Lists all available document classes.
    --list

    -o              Output file. If omitted, the output
    --output        will be the file 'problem.pdf'.

    -t              Generates the PDF for the problem's
    --tutorial      tutorial.

    --no_author     Omits problem's author.

    --no_contest    problems contest.
```

Código 48: Interface do comando `cp-tools genpdf`

```
$ cp-tools gentex --help
Usage: cp-tools gentex [-h] [-o outfile] [-c doc_class]
                       [-l list] [-g lang] [-b label]
                       [--no_author] [--no_contest]
Generate a LaTeX file from the problem description.
The options are:

    -h              Displays this help.
    --help

    -o              Output file. If omitted, the output
    --output        will be written on stdout.

    -b              Defines the problem label. The default
    --label          value is 'A'.

    -c              Document class that will be used.
    --class         The default value is 'cp_modern'.

    -g              Problem language. The default value
    --lang          is 'pt_BR'.

    -l              Lists all available document classes.
    --list

    -t              Generates LaTeX file for problem's tutorial.
    --tutorial

    --no_author     Omits problem's author.

    --no_contest    Omits problems contest.
```

Código 49: Interface do comando `cp-tools gentex`

F.6.2 Use nomes consistentes para vários níveis de subcomando

Alem disso, foi observado que a ferramenta utiliza consistentemente o padrão `<verbo-ação> <nome-objeto>`, onde primeiramente é definida a ação e depois o objeto. Alguns exemplos de comandos que utilizam esse padrão são evidenciados no Código 50. Dessa forma, esse tópico do *checklist* é considerado atendido.

```
$ cp-tools judge solutions/ac.cpp

$ cp-tools check --validator
```

Código 50: Comandos do cp-tools que utilizam o padrão *<verbo-ação> <nome-objeto>*

F.6.3 Seja responsível

Para analisar se os comandos do CLI *cp-tools* é responsível foi utilizada a ferramenta `gnomon`. O `gnomon` é um utilitário desenvolvido pela empresa *PayPal*[2], cuja a principal funcionalidade é anotar o tempo decorrido entre as impressões de *logs* de um determinado software em execução. Conforme pode ser visto no Código 51, o utilitário `gnomon` foi capaz de informar quanto tempo cada *logs* de execução ficou sendo mostrado ao usuário.

```
$ cp-tools init | gnomon --high=0.1
   0.0065s   | Initializing directory '.' ...
   0.0035s   | Ok!
   0.0004s   |
             |
     Total   | 0.0110s

$ cp-tools clean | gnomon --high=0.1
   0.0067s   | Cleaning autogenerated files...
   0.0032s   | Ok!
   0.0004s   |
             |
     Total   | 0.0110s

$ cp-tools genpdf | gnomon --high=0.1
   3.3391s   | Ok! File 'problem.pdf' generated.
   0.0008s   |
             |
     Total   | 3.3403s

$ cp-tools genboca | gnomon --high=0.1
  13.8610s   | Ok! File 'problem.pdf' generated.
   0.0007s   |
             |
     Total   | 13.8620s
```

Código 51: Medição da responsividade dos comandos da ferramenta *cp-tools*

Foram realizadas medições em todos os subcomandos do *cp-tools* e foi evidenciado que a recomendação de no máximo 100 milissegundos entre emissões de *logs* não é seguida, dessa forma este tópico do *checklist* não é atendido.

[2] A PayPal é uma empresa de tecnologia financeira que opera um sistema de pagamentos online na maioria dos países que suportam transferências de dinheiro online e serve como uma alternativa eletrônica aos métodos tradicionais de papel, como cheques e ordens de pagamento.

APÊNDICE G – Conjunto de Caracteres Portátil

Este apêndice apresenta os caracteres alfanuméricos do conjunto de caracteres portáteis de sistemas POSIX.

name	glyph	C string	Unicode	Unicode name
zero	0	0	U+0030	DIGIT ZERO
one	1	1	U+0031	DIGIT ONE
two	2	2	U+0032	DIGIT TWO
three	3	3	U+0033	DIGIT THREE
four	4	4	U+0034	DIGIT FOUR
five	5	5	U+0035	DIGIT FIVE
six	6	6	U+0036	DIGIT SIX
seven	7	7	U+0037	DIGIT SEVEN
eight	8	8	U+0038	DIGIT EIGHT
nine	9	9	U+0039	DIGIT NINE
A	A	A	U+0041	LATIN CAPITAL LETTER A
B	B	B	U+0042	LATIN CAPITAL LETTER B
C	C	C	U+0043	LATIN CAPITAL LETTER C
D	D	D	U+0044	LATIN CAPITAL LETTER D
E	E	E	U+0045	LATIN CAPITAL LETTER E
F	F	F	U+0046	LATIN CAPITAL LETTER F
G	G	G	U+0047	LATIN CAPITAL LETTER G
H	H	H	U+0048	LATIN CAPITAL LETTER H
I	I	I	U+0049	LATIN CAPITAL LETTER I
J	J	J	U+004A	LATIN CAPITAL LETTER J
K	K	K	U+004B	LATIN CAPITAL LETTER K
L	L	L	U+004C	LATIN CAPITAL LETTER L
M	M	M	U+004D	LATIN CAPITAL LETTER M
N	N	N	U+004E	LATIN CAPITAL LETTER N
O	O	O	U+004F	LATIN CAPITAL LETTER O
P	P	P	U+0050	LATIN CAPITAL LETTER P
Q	Q	Q	U+0051	LATIN CAPITAL LETTER Q
R	R	R	U+0052	LATIN CAPITAL LETTER R
S	S	S	U+0053	LATIN CAPITAL LETTER S
T	T	T	U+0054	LATIN CAPITAL LETTER T

U	U	U	U+0055	LATIN CAPITAL LETTER U
V	V	V	U+0056	LATIN CAPITAL LETTER V
W	W	W	U+0057	LATIN CAPITAL LETTER W
X	X	X	U+0058	LATIN CAPITAL LETTER X
Y	Y	Y	U+0059	LATIN CAPITAL LETTER Y
Z	Z	Z	U+005A	LATIN CAPITAL LETTER Z
a	a	a	U+0061	LATIN SMALL LETTER A
b	b	b	U+0062	LATIN SMALL LETTER B
c	c	c	U+0063	LATIN SMALL LETTER C
d	d	d	U+0064	LATIN SMALL LETTER D
e	e	e	U+0065	LATIN SMALL LETTER E
f	f	f	U+0066	LATIN SMALL LETTER F
g	g	g	U+0067	LATIN SMALL LETTER G
h	h	h	U+0068	LATIN SMALL LETTER H
i	i	i	U+0069	LATIN SMALL LETTER I
j	j	j	U+006A	LATIN SMALL LETTER J
k	k	k	U+006B	LATIN SMALL LETTER K
l	l	l	U+006C	LATIN SMALL LETTER L
m	m	m	U+006D	LATIN SMALL LETTER M
n	n	n	U+006E	LATIN SMALL LETTER N
o	o	o	U+006F	LATIN SMALL LETTER O
p	p	p	U+0070	LATIN SMALL LETTER P
q	q	q	U+0071	LATIN SMALL LETTER Q
r	r	r	U+0072	LATIN SMALL LETTER R
s	s	s	U+0073	LATIN SMALL LETTER S
t	t	t	U+0074	LATIN SMALL LETTER T
u	u	u	U+0075	LATIN SMALL LETTER U
v	v	v	U+0076	LATIN SMALL LETTER V
w	w	w	U+0077	LATIN SMALL LETTER W
x	x	x	U+0078	LATIN SMALL LETTER X
y	y	y	U+0079	LATIN SMALL LETTER Y
z	z	z	U+007A	LATIN SMALL LETTER Z

APÊNDICE H – Situação que exemplifica o comportamento do padrão *singleton* utilizado na classe `PluginManager`

O Código 52 apresenta uma parte da classe `PluginManager`. Esta é a classe responsável pelo gerenciamento de plugins na ferramenta *cp-tools*. Esta classe implementa o padrão de projeto *singleton*, seguindo a implementação sugerida por Meyers (2005).

```
#include <iostream>
#include <vector>
#include <thread>

class PluginManager {
public:
    static PluginManager& get_instance() {
        static PluginManager instance;
        return instance;
    }
private:
    void load_plugins() { std::cout << "Loading plugins..." << std::endl; }
    void release_plugins() { std::cout << "Releasing plugins..." << std::endl; }

    PluginManager(const PluginManager&) = delete;
    PluginManager& operator=(const PluginManager&) = delete;

    PluginManager() {
        std::cout << "PluginManager()" << std::endl;
        load_plugins();
    }

    ~PluginManager() {
        std::cout << "~PluginManager()" << std::endl;
        release_plugins();
    }
};

int main() {

    std::vector<std::thread> threads(10);

    for(size_t i=0; i<threads.size(); i++) {
        threads[i] = std::thread([]() {
            auto& manager = PluginManager::get_instance();
        });
    }

    for(auto& t : threads) { t.join(); }
}

$ g++ singleton.cpp -pthread -o singleton
$ ./singleton
PluginManager()
Loading plugins...
~PluginManager()
Releasing plugins...
```

Código 52: Situação que exemplifica o comportamento do padrão *singleton* na classe `PluginManager`