UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Sharpener: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido.

Pedro Morello Abbud

São Carlos – SP

Sharpener: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido.

Pedro Morello Abbud

Orientador: **Prof. Dr. Dilvan de Abreu Moreira**

Coorientador: **Prof. Dr. Orlando de A. Figueiredo**

Monografia final de conclusão de curso apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como requisito parcial para obtenção do título de Bacharel em Engenharia da Computação.
Área de Concentração: Sistemas Computacionais

USP – São Carlos
Outubro de 2019

RESUMO

ABBUD, P. M.. ***Sharpener*: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido.**. 2019. 18 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Resumo

Palavras-chave: Palavras chave 1, palavras chave 2.

ABSTRACT

ABBUD, P. M.. ***Sharpener*: Ferramenta de correção automática de programas para o ensino de programação com workflow expandido.**. 2019. 18 f. Monografia (Graduação) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Resumo em inglês

Key-words: Keyword 1, Keyword2.

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Motivação e Contextualização

Nas últimas décadas, o interesse por programação cresceu de forma extraordinária e cursos introdutórios mostram-se cada vez mais populares. No entanto, cursos de programação ainda são considerados difíceis e, com frequência, possuem taxas de desistência elevada. Iniciantes sofrem de uma vasta gama de déficits e dificuldades, desde a compreensão de constructos básicos de uma linguagem de programação à elaboração de planos para resolver um dado problema (ROBINS; ROUNTREE; ROUNTREE, 2003).

Um grande número de estudos concluem, de forma semelhante, que alunos, mesmo tendo aprendido a sintaxe e semântica de uma linguagem de progamação, tendem a falhar em compor os blocos fundamentais estudados em programas válidos. Parte deste problema é abordado por Davies (1993), que faz uma distinção de duas partes essencias no aprendizado em programação: conhecimento e estratégia. Denota-se conhecimento em programação como a parte declarativa do conhecimento, *e.g.* saber que existe um constructo chamado *for* na linguagem e seu propósito. Já estratégia em programação seria empregar de forma adequada o conhecimento em face de um problema, como o uso pertinente de um *for* em um programa.
Davies (1993) defende que grande parte das abordagens em ensino tendem a investir excessivamente na parcela de conhecimento em programação, muitas vezes deixando de lado o aspecto de estratégia.

Além desta falta de enfâse no ensino de estratégia em programação, Winslow (1996) afirma que, ainda que o aluno saiba resolver manualmente o problema, existe uma grande dificuldade na adaptação desta solução para seu equivalente em um programa de computador. Este fato corrobora com a teoria de Brooks (1983) em que se afirma que programação é o processo de construir mapeamentos entre o domínio problema, possivelmente passando por muitos domínios intermediários, para chegar finalmente no domínio da programação. Por esta teoria, o entendimento de um programa nada mais é que a reconstrução de ao menos uma parte destas relações entre domínios e que este processo é orientado a criação, confirmação e refinamento de hipóteses.

A complexidade da tarefa não só se concentra no ensino de conhecimento e estratégia em programação nem na elaboração de habilidades que relacionem o domínio do problema ao domínio da programação, mas na flexibilidade que o iniciante deve alcançar na resolução

de problemas. Desta forma, o processo de escrita de um programa deve ser oportunística e incremental, em que o plano adotado e problema sempre estão em constante reavaliação, e que isto pode resultar em desvios e reformulações do plano original (VISSER, 1990).
Para que seja possível instrumentalizar esta escrita oportunística e incremental, o aluno deve adquirir outras habilidades como *debugging* de programas e familiaridade com um ambiente de programação.

Em vista ao imenso desafio que é o ensino de programação a iniciantes, diversas ferramentas foram desenvolvidas para dar apoio pedagócio ao professor nesta tarefa. Uma razoável parcela destas ferramentas são sistemas de juízes *online*, que consistem em sistemas capazes de receber códigos desenvolvidos pelos usuários como resposta a um determinado problema, mas com a capacidade de dar um *feedback* imediato ao usuário, informando se o código está correto ou não, através de análise sintática e testes automatizados.

Já existe uma grande variedade em juízes *online* disponíveis na literatura científica, tais como: *The Huxley* [Paes *et al.* (2013)], *CodeBench* [Galvão, Fernandes e Gadelha (2016)], *UVa Online Judge* [Revilla, Manzoor e Liu (2008)], *feeper* [Alves e Jaques (2014)], *URI Online Judge* [Bez, Tonin e Rodegheri (2014)], *BOCA* [Campos e Ferreira (2004)], *RunCodes*, *Sphere Online Judge*, *HackerRank*, *CodeChef*, *InterviewBit*, *Kattis*, *LeetCode*, *CodinGame*, *CodeSignal*, *CodeWars*, *Exercism*, entre tantos outros.

Apesar da grande variedade e maturidade destas plataformas existentes, poucas têm como foco ser uma ferramenta de auxílio didático ao professor em sala de aula e praticamente nenhuma é de código aberto. Assim sendo, existe uma lacuna que pode ser preenchida por um sistema que tenha estes dois focos e dê liberdade ao professor a implementar técnicas de ensino que o convenha.

1.2 Objetivos

Pretende-se neste trabalho desenvolver uma prova de conceito de um sistema para apoio ao ensino de programação em salas de aula, que seja de código aberto e que forneça ferramentas necessárias para o professor conduzir e monitorar exercícios práticos dados em sala de aula.

1.3 Organização

No Capítulo 2 pretende-se fazer uma revisão bibliográfica das técnicas empregadas na implementação da prova de conceito do sistema *Sharpener*. A seguir no Capítulo 3 descreve-se a especificação funcional do sistema tais como seus detalhes de implementação. Finalmente no Capítulo 4 apresentam-se os resultados e possíveis trabalhos futuros.

Capítulo 2

MÉTODOS, TÉCNICAS E TECNOLOGIAS UTILIZADAS

Este capítulo tem como propósito fornecer o embasamento teórico necessário para o entendimento da construção do sistema proposta. Aqui é contido uma descrição detalhada das técnicas, métodos e tecnologias utilizadas, assim preparando o leitor para o conteúdo dos próximos capítulos.

2.1 Design de APIs

Separação de conceitos, traduzido do termo *Separation of Concerns*, é uma temática chave na arquitetura cliente-servidor da internet. Parte do porquê a internet funciona tão bem foi a preocupação, desde sua concepção, de uma interface uniforme que, desde que respeitada, daria liberdade aos desenvolvedores de implementar, em qualquer linguagem ou tecnologia, um de seus componentes.

Apesar do sucesso na separação de conceitos entre as responsabilidades que o navegador e o servidor empregam, a mesma preocupação não foi replicada entre interfaces e funcionalidades que o servidor expõe. Com frequência existe um alto acoplamento entre a interface, *Frontend*, e o servidor, *Backend*, o que resulta em uma interface não uniforme para interação dos recursos que deveriam estar expostos. Define-se aqui um recurso como qualquer conceito na internet que pode ser referenciado por um identificador único e manipulado por uma interface uniforme (MASSE, 2011).

A solução, que leva ao desacoplamento, é a interação com estas funcionalidades través de uma *API*, *Application Programming Interface*, exposta pelo *backend*, que provê formas padronizadas de acesso. Aqui não somente ganhamos portabilidade, já que possibilitamos que não só um tipo de cliente, navegadores, saibam como acessar nossos recursos, mas ganhamos espaço para criar uma camada de abstração do serviço *Web*, modelando-o em recursos. Estes recursos não serão projetados como uma cópia da organização de dados ou funcionalidades presentes no *backend* mas em representações que sejam de fácil consumo e de entendimento ubíquo pelo lado do cliente.

O desafio de criar bons serviços na internet pode ser facilitado se empregarmos estilos arquiteturais já existentes. Aqui definimos estilos arquiteturais como um conjunto coordenado

de restrições arquiteturais.
Um destes estilos arquiteturas que tem ganhado cada vez mais tração se chama *REST*, *Representational State Transfer*, ou em português, *Transferência Representacional de Estado*. Cunhado por Fielding e Taylor (2000), o termo evoca como uma sistema na *Web* deveria se comportar, uma máquina de estados virtuais em que o usuário progride através da seleção de identificadores únicos, que identificam recursos, e verbos *HTTP* que operam sobre estes recursos.

As restrições na arquitetura impostas pelo estilo *REST* são agrupadas em seis categorias:

Cliente-servidor: A separação dos papéis do cliente e servidor deve ser clara para que estes possam ser projetados e implementados de forma independente. A interação entre eles só acontece na forma de requisições, que são iniciadas pelo cliente. Servidores devem mandar respostas apenas como reações de requisições dos clientes.

Interface uniforme: interfaces *REST* possuem quatro restrições: identificação de recursos, manipulação de recursos através de representações, mensagens auto-descritivas e hipermídia como motor de estado da aplicação, *HATEOAS*.
A primeira dita que cada recurso precisa ser enderessável por um identificador único, *URI*, *Unique Resource Identifier*.
Este recurso pode ser representado de diversas formas, seja um *HTML*, que é mais adequado para um navegador, ou *JSON* que é mais apropriado para consumo de outro programa. Percebe-se aqui que a representação é apenas uma forma de interagir com o recurso, não o próprio recurso. Isto é o que dita a segunda regra.
No consumo de um *API*, o cliente especifica um recurso e seu estado desejado, enquanto o servidor deve responder com o recurso e seu estado real. Esta troca de mensagens deve ser feita utilizando *headers* e códigos de estado *HTTP* que descrevam o estado do recurso e metadados correspondentes, de forma que as mensagens sejam auto-descritivas.
A última restrição, *HATEOAS*, ajuda a aumentar a visibilidade e descoberta de recursos relacionados na *API*, que ajudam o cliente a navegar dinamicamente nos recursos expostos. Assim, quando referenciam-se outros recursos, também são fornecidos seus identificadores únicos e verbos aceitos para aquela rota.

Sistema em camadas: Restringe-se o comportamento dos componentes em camadas, em que cada camada só pode interagir com subjacentes. Entre as chamadas do cliente que requisita uma representação de um estado de um recurso, e o servidor que processa a requisição, podem haver vários servidores entre eles. Estes servidores podem prover um camada de segurança, *cache*, balanceamento de carga ou outras funcionalidades. Estas camadas não devem afetar a requisição ou resposta e nem o cliente nem servidor precisam estar cientes se elas existem ou não.

Protocolo sem estado: O servidor não deve lembrar absolutamente nada do usuário que utiliza sua *API*. Isto implica que toda requisição individual precisa conter toda informação

necessária para que o servidor processe e retorne uma resposta. Esta restrição visa aumentar a escalablidade, visibilidade e confiabilidade do sistema. Escalabilidade pois permite que haja *caching* de respostas e que o servidor possa desalocar recursos físicos entre requisições, visibilidade pois sistemas que monitoram requisições não precisam olhar além de uma requisição, pois elas contém todo o dado necessário para entender a natureza desta, e a confibilidade pois a recuperação de falhas parciais se torna mais simples (KENDALL *et al.*, 1994).

Cache: O servidor deve declarar quais dados podem ser guardados em *caches*. *Caches* ajudam a reduzir a latência percebida pelo cliente, aumentam a disponibilidade e confiabilidade de serviços e mitigam a carga de trabalho do servidor. Em resumo, *caches* reduzem todos os custos associados a um serviço na *internet*.

Código sob demanda: A internet faz bastante uso de código sob demanda, que possibilita que o servidor transfira executáveis, tais como *scripts* e *plugins*, para a execução do lado do cliente. Apesar de proposto por Fielding e Taylor (2000), esta restrição tende a estabelecer um acoplamento de tecnologias entre servidor e cliente. Por este motivo "código sob demanda" é a única restrição imposta pelo estilo arquitetural *REST* que é considerada opcional.

O maior desafio no projeto de uma *API* é abstrair componentes do sistema em recursos. O modelamento destes recursos estabelece os aspectos chaves da sua *API* e é semelhante ao processo de modelar um banco de dados relacional ou mesmo o modelamento clássico de um sistema orientado a objetos.

No processo de modelagem de recursos, rotineiramente começa-se pensando em arquétipos de recursos. Estes arquétipos nos ajudam a comunicar de forma consistente as estruturas que são frequentemente encontradas em *Designs* de *REST APIs*. Idealmente, cada recurso pertencerá a apenas um dos seguintes arquétipos: *document*, *collection*, *controller* e *store*.

Document é o arquétipo base de todos os outros, todos os outros arquétipos são especializações destes. Sua representação tipicamente incluí campos com dados ou *hiperlinks* para outros recursos. *Collection* é um diretório de outros recursos em que clientes podem recuperar ou, possivelmente, adicionar novos items a coleção. Já *Stores* são recursos que permitem que usuários guardem novos recursos, de forma que o próprio cliente escolha o identificador daquele recurso. Para recursos que se parecem mais como procedimentos que não se encaixam nos métodos padrões, *CRUD* (*create, retrieve, update, delete*), o arquétipo *Controller* é adequado. Todos estes arquétipos contém recomendações de como o identificador único deve ser escolhido, para que seja claro a quem consome a API de qual arquétipo se trata aquele recurso.

2.2 Integração e entrega contínua

REFERÊNCIAS

ALVES, F. P.; JAQUES, P. Um ambiente virtual com feedback personalizado para apoio a disciplinas de programação. In: **Anais dos Workshops do Congresso Brasileiro de Informática na Educação**. [S.l.: s.n.], 2014. v. 3, n. 1, p. 51. Citado na página 10.

BEZ, J. L.; TONIN, N. A.; RODEGHERI, P. R. Uri online judge academic: A tool for algorithms and programming classes. In: IEEE. **2014 9th International Conference on Computer Science & Education**. [S.l.], 2014. p. 149–152. Citado na página 10.

BROOKS, R. Towards a theory of the comprehension of computer programs. **International journal of man-machine studies**, Elsevier, v. 18, n. 6, p. 543–554, 1983. Citado na página 9.

CAMPOS, C. P. de; FERREIRA, C. E. Boca: um sistema de apoio a competiç oes de programaç ao. 2004. Citado na página 10.

DAVIES, S. P. Models and theories of programming strategy. **International Journal of Man-Machine Studies**, Elsevier, v. 39, n. 2, p. 237–267, 1993. Citado na página 9.

FIELDING, R. T.; TAYLOR, R. N. **Architectural styles and the design of network-based software architectures**. [S.l.]: University of California, Irvine Doctoral dissertation, 2000. v. 7. Citado 2 vezes nas páginas 12 e 13.

GALVÃO, L.; FERNANDES, D.; GADELHA, B. Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. In: **Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)**. [S.l.: s.n.], 2016. v. 27, n. 1, p. 140. Citado na página 10.

KENDALL, S. C.; WALDO, J.; WOLLRATH, A.; WYANT, G. A note on distributed computing. Sun Microsystems, Inc., 1994. Citado na página 13.

MASSE, M. **REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces**. [S.l.]: "O'Reilly Media, Inc.", 2011. Citado na página 11.

PAES, R. de B.; MALAQUIAS, R.; GUIMARÃES, M.; ALMEIDA, H. Ferramenta para a avaliação de aprendizado de alunos em programação de computadores. In: **Anais dos Workshops do Congresso Brasileiro de Informática na Educação**. [S.l.: s.n.], 2013. v. 2, n. 1. Citado na página 10.

REVILLA, M. A.; MANZOOR, S.; LIU, R. Competitive learning in informatics: The uva online judge experience. **Olympiads in Informatics**, Institute of Mathematics and Informatics, v. 2, n. 10, p. 131–148, 2008. Citado na página 10.

ROBINS, A.; ROUNTREE, J.; ROUNTREE, N. Learning and teaching programming: A review and discussion. **Computer science education**, Taylor & Francis, v. 13, n. 2, p. 137–172, 2003. Citado na página 9.

VISSER, W. More or less following a plan during design: opportunistic deviations in specification. **International journal of man-machine studies**, Elsevier, v. 33, n. 3, p. 247–278, 1990. Citado na página 10.

WINSLOW, L. E. Programming pedagogy—a psychological overview. **ACM Sigcse Bulletin**, ACM, v. 28, n. 3, p. 17–22, 1996. Citado na página 9.

ANEXO A

LINKS RELACIONADOS

<https://www.thehuxley.com>: Página da plataforma *The Huxley*, que utiliza um sistema de juiz *online* para auxiliar professores em aulas de programação.

<http://codebench.icomp.ufam.edu.br>: Página da plataforma *CodeBench*, que utiliza um sistema de juiz *online*, de forma gamificada, para auxiliar professores em aulas de programação.

<https://uva.onlinejudge.org>: Página da plataforma *UVa Online Judge*, um sistema de juiz *online* desenvolvido pela *University of Valladolid*.

<http://feeper.unisinos.br>: Página da plataforma *UVa Online Judge*, um sistema de juiz *online* desenvolvido pela *Universidade do Vale do Rio dos Sinos*.

<https://www.urionlinejudge.com.br>: Página da plataforma *URI Online Judge*, um sistema de juiz *online* desenvolvido pela *Universidade Regional Integrada*.

<https://www.ime.usp.br/~cassio/boca>: Página da plataforma *BOCA*, um sistema de juiz *online* desenvolvido pelo *Instituto de Matemática e Estatística* para apoio em competições.

<https://we.run.codes>: Página da plataforma *RunCodes*, um sistema de juiz *online* desenvolvido pelo *Instituto de Matemática e Estatística*, para auxiliar professores em aulas de programação.

<https://www.spoj.com>: Página da plataforma *Sphere Online Judge*, um sistema de juiz *online* desenvolvido pela *Sphere Research Labs*.

<https://www.hackerrank.com>: Página da plataforma *HackerRank*, que utiliza um sistema de juiz *online* para auxiliar na contratação de funcionários para empresas.

<https://www.codechef.com>: Página da plataforma *CodeChef*, que utiliza um sistema de juiz *online*, desenvolvida pela empresa *Directi*, para competições em programação.

<https://www.interviewbit.com>: Página da plataforma *HackerRank*, que utiliza um sistema de juiz *online* para auxílio na preparação de entrevistas para empresas.

<https://open.kattis.com>: Página da plataforma *Kattis*, um sistema de juiz *online* desenvolvido para promover competições.

<https://leetcode.com>: Página da plataforma *LeetCode*, que utiliza um sistema de juiz *online* para auxílio na preparação de entrevistas para empresas.

<https://www.codingame.com>: Página da plataforma *CodinGame*, que utiliza um sistema de juiz *online*, de forma completamente gamificada, para ensino e auxiliar empresas na contratação de funcionários.

<https://codesignal.com>: Página da plataforma *CodeSignal*, que utiliza um sistema de juiz *online* para auxiliar empresas na contratação de funcionários.

<https://www.codewars.com>: Página da plataforma *CodeWars*, que utiliza um sistema de juiz *online* para auxiliar professores em aulas de programaçãoe e empresas na contratação de funcionários.

<https://exercism.io>: Página da plataforma *Exercism*, que utiliza um sistema de juiz *online*, para ensino. As submissões contam com feedback de instrutores voluntários.