UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Alberto Francisco Kummer Neto

EXPLORAÇÃO DE ANÁLISES AUTOMATIZADAS DE REPOSITÓRIOS DE CÓDIGOS PARA FEEDBACK FREQUENTE A ALUNOS DE PROGRAMAÇÃO

Santa Maria, RS
2017

Alberto Francisco Kummer Neto

EXPLORAÇÃO DE ANÁLISES AUTOMATIZADAS DE REPOSITÓRIOS DE CÓDIGOS PARA FEEDBACK FREQUENTE A ALUNOS DE PROGRAMAÇÃO

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**. Defesa realizada por videoconferência.

ORIENTADORA: Prof.ª Andrea Schwertner Charão

Santa Maria, RS
2017

Ficha catalográfica elaborada através do Programa de Geração Automática
da Biblioteca Central da UFSM, com os dados fornecidos pelo(a) autor(a).

Neto, Alberto Francisco Kummer
Exploração de análises automatizadas de repositórios de códigos para feedback frequente a alunos de programação / Alberto Francisco Kummer Neto.- 2017.
76 p.; 30 cm

Orientadora: Andrea Schwertner Charão
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2017

1. Automatização de processos 2. Análise estática de código 3. Visualização integrada de resultados I. Charão, Andrea Schwertner II. Título.

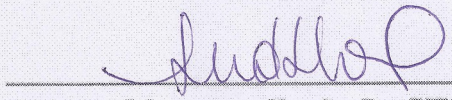
End. Eletr.: alberto@inf.ufsm.br

Alberto Francisco Kummer Neto

EXPLORAÇÃO DE ANÁLISES AUTOMATIZADAS DE REPOSITÓRIOS DE CÓDIGOS PARA FEEDBACK FREQUENTE A ALUNOS DE PROGRAMAÇÃO

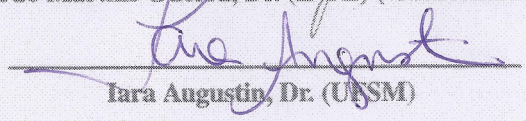
Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

Aprovado em 19 de maio de 2017:



Andrea Schwertner Charão, Dr. (UFSM)
(Presidenta/Orientadora)

Eduardo Martins Guerra, Dr. (INPE) (videoconferência)



Iara Augustin, Dr. (UFSM)

Santa Maria, RS
2017

DEDICATÓRIA

A todos que devotam-se ao avanço da fronteira do conhecimento humano.

AGRADECIMENTOS

A todos que acompanharam meu caminho até aqui, especialmente para meus pais Fátima da Cunha Kummer e Albino Carlos Kummer. Este trabalho é fruto de sua dedicação ao longo da minha jornada e mostra que todos os desafios vencidos não foram em vão.

À Professora Andrea Schwertner Charão, por aceitar-me sob sua orientação e pela paciência ao longo de todo o mestrado, pelos conselhos valiosos e pela instrução deste trabalho.

Aos colegas de pós-graduação que me acompanharam durante o desenvolvimento desse trabalho, pelo companheirismo nos momentos de descontração e apoio frente as adversidades.

À Professora Roseclea Duarte Medina e Josmar Nuernberg, pela constante presteza e disponibilidade que permitiram a realização das diversas atividades do meu mestrado.

À direção do Centro de Tecnologia, por toda a infraestrutura oferecida a comunidade discente e por ter me abrigado gentilmente durante a minha pós-graduação.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelos subsídios que permitiram minha total dedicação ao desenvolvimento deste e outros trabalhos relacionados à Ciência da Computação.

Convictions are more dangerous foes of truth than lies.

(Friedrich Wilhelm Nietzsche)

RESUMO

EXPLORAÇÃO DE ANÁLISES AUTOMATIZADAS DE REPOSITÓRIOS DE CÓDIGOS PARA FEEDBACK FREQUENTE A ALUNOS DE PROGRAMAÇÃO

AUTOR: Alberto Francisco Kummer Neto
ORIENTADORA: Andrea Schwertner Charão

Programação é uma habilidade difícil de ensinar e aprender, e as dificuldades de aprendizado são recorrentes entre alunos de Computação. Tal dificuldade decorre de sua natureza abstrata, fato que é corroborado pela extensa literatura sobre o assunto, e está diretamente relacionado aos índices de reprovação e evasão dos cursos da área. As primeiras discussões sobre o assunto datam da década de 80, e diversas trabalhos ressaltam a importância do tópico dado o seu efeito abrangente em todo o período de formação de um aluno. É possível identificar uma linha de discussões sobre metodologias para o ensino-aprendizagem de programação que seguem uma abordagens reativa, que sugerem soluções para problemas específicos identificados em turmas de programação. Tal abordagem é conhecida como padrão pedagógico, e tem como objetivo a caracterização de um problema pontual e sua respectiva solução, e cobre aspectos como contexto, forças, fraquezas, recursos e consequências. Na linha de discussão de padrões pedagógicos, há trabalhos afirmando que acompanhar o progresso dos alunos, individualmente, é um dos maiores desafios enfrentados pelos professores de programação, visto que é uma atividade trabalhosa e que demanda muito tempo do educador, e que por vezes mostra-se inviável frente a turmas grandes. Em linhas gerais, o progresso dos alunos costuma ser verificado por meio de atividades práticas de programação, que consistem na elaboração de soluções computacionais para uma série de problemas sugeridos pelo educador. Posteriormente, a análise minuciosa dos códigos dos alunos é capaz de indicar deficiências de aprendizado. A acompanhamento dos alunos por meio de atividades práticas também possibilita o emprego de ferramentas de análise automatizada de código, que potencialmente resulta em uma redução da carga de trabalho do educador, que poderia ser melhor gasto na avaliação de novas estratégias e metodologias de atividades didáticas. Este trabalho propõe uma solução para análise automatizada de repositórios de código de alunos de programação empregando-se ferramentas de análise estática de código. O método proposto sugere que as atividades práticas de programação sejam desenvolvidas dentro de um repositório *git* fazendo uso de serviços de hospedagem *online* como *Github* e *Bitbucket*, e prevê a análise simultânea de vários repositórios de código de uma turma de programação, agregando os resultados em uma interface de visualização integrada. Por tratar-se da automatização de um processo, o método possibilita um *feedback* rápido para instrutor e alunos da disciplina, promovendo de um parecer antecipado sobre as soluções das atividades práticas durante o seu desenvolvimento, habilitando a correção das implementações antes do prazo de entrega final das atividades. Adicionalmente, a abordagem estimula a frequente interação com sistemas de controle de versão, exercitando as habilidades dos alunos com esse tipo de ferramenta. Os experimentos conduzidos mostraram que a abordagem é válida, fornecendo indícios ao educador sobre a formação de seus alunos. Os educadores que cederam acesso aos repositórios de código de suas disciplinas relataram o interesse dos alunos que participaram do experimento, que foi verificado pela solicitação de acesso as detecções apontadas pela ferramenta.

Palavras-chave: Automatização de processos. Análise estática de código. Visualização integrada de resultados.

ABSTRACT

PROVIDING CONTINUOUS FEEDBACK TO PROGRAMMING STUDENTS THROUGH AUTOMATED ANALYSIS OF SOURCE CODE REPOSITORIES

AUTHOR: Alberto Francisco Kummer Neto
ADVISOR: Andrea Schwertner Charão

Programming is a difficult subject, both to teach and learn. Thus, learning difficulties are common between CS students, commonly by the abstract nature of this subject. This concern is reinforced by its lenghty literature, and several studies show its close relation with failure and evasion rates in related undergraduate courses. The first papers of this subject date from 80's, and several authors emphasize its importance due to its great affect to students formation process. Within the literature, it is possible to identify that "reactive" approaches gained more attention. This kind of discussions addresses very specific issues in programming of teaching-learning process, offering regular-shaped solutions (known as pedagogical patterns) to some problems. A pedadogical pattern suggests the capture and report of good teaching and learning practices, and emphasizes aspects such as context, forces, weakenesses, resources e consequences. Some pedadogical patterns discuss methodologies about how to keep track of student advances. The tracking activity could be overwhelming to teachers, as its complexity and time requirements grows with the classroom size. In general, teachers keep track of students progress through practice activities, which require that students to write some code to solve a problem specified by the teacher. Thus, a thorough analysis of students source code could indicate the learning defficiencies of some programmimg subjects. Practice activities also allow the use of tools to perform automatic code analysis, which could free some teacher time to more significant activities that might help in classes. This work introduces a tool to automate the analysis of source code repositories of programming students employing static code analysis tools. The proposed method suggest that all didactic activities be developed inside git repositories, and the publishing of solutions might be done using hosting servers like *Github* and *Bitbucket*, allowing the simultaneous analysis of several source code repositories of students of a programming class. It also aggregates the result of such analysis into a single report that can be viewed through a integrated user interface. As it is about the automation of a process, the tool allows the optimization of instructor time to evaluate students progress during the development of practical activities. This optimization reflects on a quick feedback to students as well, allowing a anticipated feedback to student about their work-in-progress solutions. Such feedback enables the improvement of their implementations before activities due date. The proposed approach has a additional gain of reinforce the usage of version control systems, a generally overlooked topic of CS courses. The conducted experiments showed that such approach is valid, as it gives evidences about students training. The teachers who participaded of study cases reported a strenghten engagement of students in classroom activities, as verified by the frequent requests of access to to detection reports compiled with the method.

Keywords: Process automation. Static code analysis. Integrated view of results.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SVN	Acrônimo para a ferramenta Subversion
Git	Ferramenta de versionamento distribuído git
IDE	Ambiente Integrado de Desenvolvimento, do inglês *Integrated Development Environment*
API	Interface de Programação de Aplicação, do inglês *Application Programming Interface*
HTTP	Protocolo de Transferência de Hipertexto, do inglês *Hypertext Transfer Protocol*
CSV	Arquivo de Texto Separado por Vírgula, do inglês *Comma Separated Values*
AST	Árvore de Sintaxe Abstrata, do inglês *Abstract Syntax Tree*

SUMÁRIO

1 INTRODUÇÃO

Dificuldades de aprendizado são comuns em cursos relacionados à informática e decorrem da natureza abstrata de disciplinas de programação (LAHTINEN; ALA-MUTKA; JÄRVINEN, 2005). Por esta razão, é frequente que os estudantes da área apresentem um baixo aproveitamento dos estudos, um fato que está relacionado a taxa de evasão dos cursos de computação (MARTINS; LOPES; RAABE, 2012; NETO, 2013)).

É usual que o desenvolvimento das habilidades de um aluno sejam verificadas através de atividades práticas de programação, que se iniciam com aplicações simples (implementação de uma calculadora, por exemplo) rumo a aplicações mais sofisticadas (implementação de um editor de texto, por exemplo). O esforço que o instrutor desempenha para acompanhar o progresso de cada aluno aumenta à medida que o grau de complexidade das tarefas cresce.

A estratégia de avaliação por meio de atividades se baseia em cronogramas sucintos. Em geral, o instrutor apresenta aos alunos uma descrição do que deve ser feito e um horizonte de tempo para desenvolvimento, teste e entrega das soluções. Posteriormente, o instrutor faz a correção das soluções dos alunos, para as quais são atribuídas notas ou conceitos conforme os critérios de avaliação utilizados. Frequentemente, a etapa de correção dos trabalhos é manual. Trabalhos recentes da literatura sugerem o uso do sistema de versionamento git como uma ferramenta de apoio, permitindo um rápido acompanhamento das atividades práticas de programação durante seu período de desenvolvimento, reduzindo a complexidade da etapa final de avaliação das atividades (KELLEHER, 2014; HAARANEN; LEHTINEN, 2015).

A maioria dos cursos de computação apresenta uma sequencia recomendada para integralização do currículo, de forma a maximizar o aproveitamento das disciplinas e garantir que cada aluno desenvolva progressivamente as habilidades esperadas. Nesse contexto, é comum a ocorrência de atividades em grupo em disciplinas mais avançadas, mas sua prática em disciplinas iniciais é desencorajada por dificultar o acompanhamento do desenvolvimento individual de cada aluno.

De maneira a mitigar a sobrecarga sobre instrutores, alguns cursos optam pela adoção da figura do monitor de disciplina, que age sob a supervisão do instrutor. Eventualmente, o monitor é responsável por uma primeira avaliação das soluções desenvolvidas pelos alunos, e

suas observações são posteriormente utilizadas na avaliação do instrutor.

Alguns instrutores buscam a automatização de etapas da análise das soluções dos alunos por meio de soluções próprias. Em outros casos, ferramentas automatizadas são empregadas com um objetivo específico, como ferramentas para detecção de plágio (PRECHELT; MALPOHL; PHILIPPSEN, 2002; SCHLEIMER; WILKERSON; AIKEN, 2003). Por fim, boa parte dos instrutores adotam um procedimento manual para correção das soluções dos alunos, uma prática de natureza repetitiva e sujeita a erros.

A disponibilidade de ferramentas que exerçam parte das incumbências do professor pode beneficiar a todos envolvidos no processo de ensino aprendizagem, reduzindo a sobrecarga sobre instrutores e monitores e permitindo que seus esforços sejam direcionados no acompanhamento do processo cognitivo de alunos que apresentam um desempenho aquém do esperado.

Diversas ferramenta automatizadas para análise de código estão disponíveis (MARJAMÄKI, 2013; WALDRON et al., 2016; SONARSOURCE, 2017), mas seus empregos se limitam a análise de somente um projeto de software por vez.

1.1 Objetivos

A proposta desta dissertação é uma abordagem de acompanhamento educacional baseada na análise automatizada de repositórios de código de alunos de programação. A automatização viabiliza a verificação e agregação de informações sobre diversos repositórios de código, empregando-se as ferramentas atualmente disponíveis para avaliação da qualidade de código fonte. Tal ferramenta seria capaz de auxiliar o instrutor a promover o *feedback* frequente a alunos de programação. Esse objetivo principal pode ser alcançado por meio da verificação dos seguintes objetivos específicos:

- Dar suporte a análise simultânea de múltiplos repositórios de código;
- Definir e implementar uma interface de programação para promover acesso homogêneo a ferramentas de análise de código distintas;
- Implementar e validar um mecanismo para comunicação com ambientes remotos de hospedagem de repositórios de código comumente utilizados;

- Desenvolver formas de visualizar o resultado das análises em um ambiente *online* e oferecer métricas de código com alguns níveis de granularidade.

1.2 Estrutura do trabalho

O restante desta dissertação está organizado como segue. O Capítulo 2 fornece a fundamentação utilizada na constituição deste trabalho, apresentando-se uma discussão da literatura sobre análise automatizada de código (Seção 2.1) e metodologias para ensino-aprendizagem de programação (Seção 2.2). O Capítulo 3 apresenta uma revisão da literatura dos trabalhos relacionados ao tema em estudo, com destaque para referências que utilizam ferramentas de versionamento como plataforma de ensino (Seção 3.1), bem como os trabalhos mais relevantes sobre análise estática de códigos para detecção de falhas de *design* e implementação de software (Seção 3.2). O Capítulo 4 apresenta os materiais e métodos utilizados pela metodologia desenvolvida. Dentro do Capítulo 5, as seções 5.1, 5.2 e 5.3 trazem a discussão sobre os casos de estudo de aplicação da metodologia proposta em três disciplinas distintas do Curso de Ciência da Computação da Universidade Federal de Santa Maria, e a Seção 5.4 trás as considerações finais sobre os experimentos com a metodologia. O Capítulo 6 traz o fechamento do trabalho, e as seções 6.1 e 6.2 apresentam as produções científicas relacionadas ao tema em estudo e sugestões de trabalhos futuros, respectivamente.

2 FUNDAMENTAÇÃO

Este capítulo introduz as fundamentações que embasam o desenvolvimento deste trabalho, com ênfase na bibliografia referente aos tópicos de análise automatizada de código e metodologias de ensino-aprendizagem de programação.

2.1 Análise automatizada de código

Na história da computação, Johnson (1978) apresentou um dos trabalhos seminais sobre análise automática de código fonte com o objetivo identificar defeitos em códigos fonte de programas de computador de maneira automatizada. Esse trabalho deu origem a ferramenta *lint*, que foi distribuída como parte integrante das ferramentas de sistema do sistema operacional UNIX, e tinha como objetivo analisar programas escritos com a linguagem de programação C. Por tratar-se de uma linguagem de programação nova, introduzida em 1972 por Dennis Ritchie, a forma com que os compiladores C suportavam as construções da linguagem variava substancialmente de um sistema computacional para outro, o que levava a introdução de defeitos súbitos em aplicações aparentemente funcionais. Nos casos mais brandos, as aplicações simplesmente deixavam de compilar devido ao uso de construções e dialetos que fugiam da especificação oficial da linguagem.

Lint foi criado para fazer a identificação automática do uso de extensões não oficiais da linguagem C e teve um papel importante na consolidação da linguagem, enfatizando uma das suas características mais importantes: ser portátil. A aceitação da ferramenta foi tanta que a análise automática de código fonte foi incorporada dentro de diversos compiladores e interpretadores da época, uma tendência que permanece até os dias atuais (DARWIN, 1991; STALLMAN et al., 2003). Em geral, as ferramentas disponibilizadas para desenvolvimento com linguagens de terceira geração e subsequentes oferecem suporte a algum tipo de análise automatizada de código (AHO; SETHI; ULLMAN, 1986).

Eventualmente, a análise feita pelas ferramentas extrapolou o escopo do estilo de programação e validação das construções da linguagem utilizadas, direcionando-se a uma análise mais profunda conhecida como análise estática de código (PIERCE, 2002). Este termo surgiu como

uma classificação genérica ao tipo de análise de código que busca a identificação automática de defeitos e falhas de concepção em código fonte de aplicações e se baseia na verificação minuciosa de código fonte. Ao tornarem-se mais robustas, as tarefas de análise de código passaram a exigir um esforço computacional cada vez maior, o que culminou no desenvolvimento modular de aplicações. Dessa forma, cada módulo poderia ser analisado separadamente, viabilizando o uso geral das técnicas de verificação de código em conjunto as ferramentas de desenvolvimento.

2.1.1 Ambientes integrados de desenvolvimento e a programação assistida por computador

Com o eventual aumento dos recursos computacionais, o escopo de emprego de ferramentas de análise automática de código foi ampliado, além de permitir o desenvolvimento de análises mais sofisticadas. Eventualmente, surgiram as primeiras plataformas unificadas de desenvolvimento que forneciam ao programador um acesso rápido às ferramentas utilizadas durante o ciclo de desenvolvimento de uma aplicação, e posteriormente consolidaram-se como ambientes integrado de desenvolvimento (IDEs) (DELINE; ROWAN, 2010).

As primeiras IDEs basicamente ofereciam um mecanismo para edição de código fonte e acesso uma série de atalhos a ferramentas comumente utilizadas no desenvolvimento, como a chamada ao compilador ou invocação de um depurador de código. Alguns ambientes ofereciam recursos mais poderosos que o simples acesso a comandos de compilação via atalhos, permitindo a visualização de erros de sintaxe diretamente no editor de código do ambiente integrado (WEXELBLAT, 2014). Assim, a principal motivação para o desenvolvimento dos IDEs foi expectativa de redução do tempo gasto pelo programador na frequente transição de uma ferramentas de trabalho para outra.

Outra atividade que demanda uma quantidade significativa do tempo de um programador é a leitura de código fonte, que tende a crescer proporcionalmente ao tamanho das aplicações (ERLIKH, 2000). Nesse contexto, é usual que o programador mantenha paralelamente várias linhas de raciocínios, pois o código de aplicações "não triviais" tende a ser uma composição de diversas unidades menores. Consequentemente, essa fragmentação torna a tarefa de leitura de código dispersiva (KO et al., 2006; PLUMLEE; WARE, 2006). Assim, um segundo obje-

tivo dos ambientes integrados de desenvolvimento é aumentar a produtividade do programador, fornecendo mecanismos para transição rápida entre trechos de código longínquos. Entre os recursos mais comuns estão o de dobramento de código, o de completamento automático de código sensível à sintaxe da linguagem e semântica das construções, e ainda o recurso *outlines* para visualização de trechos de código na forma de janelas *popups*. A Figura 2.1 trás um exemplo de IDE com suporte ao três recursos mencionados.

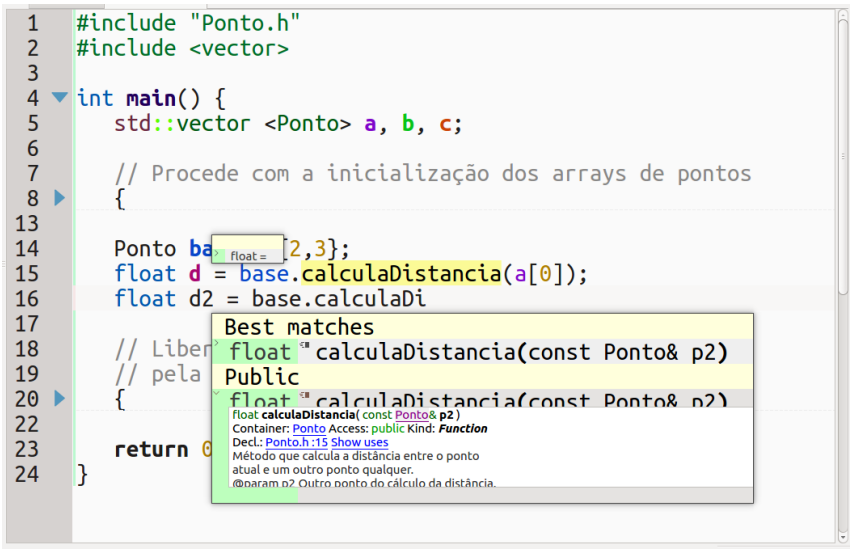


Figura 2.1 – Exemplo dos recursos oferecidos por um ambiente integrado de desenvolvimento para a linguagem de programação C++, como destaque aos recursos de dobramento de código e completamento de código sensível a semântica.
Fonte: Autoria própria.

De maneira geral, os recursos de assistência de código oferecidos pelos ambientes integrados de desenvolvimento estimulam a adoção uma postura de programação defensiva, fornecendo resultados da análise automatizada dos trechos de código que o programador está utilizando, em tempo real, e que pode ser acompanhada dentro da própria interface de edição de código fonte. Além da detecção automática de defeitos, algumas ambientes de desenvolvimento também oferecem ações automáticas para auxiliar a implementação de melhorias no código fonte, geralmente na forma de refatoração de código (ROBERTS; BRANT; JOHNSON, 1997; FOWLER; BECK; BRANT, 1999).

2.1.2 Detecção de *code smells* em programas de computador

Indicativos de defeitos mais profundos, como os relacionados à modelagem e ao design de um programa de computador, não podem ser verificados com grande precisão através de técnicas convencionais de análise estática de código. Esta constatação refletiu no estabelecimento de um novo tópico de estudo pela comunidade de engenharia de software, que foi amplamente difundido em conjunto às técnicas de refatoração de código, conhecido como *code smells* (FOWLER; BECK; BRANT, 1999; LANZA; MARINESCU, 2007). A refatoração de código surgiu como uma referência formal para as ações corretivas que fazem uso um tipo mais avançado de análise estática de código, e que está intrinsecamente relacionada à análise da estrutura lógica e semântica das partes que compõem uma aplicação.

Os primeiros estudos que buscam automatizar a identificação de possíveis defeitos estruturais e semânticos em código fonte datam de 2002. No trabalho de Emden e Moonen (2002), os autores apresentam uma metodologia para identificação de possíveis defeitos estruturais no desenvolvimento de uma aplicação através da análise estática de código, conforme a arquitetura apresentada pela Figura 2.2. Por tratar-se de uma forma de análise nova, a arquitetura desenvolvida apresenta componentes característicos de etapas de compilação de código (como "árvore de sintaxe abstrata"), que são agregados de maneira a estabelecer um modelo de representação de código fonte que viabilize a identificação de *code smells*. Os autores empregaram a arquitetura proposta na análise de uma aplicação escrita em Java, e apontaram uma série de defeitos relacionados ao uso incorreto ou deficiente dos conceitos de orientação a objetos em seu código fonte, ocasionando problemas de design que podem ser facilmente resolvidos usando-se os recursos de herança e polimorfismo oferecidos pela linguagem de programação Java.

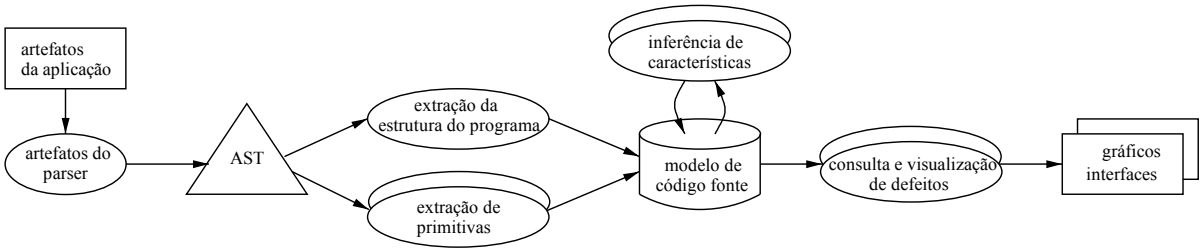


Figura 2.2 – Arquitetura proposta por Emden e Moonen (2002) para a construção de uma ferramenta de identificação de *code smells*.

Fonte: Traduzido do inglês de Emden e Moonen (2002).

Utilizando-se de uma arquitetura similar, Olbrich et al. (2009) apresentaram a análise de duas aplicações *open source* quanto a presença de indícios de defeitos de código *shotgun sugery* (classes fortemente acopladas) e *god class* (classe sobrecarregada de responsabilidades). Em linhas gerais, os autores apresentaram uma série de regras capazes de interagir com um modelo de representação de código fonte, algo similar ao mecanismo de inferência de características de Emden e Moonen (2002). Os autores relataram sucesso em acompanhar o amadurecimento do design das aplicações ao comparar o crescimento das aplicações de teste e o número de ocorrência dos defeitos em diversas versões de seu código fonte. Por fim, os autores concluem o trabalho ressaltando que a metodologia cumpriu seu objetivo, e que a granularidade da análise necessita de um ajuste fino pois apenas alguns componentes das aplicações analisadas que apresentavam, de fato, os defeitos observados pelo estudo.

2.2 Metodologias de ensino-aprendizagem de programação

Dificuldades de aprendizado são comuns em cursos de programação e decorrem principalmente de sua natureza abstrata (LAHTINEN; ALA-MUTKA; JÄRVINEN, 2005). Por esta razão, os processos de ensino-aprendizagem são frequentemente discutidos pela comunidade científica e educadores da área de Computação (PEARS et al., 2007). De fato, o aprendizado das disciplinas de programação é um grande desafio para alunos iniciantes, sendo recorrente a verificação de um baixo rendimento entre alunos dos períodos iniciais dos cursos de computação (LAHTINEN; ALA-MUTKA; JÄRVINEN, 2005; MARTINS; LOPES; RAABE, 2012; NETO, 2013)). Como um agravante, a presença de baixo desempenho nas disciplinas iniciais de programação tende a se repetir a médio e longo prazo, perdurando por todo o período de formação do aluno (RAABE, 2007).

Mesmo para os alunos que detém alguma experiência com programação, há o permanente desafio do aprimoramento das habilidades, paralelamente ao aprendizado de novos paradigmas de programação e exercício de técnicas de desenvolvimento ágil (JENKINS, 2002; TAN; TING; LING, 2009; MARTINS; LOPES; RAABE, 2012). Os autores de Sheard et al. (2009) frisam essas constatações, e apontam que o considerável volume da literatura relacionada às dificuldades de programação apenas reitera a afirmação que programação é tão difícil

de aprender quanto ensinar. Complementarmente, Beaubouef e Mason (2005) apontam que as dificuldades de aprendizado de programação refletem em uma elevada taxa de reprovação e evasão dos cursos relacionados a informática.

Diversos estudos vêm investigando os contextos em que as dificuldades na aprendizagem de programação frequentemente ocorrem (LAHTINEN; ALA-MUTKA; JÄRVINEN, 2005; DALE, 2006; NGUYEN et al., 2014). De toda a forma, é importante ressaltar que educadores e instituições de ensino devem exercer uma permanente reflexão sobre as metodologias e as práticas de ensino-aprendizagem adotadas frente à turmas de programação, e uma constante avaliação sobre como uma mudança de metodologia poderiam causar um impacto positivo em toda a experiência de formação do aluno (DENNING, 1989).

Uma das correntes de pesquisa sobre metodologias de ensino-aprendizagem de programação discute o reflexo das respostas para as perguntas mais comuns desse contexto, como por exemplo "*que linguagens adotar com iniciantes de programação?*". Nesse sentido, autores como Soloway e Spohrer (1988) e Winslow (1996) buscaram respostas para essa e outras perguntas frequentes e seus respectivos impactos nos processos cognitivos de aprendizagem de programação.

Em uma segunda corrente estão os estudos que adotam uma abordagem mais reativa, e propõem formas de mitigar dificuldades que surgem de maneira relativamente bem estabelecida, em uma abordagem centrada nos chamados "padrões pedagógicos" (WILSON; SHROCK, 2001; CASPERSEN; KOLLING, 2009). Embora sejam oriundos da comunidade interessada especificamente no ensino-aprendizagem de programação orientada a objetos (SHARP et al., 1996), os "padrões pedagógicos" eventualmente atingiram um escopo mais amplo e passaram a ser abertamente utilizado em discussões de metodologias de ensino-aprendizagem de programação e seus diversos problemas (LARSON; TREES; WEAVER, 2008; KÖPPE et al., 2015; CHARÃO et al., 2016).

A abordagem de padrão pedagógico consiste em capturar as boas práticas de ensino-aprendizagem exercidas por educadores, de uma forma compacta e organizada, e que suficientemente completa para que possa ser facilmente comunicada e reaplicada em contextos semelhantes. Desde o trabalho seminal de Sharp et al. (1996), diversos padrões pedagógicos

foram introduzidos, catalogados e publicados, para os mais variados contexto. Estritamente sobre o tema de ensino-aprendizagem de programação, há padrões diversificados que incentivam os alunos a construírem programas com erros (padrão *Mistake* de Bergin et al. (2012)), e ainda padrões sobre a programação em tempo real junto aos alunos como uma alternativa a aulas expositivas com *slides* (padrão *Show Programming* de Schmolitzky (2007)). Autores da área como (BERGIN et al., 2012) apontam que, embora alguns padrões pareçam triviais para educadores experientes, ele são uma referências útil para instrutores menos experientes ou para aqueles que busca diversificar sua didática de sala de aula.

Padrões pedagógicos tem conquistado espaço dentro da comunidade científica, e está presente em conferências tradicionais sobre ensino de computação, como SIGCSE (*ACM Technical Symposium on Computing Science Education*) e ITiCSE (*ACM Conference on Innovation and Technology in Computer Science Education*) (FINCHER; UTTING, 2002; SHARP; MANNS; ECKSTEIN, 2003).

3 TRABALHOS RELACIONADOS

Este capítulo trás um compilado dos trabalhos da literatura de ensino-aprendizagem de programação que possuem uma maior similaridade ao assunto abordado. São apresentadas as discussões que motivam o uso do sistema de versionamento git como plataforma de ensino. Também é feita um apanhado dos trabalhos relacionados a análise automatizada de código fonte de alunos de programação.

3.1 Sistema de versionamento *git* como ambiente de ensino

Sistemas de versionamento são sistemas utilizados para registro das modificações feitas em um projeto de software ao longo do tempo. Inicialmente projetados como um sistema de *backup* de código fonte, os sistemas de versionamento permitem que o usuário navegue dentro de um histórico de versões e faça uma rápida comparação entre versões distintas de códigos presentes no histórico da ferramenta. Adicionalmente, os sistemas de versionamento de primeira geração (ROCHKIND, 1975; TICHY, 1985) também armazenavam metadados com registros de autor, data, hora e comentário sobre as modificações feitas no repositório.

Seguindo os preceitos das ferramentas de primeira geração, sistemas modernos como *CVS* e *Subversion* classificam-se como sistemas centralizados: todo o repositório é mantido em uma máquina de acesso comum aos colaboradores de um projeto de software e as operações sobre o histórico (*checkouts* e *commits*, por exemplo) são feitas diretamente no servidor, como ilustrado na Figura 3.1 (PILATO; COLLINS-SUSSMAN; FITZPATRICK, 2008). Em contrapartida, sistemas como *BitKeeper* e *git* utilizam o modelo de sistema de versionamento distribuído, em que cada colaborador possui uma cópia completa em sua estação de trabalho, e um ou mais servidores remotos são utilizados para a integração de código, como ilustrado na Figura 3.2 (HENSON; GARZIK, 2002; CHACON; STRAUB, 2014).

Git é um dos sistemas de versionamento distribuído mais utilizados na atualidade (GOOGLE TRENDS, 2017). Git foi inicialmente projetado por Linus Torvalds com um substituto *open-source* a ferramenta proprietária *Bitkeeper*, e teve como objetivo adicional cobrir deficiências de seu antecessor que tornavam as tarefas de revisão e integração de código do sistema

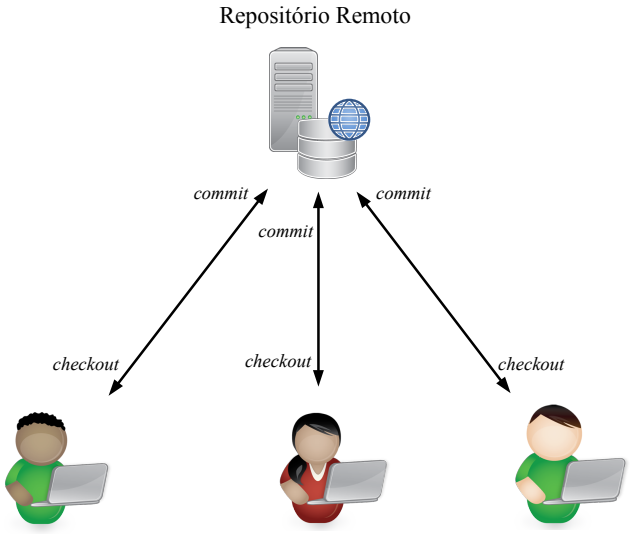


Figura 3.1 – Formas básicas de interação com sistemas de versionamento centralizado. Todos os colaboradores do projeto interagem diretamente com o servidor que hospeda o repositório, e todas as operações são executadas via rede (local ou remota). A ausência de conectividade impede a interação com a ferramenta de versionamento.

Fonte: Autoria própria.

operacional Linux difíceis de serem executadas (LINUX.COM, 2017). É fácil encontrar sites que ofereçam gratuitamente serviços de hospedagem online de repositórios git, como *Github* e *Bitbucket*. A configuração de um servidor git é simples e utiliza ferramentas que já vem instaladas na maioria das distribuições Linux atuais. Em conjunto a esses fatores, git permite que uma quantidade massiva de desenvolvedores colaborem com um único projeto de software, e apesar de ser considerado de difícil utilização, ganhou popularidade entre desenvolvedores do mundo todo.

Inspirados na metáfora de sistemas de versionamento, diversos autores propuseram a utilização de repositórios como ambiente de ensino de disciplinas de Cursos de Computação, como Subversion (CLIFTON; KACZMARCZYK; MROZEK, 2007) e mais recentemente, git (LAADAN; NIEH; VIENNOT, 2010; LAWRANCE; JUNG; WISEMAN, 2013; KELLEHER, 2014; HAARANEN; LEHTINEN, 2015; CHARÃO et al., 2016) como uma alternativa a plataforma de ensino convencionais como Blackboard e Moodle.

Observam-se diversas vantagens para o instrutor com o uso de sistemas de versionamento como plataforma de ensino. Lawrance, Jung e Wiseman (2013) apontam que o sistema de versionamento facilita a comunicação entre instrutores e alunos por centralizar em um único

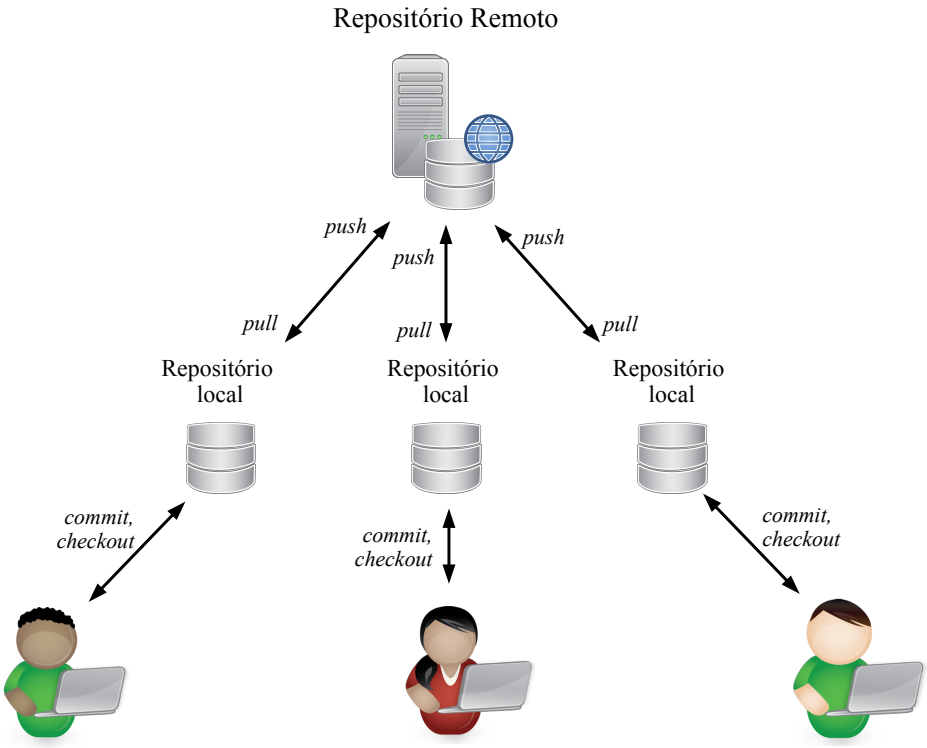


Figura 3.2 – Formas básicas de interação com sistemas de versionamento distribuídos. Cada colaborador detém uma cópia completa do repositório, e todas as operações são executadas localmente. Eventualmente, as modificações podem ser disponibilizadas para os demais colaboradores através de um servidor que também hospeda uma cópia do repositório.

Fonte: Autoria própria.

ambiente todas as informações sobre o andamento da disciplina (notas, tira-dúvidas sobre atividades em andamento, assistência em horário não-letivo). Além disso, os autores comentam algumas vantagens diretamente relacionadas à forma de funcionamento de sistemas de versionamento de código, como a cooperação facilitada no desenvolvimento de atividades de grupo, facilidade de comunicação entre diversos instrutores e monitores de disciplina, e ainda a identificação antecipada de deficiências de aprendizado com base nos hábitos dos alunos. (Como um benefício adicional,) Por fim, a capacidade de interação com sistemas de versionamento é vista como um diferencial por ser uma habilidade atípica entre alunos formados (HAARANEN; LEHTINEN, 2015). No entanto, os autores que tais metodologias incorrem sobrecarga para os instrutores, que além de conduzir as disciplinas, passam a desempenhar funções administrativas para configuração e manutenção de servidores que hospedam os repositórios de código, conforme apontado por Lawrance, Jung e Wiseman (2013) e Kelleher (2014).

3.2 Detecção automática de defeitos e falhas de design em código fonte

A detecção de defeitos e a busca por soluções para problemas de concepção costumam estar presentes durante todo o ciclo de desenvolvimento de uma aplicação. Boa parte das linguagens de programação e ambientes integrados de desenvolvimento oferecem vários tipos de ferramentas que auxiliam nas tarefas de teste e refatoração de código de forma semi automatizada. Como todo o desenvolvimento de uma aplicação se dá dentro desse ambiente integrado, o programador é capaz de verificar constantemente a qualidade da aplicação que está sendo desenvolvida através das dicas e recomendações feitas pela própria IDE, permitindo a correção prematura de defeitos menores presentes no código.

A capacidade de detecção prematura de defeitos pode ser ampliada com a adoção de algumas técnicas de desenvolvimento de código. Complementarmente à análise estática, existem técnicas de desenvolvimento de software que fazem uso de instrumentação de código e códigos de teste para promover a análise dinâmica da aplicação. Uma forma comum de implementação deste tipo de análise se dá por meio de testes de unidade, na qual o programador pode atestar, pontualmente, o correto funcionamento de cada elemento do sistema de software (MCCONNELL, 2009). Testes de unidade permitem a validação dos componentes de uma aplicação por meio de várias sequências de entrada e a verificação das respectivas saídas. Para uma aplicação escrita na linguagem de programação C, por exemplo, um teste unitário pode validar se uma função está operando corretamente por meio da comparação da saída encontrada e da saída esperada para diversas entradas cuja saída o programador conhece a priori. Um exemplo de teste de unidade pode ser visto no Algoritmo 1.

Ainda no contexto de testes de unidade encontram-se as metodologias de desenvolvimento orientada a testes (TDD, do inglês *Test Driven Development*), nas quais a escrita dos testes de unidade deve preceder a escrita da aplicação propriamente dita. Dessa forma, cada elemento que compõe a aplicação tem sua funcionalidade verificada individualmente. O desenvolvimento orientado a testes também mitiga a ocorrência de falhas de concepção, e promove a redução de acoplamento do sistema como um todo (BECK, 2003). Inerentemente, a demanda constante da escrita de código-fonte de testes implica em alguma sobrecarga de trabalho no desenvolvimento de aplicações.

Algoritmo 1 Exemplo de unidade de teste em C por meio da macro `assert`.

```
/**
 * Testa se o parâmetro é um valor par.
 * @return 0 em caso afirmativo ou 1 caso contrário.
 */
int is_odd(int value) {
   return value % 2 != 0;
}

/**
 * Testes unitários para a função @link{#is_odd}.
 */
void tests_is_odd() {
   assert(is_odd(1) == 1 && "is_odd(1) incorreto.");
   assert(is_odd(3) == 1 && "is_odd(3) incorreto.");
   assert(is_odd(2) == 0 && "is_odd(2) incorreto.");
   assert(is_odd(4) == 0 && "is_odd(4) incorreto.");
}
```

Algumas linguagens de programação e ambientes de desenvolvimento integrado empregam técnicas avançadas de análise de código, que permitem a detecção automatizada de certos defeitos que usualmente só são capturados por meio da análise dinâmica da aplicação. Entre as detecções mais usuais estão a de vazamento e congestão de memória (NETHERCOTE; SEWARD, 2007). Linguagens como Java ainda fornecem o mecanismo de anotação e documentação nativa de código, que auxiliam no processo de refatoração e design de novos componentes de uma aplicação (KRAMER, 1999).

3.2.1 Detecção automática de defeitos código fonte de alunos de programação

Em disciplinas inicias dos cursos de computação, é comum que professores sejam mais restritivos quanto ao uso de ferramentas para auxiliar no desenvolvimento das atividades. Em uma disciplina que use a linguagem C, por exemplo, é comum que seja solicitado ao aluno a interação direta com o compilador, e incentiva-se o uso de parâmetros de diagnóstico para geração de mensagens de aviso. Em disciplinas mais avançadas, pode ser sugerido aos alunos a utilização de um ambiente integrado de desenvolvimento, bem como algumas ferramentas mais avançadas para a análise das aplicações (ferramentas com instrumentação automática como *gprof* e *Valgrind*, por exemplo).

Em linhas gerais, a inspeção manual é a forma mais frequente de análise de código

para detecção de defeitos em software escritos em linguagens de programação profissionais, e a literatura sobre o metodologias automatizadas é escassa (LEE; KESTER; SCHULZRINNE, 2011). A maioria dos trabalhos relacionados à análise de código de alunos de programação concentram-se na discussão de linguagens e ambientes específicos para o ensino de programação por meio de pseudo linguagens como "portugol" (BRUSILOVSKY et al., 1997; SOUZA, 2016), ou por meio de programação visual (BROWN, 1987). Outras abordagens sugerem a visualização do funcionamento de algoritmos escritos em linguagens reais (MORENO et al., 2004; EGAN; MCDONALD, 2014; MYERS, 1986).

Outra prática comum empregada para a detecção de defeitos em aplicação desenvolvidas por alunos de programação faz uso da saída das aplicações para uma série de entradas cujo resultado é conhecido a priori. Assim, diversas soluções podem ser testadas sistematicamente frente a um conjunto de entradas de teste pré estabelecidos, seguindo a metodologia de testes de software *black box* (BEIZER, 1995; MYERS; SANDLER; BADGETT, 2011). Com a adição de algumas restrições do ambiente de execução, essa abordagem é frequentemente utilizada em competições e maratonas de programação (CAMPOS; FERREIRA, 2004).

4 MÉTODO E IMPLEMENTAÇÃO DA SOLUÇÃO PROPOSTA

Este capítulo apresenta o método proposto pelo autor desta dissertação para análise automatizada de repositórios de códigos de alunos de programação. É feita uma discussão sobre os aspectos funcionais de uma ferramenta resultante da implementação do método, com ênfase na especificação dos requisitos de operação. A seguir, são discutidos os modelos para entrada e saída de dados, e limitações decorrentes de aspectos técnicos que foram encontrados durante a implementação do método.

4.1 Visão geral do método proposto

O método proposto nesta dissertação sugere o uso da análise automatizada de repositórios de código para a promoção do *feedback* frequente a alunos de programação, e pode ser materializado na forma de uma ferramenta de software. Seguindo os objetivos listados na Seção 1.1, tal implementação deve prezar pelos aspectos de modularidade e extensibilidade, de forma a promover a extensibilidade da ferramenta para contextos diversificados, permitindo a rápida inclusão de suporte a novas linguagens de programação e novos analisadores de código, por exemplo.

Como pré requisito geral, o método proposto prevê a análise simultânea de diversos repositórios de código de alunos de programação. Como as ferramentas de análise automatizada de código disponíveis são desenvolvidas para uso pontual (somente um repositório de código por vez, ou um subconjunto dos código fonte da uma aplicação), algumas adaptações se fazem necessárias. Na atual implementação, segue-se uma abordagem iterativa sobre os repositórios de código, que são analisados individualmente em uma primeira etapa. Em uma etapa posterior, os resultados individuais são agregados, compondo-se um grande relatório unificado de detecções.

Como previamente observado, a solução sugere o uso das ferramentas de análise de código bem estabelecidas, como o *cppcheck* (MARJAMÄKI, 2013), em um escopo distinto daquele para qual foram projetadas. Dessa forma, para cada repositório de programação analisado, faz-se a execução de cada um dos analisadores suportados pela ferramenta, que tem a

sua saída capturada e convertida para um formato padrão de representação via JSON ou XML (CROCKFORD, 2006; BRAY et al., 1997). A Figura 4.1 demonstra em alto nível a forma com que a ferramenta atua.

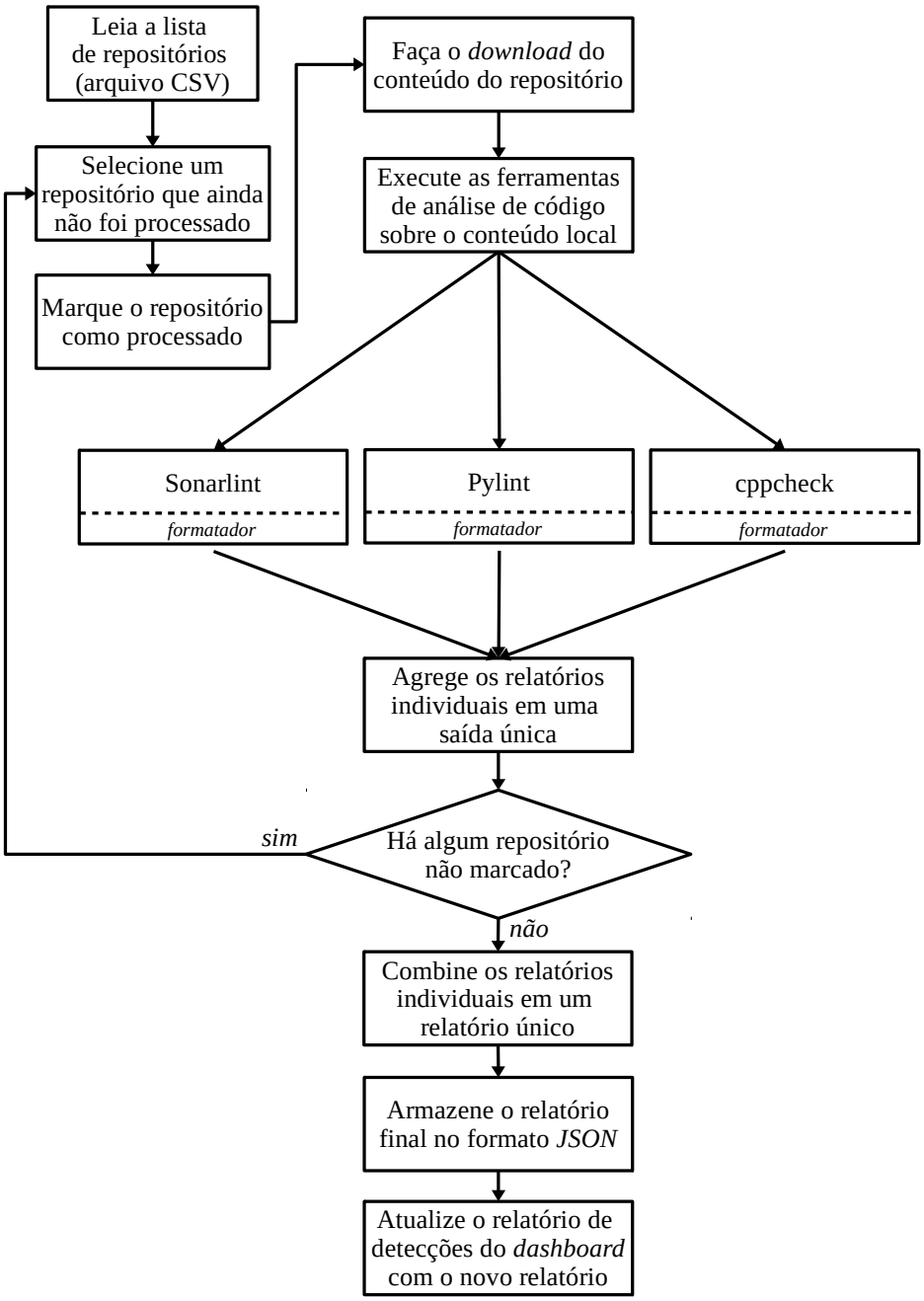


Figura 4.1 – Representação em fluxograma dos processos envolvidos no método proposto.

Fonte: Autoria própria.

4.1.1 Requisitos de implementação da solução proposta

É natural que a especificação de um *software* capaz de implementar o método proposto nesta dissertação obedeça os requisitos específicos elencados na Seção 1.1. A discussão sobre o cumprimento de tais objetivos é feita nas subseções a seguir.

4.1.1.1 Entradas e saídas

O primeiro objetivo específico trata do suporte simultâneo a múltiplos repositórios de código. Assim, a implementação do método utiliza como entrada um arquivo de texto separado por vírgula (CSV, do inglês *Comma Separated Values*), e cada linha possui obrigatoriamente quatro campos, conforme o exemplo da Tabela 4.1. O primeiro campo apresenta o nome do aluno responsável pelo repositório, seguido pelo campo que apresenta o nome do projeto de software desenvolvido dentro do repositório. O terceiro campo indica o endereço remoto para acesso ao repositório de código e o quarto campo contém qual versão do código fonte deve ser utilizada na análise (útil para casos em que a versão mais recente do código extrapola a data de entrega estabelecida pelo professor).

Usuário	Nome do projeto	*URL*	*commit*
dvmuccillo	designare	https://github.com/dvmuccillo/designare	e7d98
cassiandrei	logicinalogicway	https://github.com/cassiandrei/logicinalogicway	6d0d3
vinniekun	projsoft2	https://bitbucket.org/Vinniekun/projsoft2	dc898

Tabela 4.1 – Exemplo de entrada para a ferramenta desenvolvida.

O segundo objetivo específico sugere utilização de múltiplos analisadores de código, denotado em alto nível pela operação *formatador* na Figura 4.1. Essa operação é capaz de formatar a saída de uma analisador e convertê-lo para um formato padrão, e deve obrigatoriamente conter os campos listados na Tabela 4.2. O campo *file* lista qual arquivo apresenta um defeito, que é "explicado" pelo conteúdo do campo *issue_description*. Adicionalmente, o defeito tem um grau de severidade *severity* e uma chave de identificação *issue_key*, comum a todas as detecções do mesmo tipo. Por fim, o arquivo JSON de saída da operação *format* é uma lista de objetos contendo todas as detecções feitas no repositório de código.

Campo	Conteúdo
file	Arquivo que apresentou o defeito
severity	Grau de severidade do defeito
issue_description	Descrição pontual do defeito
issue_key	Identificador único do tipo de defeito

Tabela 4.2 – Campos esperados para cada detecção feita por um analisador.

Ainda sobre o segundo objetivo específico, há a necessidade de um mecanismo comum de acesso aos analisadores de código. Valendo-se da tipagem dinâmica da linguagem de programação que a ferramenta foi implementa (Python), estabeleceu-se uma interface de programação que é suportada por todos os analisadores suportados pela implementação do método. Tal interface deve, obrigatoriamente, conter o método *mainAnalyser*, e espera os seguintes parâmetros para execução: *project_id*, que especifica um apelido para o repositório a ser analisado, e *path*, que é o caminho em que os arquivos do repositório se encontram. Esse método, então, devolve ao chamador uma *string* JSON, que contém as detecções feitas pelo analisador. Atualmente, as linguagens de programação e analisadores cobertos pela implementação da ferramenta estão listados na Tabela 4.3.

Analisador	Linguagens suportadas	Referência
sonarlint	Java, Javascript, PHP, Python	SonarSource (2017)
cppcheck	C, C++	Marjamäki (2013)
pylint	Python	Thenault (2017)

Tabela 4.3 – Analisadores suportados pela atual implementação da metodologia.

4.1.1.2 Mecanismos para comunicação com repositórios remotos git

O terceiro objetivo específico sugere a adoção de uma interface padronizada para comunicação com os repositórios remotos. A atual implementação do método dá suporte a dois serviços de hospedagem de repositório *git* populares. *Github* provê hospedagem gratuita para o público geral, e oferece acesso aos dados de um repositório por meio de uma interface programação de *webservice*, e implementa a proteção do serviço por meio do mecanismo *OAuth* (HARDT, 2012).

Bitbucket é o segundo serviço de hospedagem de repositórios suportado e dispõe uma API de acesso semelhante. Embora *Bitbucket* também adote o mecanismo de autenticação *OAuth*, não são disponibilizados servidores de autorização para acesso ao serviço, o que dificulta o a comunicação com a plataforma por exigir do cliente do serviço a implementação de um servidor próprio para autorização de acesso ao *webservice*.

O acesso via *webservice* aos serviços de hospedagem foi implementado por meio da biblioteca de comunicação de hipertexto *urllib2*, que faz parte da biblioteca padrão da linguagem de programação Python (ROSSUM; DRAKE, 1995; ROSSUM et al., 2007). Adicionalmente, o acesso a serviços de hospedagem git é genericamente feito por meio da operação de clonagem nativa da ferramenta de versionamento.

4.1.1.3 Visualização de resultados por meio de dashboard

De posse da lista de repositórios e dos resultados de suas análises, o processamento, filtragem e exibição das detecções pode ser implementada. Nesse contexto, a adoção do formato de representação das detecções com JSON possibilitou o uso do relatório final como um banco de dados de detecções, e optou-se por exibir os resultados das análises em uma interface *web*, que nativamente suporta JSON e permite acesso aos dados a partir de qualquer dispositivo conectado à internet.

Usando-se o serviço de hospedagem de páginas estáticas *Github pages* (GITHUB, 2017) em conjunto ao banco de detecções em formato JSON, foi concebida uma interface de visualização de detecções no formato de *dashboard*, como demostrado pela Figura 4.2. Apesar do serviço não dispor de hospedagem dinâmica de conteúdo, a adoção de uma base de detecções com JSON permite que o processamento dos dados seja feita pelo navegador *web*, habilitando os uso de recursos usualmente restritos a páginas *web* dinâmicas.

A interface dispõe ao usuário três formas de visualização das detecções. A tela inicial do *dashboard* preocupa-se em exibir dados mais gerais acerca dos repositórios analisados, e aponta a contabilização geral dos 10 erros mais encontrados em lista e em um gráfico de colunas. Além disso, também são apontados metainformações sobre o ambiente de execução da

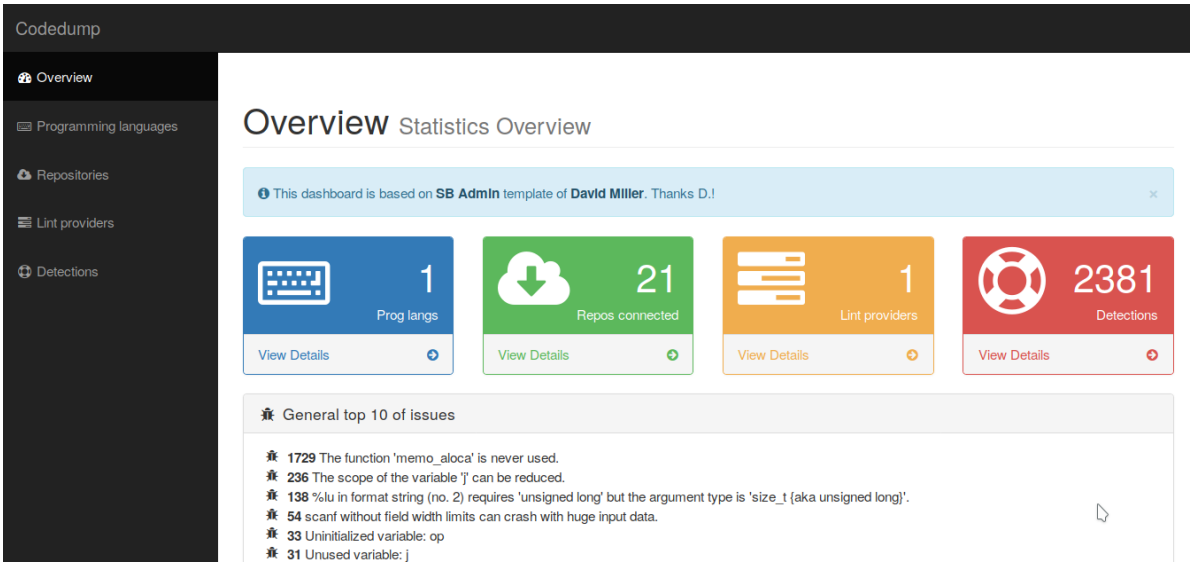


Figura 4.2 – Tela inicial do *dashboard* de visualização.

Fonte: Autoria própria.

ferramenta conforme botões presentes no topo da página, que exibem o número de linguagens de programação suportadas, o número de repositórios analisados, o número de analisadores utilizados, e total de ocorrências detectadas.

Além da exibição da tela inicial, foram implementadas outras duas formas para visualização das detecções, que estão acessíveis pelos botões “Repos connected” e “Detections”, ou pelo menu lateral em “Repositories” e “Detections”, respectivamente. A visualização por repositório é demonstrada na Figura 4.3, e foca na exibição do total de detecções por repositório. A forma de visualização por detecção, por sua vez, permite a visualização das detecções com granularidade fina, conforme mostrado na Figura 4.4.

A tela “Detections” foi desenvolvida com o objetivo de permitir um maior controle da visualização, e oferece opções para filtragem e ordenação dos tipos de defeitos a serem listados. Na atual implementação, é possível omitir detecções com base em sua severidade, ou ainda por linguagem de programação. Além disso, todas as listagem feitas nessa tela são interativas. Um clique sobre uma entrada da tabela “Detection” filtra a contagem de ocorrências mostradas na tabela “Details”, que é responsável por mostrar o número de ocorrências de tal defeito em cada repositório de código analisado. Um clique sobre uma de suas entradas revela quais arquivos do repositório apresentam tal defeito, e “Repository details” apresenta um *link* para acesso aos arquivos relacionados.

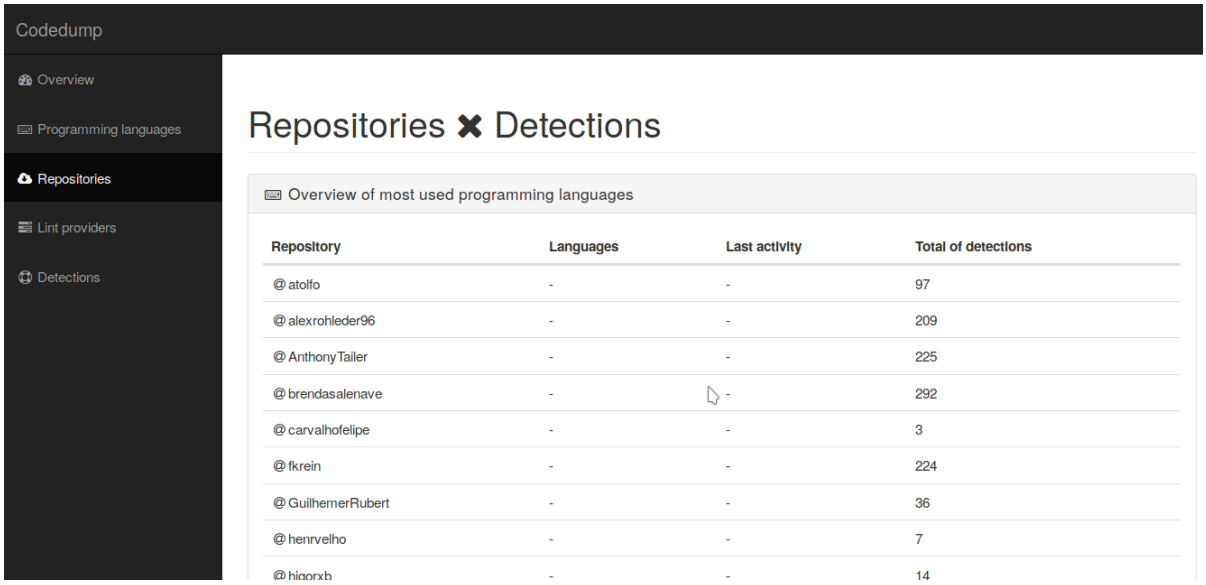


Figura 4.3 – Visão dos resultados de detecção por repositórios.

Fonte: Autoria própria.

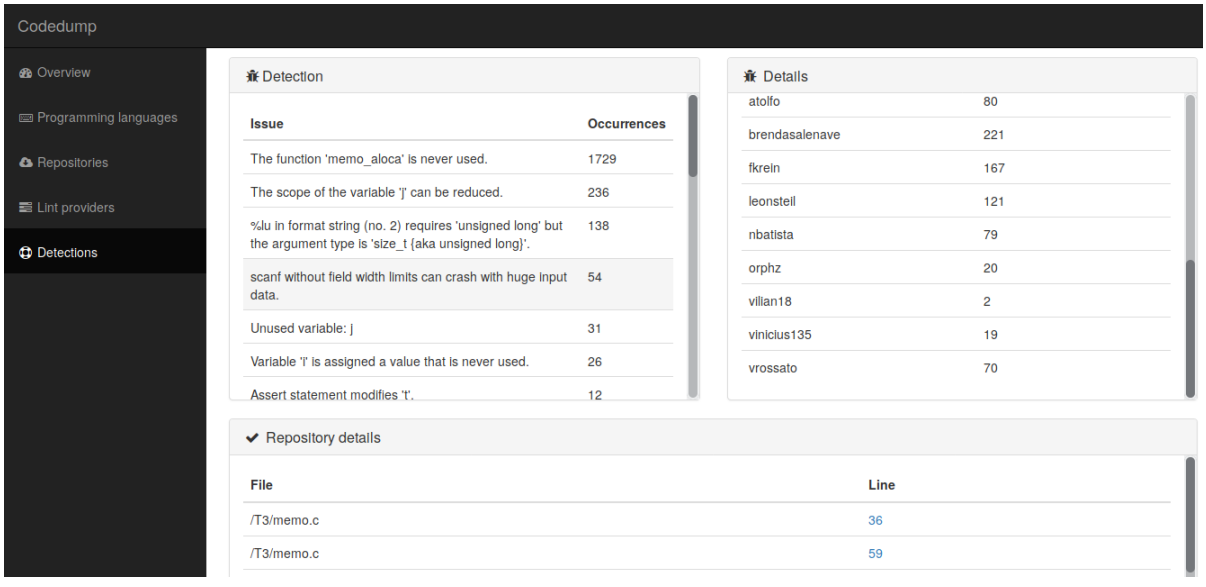


Figura 4.4 – Visão detalhadas dos resultados por tipo de defeito. Na imagem, estão destacadas as ocorrências para o defeito *the function 'memo_aloca' is never used*, e estão listadas na tabela "Repository details" as ocorrências desse defeito no repositório do aluno *AnthonyTailer*.

Fonte: Autoria própria.

5 DISCUSSÃO DO MÉTODO

Este capítulo concentra as discussões referentes a três aplicações do método apresentado no Capítulo 4. Os experimentos foram conduzidos em disciplinas distintas, que são ofertadas regularmente aos alunos dos Cursos de Ciência da Computação e Sistemas de Informação da Universidade Federal de Santa Maria.

O primeiro caso de estudo relata a primeira experiência com o método e teve como objetivo principal a verificação da viabilidade da abordagem em extrair informações sobre o código desenvolvido pelos alunos da disciplina. O segundo caso de estudo relata a experiência com o método em uma disciplina do final de curso, e teve como objetivo verificar os benefícios do seu emprego dentre os alunos experientes de um curso relacionado à computação. A terceira experiência com o método deu-se em uma turma de início de curso. Por tratar-se de uma turma grande, foi verificada a capacidade da ferramenta em permitir a rápida verificação de progresso dos alunos frente as atividades práticas propostas pelo instrutor da disciplina.

5.1 Caso de estudo I: Aplicação da metodologia em uma turma de Paradigmas de Programação

Uma versão preliminar da abordagem apresentada no Capítulo 4 foi aplicada a uma turma da disciplina de Paradigmas de Programação[1], que é regularmente ofertada aos alunos de terceiro período dos Cursos de Ciência da Computação e Sistemas de Informação da Universidade Federal de Santa Maria. É nesta disciplina que os alunos de ambos os cursos tem seu primeiro contato com paradigmas de programação distintos da programação imperativa com a Linguagem C, com destaque para os paradigmas funcional, orientado a objeto, concorrente e lógico.

Por tratar-se de uma turma grande com uma elevada variedade de atividades, o instrutor frequentemente precisa ajustar a granularidade de acompanhamento dos alunos para cumprir as atividades previstas no cronograma da disciplina. Desta forma, foram aplicadas aos alunos vá-

[1] Cronogramas e atividades da disciplinas disponíveis em <www-usr.inf.ufsm.br/~alberto/dissertacao-ufsm2017.php?key=elc117>.

rias atividades de fixação fora do período de aula, de forma a consolidar e exercitar os conceitos de diversos paradigmas de programação. Posteriormente, os alunos foram avaliados através de atividades práticas e uma prova dissertativa. Nesse contexto, buscou-se verificar se emprego de uma ferramenta automatizada para análise dos códigos fonte produzidos pela turma poderia facilitar o acompanhamento dos alunos por parte do instrutor, se a abordagem se mostra escalável ao tamanho da turma, e se os resultados da análise possibilitariam um *feedback* útil para todos os envolvidos.

Na turma em questão, o método foi aplicado no contexto do paradigma de programação orientado a objeto com Java, para as soluções desenvolvidas para resolver o problema exposto na Figura 5.1. Conforme a descrição a atividade, o aluno deveria reaproveitar código de uma atividade de fixação anterior, e prosseguir com a implementação da solução do problema apresentado conforme os recursos oferecidos pelo paradigma de orientação a objeto. Os requisitos da solução foram elaborados para encorajar a familiarização do aluno com as bibliotecas nativas da linguagem Java. A solução deveria ser entregue no período de uma semana, através de um repositório hospedado no *Github*. A metodologia do caso de estudo consistiu em coletar o código fonte dos repositórios dos alunos da disciplina, e experimentar o método de análise automática para avaliação das soluções provenientes da atividade sugerida pelo instrutor.

Objetivo do trabalho: Encontrar a figura geométrica de maior área dentre os itens de uma lista de objetos, usando-se técnicas de programação orientada a objetos com Java.

Descrição: Considere o ambiente de produção de uma padaria artesanal. Numa dada fornada, há 50 bolachas de formatos variados, espalhadas sobre uma assadeira retangular. Cada bolacha está numa posição específica, determinada por pontos situados na assadeira, sem sobreposição. A partir do cálculo da a área de cada bolacha, sua tarefa é localizar a maior bolacha de uma fornada.

Figura 5.1 – Descrição da atividade prática na qual a metodologia foi aplicada. A versão completa do texto está disponível no *site*[2] da disciplina.

Fonte: Autoria própria.

Duas observações devem ser feitas sobre o caso de estudo, visto que tratou-se da aplicação de uma versão experimental do método. Primeiramente, optou-se pela utilização de uma

[2] Veja em <www-usr.inf.ufsm.br/~alberto/dissertacao-ufsm2017.php?key=elc117>.

ferramenta de análise de código própria, capaz de oferecer um relatório de detecções reduzido para os defeitos esperadas para um trabalho inicial de disciplina. Em segundo, não houve a intenção de se construir uma interface de usuário para visualização do relatório de detecções. O relatório de detecções foi armazenado em um arquivo de texto puro, que foram posteriormente apresentados aos alunos por meio de uma apresentação de *slides* construída pelo instrutor da disciplina no formato de apresentação de Charão et al. (2016), em que vários trechos de código desenvolvido pelos próprios alunos da disciplina são apresentados e discutidos, com o objetivo de coibir as más práticas de programação verificadas.

A ferramenta de análise de código desenvolvida serviu para a detecção de erros mais trivialmente cometidos por alunos iniciantes na programação orientada a objetos, mas que apontam falhas na compreensão de conceitos fundamentais de tal paradigma. Tal abordagem teve como objetivo avaliar a aplicabilidade da metodologia proposta, e verificar sua potencial utilidade tanto para alunos quanto instrutores. Um sumário das detecções suportadas está presente na Tabela 5.1.

Defeito	Descrição
Ausência de '`static`'	Definição incorreta de constantes de classe. Cada objeto detêm uma cópia idêntica de algum atributo imutável
Ausência de '`final`'	Definição incorreta de constantes de classe. Objetos da classe compartilham uma constante sem proteção contra modificação
Visibilidade de atributos	Uso direto de atributos de objeto, com ou sem a presença de *getters* e *setters*
'`static`' em atributos de objeto	Compartilhamento incorreto de atributos entre várias instâncias de uma classe

Tabela 5.1 – Defeitos detectados pela implementação própria de analisador de código Java.

A métrica utilizada para avaliação dos resultados da análise baseou-se na contabilização de possíveis defeitos de código nas soluções de cada aluno. Embora a análise tenha apresentado pouca informação, foi possível identificar deficiência no uso dos conceios de programação orientada à objetos com Java, e portanto, a ferramenta foi empregada com sucesso. A discussão sobre as soluções dos alunos foi feita com base nos resultados na análise automatizada, que foram apresentados para a turma em uma data posterior a entrega da atividade prática, em

um formato similar à Figura 5.2. Na ocasião, verificou-se implicitamente o interesse dos alunos pela proposta da metodologia, segundo o relato do instrutor a respeito das discussões levantadas no decorrer discussão dos resultados da análise.

Aluno	Arquivo:Classe	1	2	3	4
Ana Luisa	InsereForma.java:InsereForma Ponto.java:Ponto		distCentroMax		alt larg tam
Filipe Sim	Ponto.java:Ponto				x y
Pablo Ca	Assadeira.java:Assadeira Bolacha.java:Bolacha Bolacha.java:Bolacha			altura largura area ponto	se id
Daniel Ma	Bolacha.java:Bolacha			tipo	
Caroline	Bolacha.java:Bolacha bolachaRetangulo.java:bolachaRetangulo Ponto.java:Ponto Bolacha.java:Bolacha			tam ponto tipo base altura coordX coordY tam ponto	
Adonai G	Array.java:Array			Bolachas	
João Vito	Pont.java:Ponto				x y
Vinícius T	Bolacha.java:Bolacha bolachaRetangulo.java:bolachaRetangulo Ponto.java:Ponto			tam ponto base altura coordX coordY	
Francisco	Bolacha.java:Bolacha Ponto.java:Ponto				tipo x y
Leonardo	chapa.java:Chapa	gerador			
Gabriel C	Ponto.java:Ponto	posX			posX
		2	1	11	7

N#	Descrição
1	falta 'final'
2	falta 'static'
3	visibilidade publica de atributo de objeto
4	static em atributo de objeto

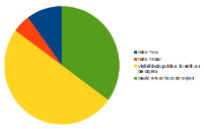


Figura 5.2 – Resultados apresentados aos alunos no primeiro caso de estudo, no formato de *slides*.

Fonte: Autoria própria.

Alguns aspectos da metodologia mostraram-se como ameças ao seu emprego. Primeiramente, a ausência de uma interface com o usuário ocasionou uma carga de trabalho extra para o instrutor da disciplina, que ficou responsável pela agregação e filtragem dos resultados obtidos com a ferramenta. Um outro ponto falho decorre do potencial não explorado da ferramenta, que é a promoção de um *feedback* rápido durante o desenvolvimento das atividades práticas. Tal *feedback* permitiria a identificação rápida de defeitos a tempo de correção antes do término do prazo de entrega das soluções, que na ocasião se mostrou inviável por demandar uma carga de trabalho extra sobre o instrutor para a geração do relatório de detecções.

5.2 Caso de estudo II: Aplicação da metodologia em uma turma Projeto de Software

A segunda ocasião em que a metodologia foi posta em prática foi em uma turma da disciplina de Projeto de Software II, obrigatória para alunos do sétimo período do Curso de Sistemas de Informação da Universidade Federal de Santa Maria. Tal disciplina exige que os alunos elaborem um projeto de software para resolver um problema de sua escolha. O planejamento deve incluir uma discussão da escolha do sistema de software adotado (linguagem, ferramentas, etc.), a análise de requisitos do software (funcionais e não-funcionais), bem como seus casos de uso, uma visão geral da arquitetura do software, definição de um padrão de codificação e documentação, e por fim, estratégias para teste do protótipo desenvolvido[3].

Trata-se de uma disciplina extensa, na qual os alunos podem se organizar em grupos colaborativos para desenvolvimento das atividades, e há encontros semanais com o instrutor da disciplina o acompanhamento do progresso das atividades. Esse caso do estudo teve como objetivos a verificação da potencial utilidade da metodologia frente a uma turma experiente e aplicações elaboradas, e introduziu-se uma interface de usuário para visualização dos relatórios das análises. A metodologia foi aplicada em dois momentos para a turma do segundo semestre do ano 2016, num total de 13 alunos e 8 projetos distintos, conforme o sumário apresentado na Tabela 5.2. A heterogeneidade dos projetos demandou a inclusão de ferramentas de análises de código de terceiros para cobrir a variedade de tecnologias utilizadas pelos alunos em suas soluções.

Alunos envolvidos	Nome do projeto	Tecnologias utilizadas
2	Designare	Python, Javascript
2	RP-Email	Javascript
2	ProgressCode	Python, Java, Javascript
2	Logic in a Logic Way	Java
2	MicroCosmos	Python
1	Try2Survive*	C#
1	Portaria*	C#
1	GradHora	PHP

Tabela 5.2 – Projetos de software desenvolvidos durante o curso da disciplina.

[3] Veja <http://www-usr.inf.ufsm.br/~alberto/dissertacao-ufsm2017.php?key=elc1069-ementa> e <http://www-usr.inf.ufsm.br/~alberto/dissertacao-ufsm2017.php?key=elc1073-ementa> para mais informações.

O experimento com a ferramenta foi feito após as primeiras entregas parciais dos projetos, conforme estipulado pelo instrutor no início das atividades da disciplina. Nesta aplicação da metodologia, foi feita a utilização da ferramenta de análise estática de código *sonarlint*, que é capaz de avaliar a qualidade e identificar defeitos em códigos escritos nas linguagens de programação Java, Javascript, Python e PHP (SONARSOURCE, 2017). A metodologia utilizada foi similar ao caso de estudo anterior, com a adição de que os resultados da análise foram apresentados no formato de página *web* no estilo *dashboard*, resolvendo o problema da ausência de uma interface de usuário amigável para acompanhamento dos resultados da análise.

Dada a natureza distinta dos projetos e as diversas tecnologias utilizadas, a gama de detecções apontadas pela ferramenta foi diversificada, e soluções distintas apresentaram problemas em comum. Os 3 projetos que usaram a linguagem Python apresentaram problemas com o uso de objetos, nos quais trechos de código faziam referência a membros de objeto inexistentes ou usaram membros de classe como de objetos (49 ocorrências), mistura dos estilos de declaração de classes entre várias versões da linguagem (48 detecções), ou ainda a definição de membros de objeto fora do método `__main__`. Embora estas últimas duas detecções não impeçam o funcionamento das aplicações, sua elevada ocorrência demonstra falhas de projeto e execução da solução de software, e uma deficiência mais profunda dos alunos acerca do funcionamento da linguagem de programação Python.

Os alunos envolvidos demonstraram interesse no relatório gerado pela ferramenta, requisitando ao instrutor da disciplina acesso ao seu conteúdo. A solução adotada para atender tal pedido foi a utilização do serviço de hospedagem de páginas estáticas *Github.io*, adaptando-se o formato do relatório de saída da ferramenta para JSON, para que pudesse ser carregado dinamicamente pela interface via Javascript (CROCKFORD, 2006). Em linhas gerais, o experimento demonstrou a utilidade da ferramenta ao promover o *feedback* antecipado sobre a qualidade das soluções desenvolvidas pelos alunos. Por tratar-se da automatização do processo manual de correção dos trabalhos, a metodologia possibilitou a rápida identificação de falhas no código fonte das aplicações, assentindo a correção de alguns defeitos e falhas de design antes da data final de entrega das soluções. As métricas utilizadas na contabilização das detecções indicaram os 10 problemas mais frequentes, a contagem de detecções por repositório, a contagem das ocor-

rências por defeito. Em especial, a filtragem de detecções por repositório despertou o interesse e motivos os alunos, e forneceu dados ao instrutor para argumentação sobre o andamento das soluções na discussão em sala de aula.

Conclui-se que novas ferramentas de análise de código podem ser facilmente empregadas junto a ferramenta proposta, de maneira a compatibilizar a metodologia com linguagens de programação distintas. A abordagem permaneceu válida frente a alunos experientes, e seu emprego em disciplinas cujo foco não é necessariamente o aprendizado de um novo tópico de programação se mostrou relevante. Embora o *feedback* possibilitado pela ferramenta tenha sido bem recebido, não houve tempo hábil para o emprego de um instrumento para verificação das percepções dos alunos e verificação da efetividade do método na promoção da qualidade das soluções de *software* desenvolvidas. Verificou-se também que a metodologia de análise automatizada é dependente da existência de ferramentas de análise de código para cada linguagem de programação utilizada por alunos da turma. Em especial, a ausência de ferramentas gratuitas para análise de código escrito em C# impossibilitou a aplicação da metodologia em 2 projetos de software desenvolvidos no decorrer da disciplina, destacados com * na Tabela 5.2. É importante ressaltar que a presença de código gerado por máquina ocasionou a contabilização de falsos positivos no relatório de análise, indicando a necessidade de uma etapa de pré-processamento de código fonte.

5.3 Caso de estudo III: Aplicação da metodologia em uma turma Laboratório de Programação

O experimento mais recente com a metodologia proposta foi aplicada aos arquivos de um repositórios de uma turma da disciplina de Laboratório de Programação, obrigatória para os Curso de Ciência da Computação e Sistemas de Informação da Universidade Federal de Santa Maria. Trata-se de uma disciplina de segundo período dos cursos, na qual se exige que alunos projetem e desenvolvam soluções para diversos problemas práticos utilizando-se da linguagem de programação C. Esse experimento teve como objetivo a consolidação da proposta de pesquisa, verificando se as informações agregadas pela metodologia são relevantes no acompanhamento do processo de aprendizagem dos alunos, e se tal abordagem traria benefícios para

turmas iniciais de cursos que envolvem programação.

A escolha de uma linguagem de programação próxima a linguagem de máquina se justifica pela necessidades dos alunos a se habituarem ao gerenciamento da arquitetura de um computador (KERNIGHAN; RITCHIE, 2006). Além disso, há um requisito por parte dos instrutores que os alunos desenvolvam suas soluções de ponta a ponta, sendo permitida somente o uso de funcionalidades básicas presentes na biblioteca padrão da linguagem. Ambos objetivos convergem para que o aluno desenvolva um modelo mental de como o computador realmente está operando para resolver um problema, ao mesmo tempo que a linguagem é suficientemente flexível para que o aluno compreender incrementalmente sobre o funcionamento dos vários componentes da arquitetura de um computador de maneira a aproximar o modelo mental de computador do equipamento real.

Após o convite feito ao instrutor responsável, a metodologia foi aplicada em repositórios arquivados de uma execução anterior da disciplina, num total de 26 alunos. Segundo relato do instrutor, os alunos deveriam ser capazes de implementar vários conceitos sobre entrada e saída, uso de alocação dinâmica de memória e implementação de estruturas de dados e suas operações. Ao decorrer da disciplina, os alunos desenvolveram 9 atividades práticas de programação, conforme listado na Tabela 5.3. As primeiras atividades tiveram o objetivo de relembrar conceitos fundamentais de programação imperativa com a linguagem C, e as atividades seguintes apresentaram um grau de complexidade crescente. Informação detalhadas sobre as atividades estão disponíveis no site[4].

Os defeitos detectados pela ferramenta foram os esperados frente ao contexto de um aluno de segundo período do curso se computação. Dado que a disciplina trabalha com um tema abstrato, o desenvolvimento e a implementação de algoritmos mostram-se como atividades de difícil execução, e o grau de aprendizado dentre alunos de uma mesma turma costuma ser irregular (LAHTINEN; ALA-MUTKA; JÄRVINEN, 2005), o que justifica a elevada variedade de defeitos apontado pela ferramenta.

Conforme o sumário apresentado na Tabela 5.4, as 10 detecções mais frequentes são compostas por problemas leves (como o escopo exageradamente grande de variáveis, e variá-

[4]Disponível em <http://www-usr.inf.ufsm.br/~alberto/dissertacao-ufsm2017.php?key=elc1067>.

veis não utilizadas), bem como problemas graves. Em especial, a ferramenta mostrou que as soluções apresentadas pelos alunos frequentemente apresentaram problemas de vazamento de memória (memória alocada dinamicamente não foi liberada) ou ainda vazamento de recursos (descritores de arquivos não foram fechados).

Parte das detecções apontadas pela ferramenta estavam presentes nas bibliotecas desenvolvidas pelo instrutor da disciplina, e que tinham o objetivo de auxiliar o desenvolvimento das soluções dos alunos. Conforme o exemplo ilustrado no Algoritmo 2, a inicialização e atualização da variável `vec` depende da linha de compilação utilizada. Alguns compiladores omitem a execução da macro `assert` caso alguma otimização de código esteja habilitada.

Os resultados obtidos com auxílio das ferramenta foram submetidos para apreciação do instrutor da disciplina. Segundo as considerações do instrutor a metodologia foi bem aceita mas com algumas ressalvas. A implementação atual da ferramenta foi configurada para filtrar avisos referentes a estilo de codificação. Assim, nenhuma detecção referente a inconsistências do estilo de codificação foram apresentadas, que é justamente um dos pontos considerados pelo

Assunto	Objetivos
Arquivos, entrada e saída	Operar sobre os dados obtidos de uma série de arquivos de texto
Alocação dinâmica de memória	Gerir o uso de memória dinâmica
Tipo abstrato de dados	Conceito de interfaces de programação por meio de ponteiros opacos
Programação do jogo ‘paciência’	Redução de acoplamento; Desenvolvimento de software com MVC
Editor de texto com listas encadeadas	Retomar conceito das atividades práticas anteriores
Árvore de expressão aritmética	Aplicação de árvores binárias, de busca e balanceadas
Resolução de árvores de expressão aritmética	Operações sobre estruturas de dados do tipo árvore
Grafo com lista de adjacências	Introdução de conceitos e modelos para representação computacional de grafos
Caminho mais curto em grafo não-direcionado	Operação sobre estruturas de dados do tipo grafo

Tabela 5.3 – Lista de atividades práticas desenvolvidas durante a disciplina.

# Ocorrências	Defeito
1729	Funções não utilizadas
236	Variáveis definidas em escopo muito amplo
138	Erro do uso das máscaras de formato com `scanf`/`printf`
54	Uso de funções de entrada de dados inseguras
33	Uso de variáveis não inicializadas
31	Variáveis declaradas mas não utilizadas
26	Variáveis declaradas e inicializadas, mas não utilizadas
21	Vazamento de memória
15	Vazamento de recurso (descritores de arquivo)
12	Uso de `assert` modifica uma variável dependendo do comando de compilação utilizado

Tabela 5.4 – Dez detecções mais frequentes contabilizadas pela ferramenta.

instrutor da disciplina. Além disso, o instrutor apresentou questionamentos sobre a validação das soluções dos alunos, no sentido de verificar se a saída do programa condiz com o resultado esperado para uma série de entradas conhecidas. Tal funcionalidade trata-se da análise dinâmica da aplicação, que em geral é conduzida pelo professor juntamente à ferramenta Valgrind (NETHERCOTE; SEWARD, 2007).

Segundo as considerações do instrutor, a ferramenta mostrou-se útil e sua proposta poderia cobrir uma das lacunas presente no âmbito acadêmico, que é a falta de ferramentas para auxiliar os instrutores de uma disciplina na atividade de avaliação de código fonte dos alunos. No entanto, houve crítica quanto a forma com que análises foram conduzidas visto que o *feedback* fornecido pelas ferramentas de análise empregadas trouxe indicativos de potenciais problemas com o código fonte em um formato que tem maior utilidade para o aluno que para o instrutor.

5.4 Considerações do capítulo

Neste capítulo, foram apresentados três casos de estudo. O método proposto foi aplicado em disciplinas distintas, que diferem consideravelmente entre si por serem disciplinas iniciais, intermediárias, e avançadas de cursos relacionados à Computação.

Algoritmo 2 Exemplo de código em C que faz uso indisciplinado da macro `assert`. Conforme compilador e otimizações utilizadas, a omissão da macro pode causar a introdução de defeitos no código fonte.

```
#include <stdio.h>
#include <assert.h>

#define SIZE 100

/*
 * assfault.c: Programa que demonstra o efeito de uso de 'assert'
 * conforme os parâmetros da linha de compilação.
 *
 * Compilando assim resulta em uma aplicação funcional:
 *    $ gcc -o assfault assfault.c
 *
 * Se compilado com '-D NDEBUG', a aplicação deixa de funcionar:
 *    $ gcc -o assfault assfault.c -D NDEBUG
 *
 * 'NDEBUG' é implícito em alguns compiladores quando a otimização
 * de código está habilitada.
 */
int main() {
   int *vec;

   /*
    * Garante que houve sucesso com a alocação
    * de memória ou aborta o programa.
    */
   assert ((vec = malloc(sizeof(int) * SIZE)) != NULL);

   free(vec);
   return 0;
}
```

O método possui poucas dependências. Entre os requisitos estão o uso de repositório de versionamento de código, a disponibilidade de ferramentas de análise de código para as linguagens de programação de interesse, e alguma configuração do instrutor de maneira a ajustar o comportamento da ferramenta ao cronograma da disciplina. Após essa configuração inicial, a ferramenta opera de maneira automatizada, e basta que o instrutor a execute periodicamente. Caso seja de interesse, o instrutor pode implementar suas próprias ferramentas de análise de código e integrá-las à arquitetura do método.

Embora não tenha possível aferir com precisão sobre a realização de todos os objetivos de cada caso de estudo, verificou-se o interesse pela abordagem tanto por instrutores quanto alunos. As informações provenientes do relatório de análise sugere que o método é capaz de colaborar para o aperfeiçoamento das habilidades dos alunos de programação, bem como fornecer

ao instrutor da disciplina informações sobre o andamento das atividades práticas. O emprego do método permanece válido mesmo para disciplinas cujo o aperfeiçoamento das habilidades de programação dos alunos não é o principal objetivo.

A interface de usuário interativa permite a verificação do resultado da análise com enfoque no aluno, em um grupo de alunos, ou na turma como um todo. Também é possível visualizar o relatório de detecções pelo tipo de defeito encontrado, e filtrar as ocorrências a um conjunto de códigos fonte de um repositórios em específico. Tais formas de visualização auxiliam o instrutor a compreender as dificuldades que os alunos estão enfrentando na realização das atividades práticas.

Entre as ameaças ao emprego da metodologia proposta estão a inexistência de analisadores para algumas linguagens de programação específicas, e a presença de código gerado por máquina, que pode introduzir falsos positivos no relatório de análise. A etapa de configuração podem ocasionar alguma resistência à implantação da ferramenta, que pode ser mitigado pelo uso de *git hooks* para atualização automática do relatório de detecções. Por fim, a interface de visualização mostrou-se deficiente na filtragem dos resultados conforme a gravidade das detecções, destacando a necessidade de filtros para omissão de resultados irrelevantes ao contexto das atividades e aos critérios observados pelo instrutor na avaliação das soluções.

6 CONCLUSÃO

Foi proposto neste trabalho um método para análise automatizada de repositórios de código de alunos de programação usando-se ferramentas de análise estática de código, com o propósito de se fornecer um *feedback* para alunos de programação. A metodologia proposta sugere que as atividades didáticas sejam desenvolvidas dentro de um repositório *git* fazendo uso de serviços de hospedagem *online* como *Github* e *Bitbucket*.

A arquitetura da ferramenta que materializa o método permite a análise simultânea de vários repositórios de código de uma turma de programação, fazendo uso de ferramentas de análise de código convencionais e agregando os resultados em uma interface de visualização interativa. A implementação do método mostrou-se flexível e sua arquitetura permite que outras ferramentas de análise de código sejam empregadas com facilidade, permitindo seu emprego em contextos distintos para os quais foi originalmente proposto. A exemplo, sua natureza modular possibilitou a rápida construção de mecanismos para comunicação com servidores de hospedagem de código.

Os experimentos de aplicação da metodologia mostraram que a ferramenta é útil, e permite que o instrutor das disciplinas de programação tenham uma noção geral do progresso das implementações das soluções dos alunos, tornando rápida a verificação de tipos comuns de defeitos de código. Por tratar-se da automatização de um processo, a ferramenta se mostrou capaz em reduzir o tempo despendido pelo instrutor das disciplinas na correção das atividades, o que também possibilitou um *feedback* rápido para os alunos envolvidos. Além disso, a ferramenta permitiu a antecipação de um parecer sobre as soluções, possibilitando que os alunos façam correções em suas soluções antes do prazo de entrega final das atividades.

Embora a percepção geral sobre a utilidade da ferramenta tenha sido majoritariamente positivos, diversas considerações foram feitas em relação às suas limitações. O emprego de ferramentas de verificação de código mais específicas permitiria a detecções mais relevantes conforme os critérios avaliados pelo instrutor da disciplina. Além disso, os componentes de visualização mostraram-se pouco flexíveis por não permitirem livremente a filtragem e processamento dos resultados encontrados.

A metodologia também trouxe benefícios secundários. A frequente interação dos alunos

com a ferramenta *git* promoveu um aperfeiçoamento de uso de versionamento de código, algo que é apresentado pela literatura como uma deficiência comum a alunos de graduação recém formados. A adoção de repositórios centraliza todo o desenvolvimento dos trabalhos dos alunos, flexibilizando a cooperação entre membros das atividades em grupo. O uso de *git* configurou um mecanismo homogêneo de divulgação das soluções, facilitando sua aquisição por parte do instrutor da disciplina.

6.1 Produção científica

O autor desta dissertação atuou na coautoria e apresentação dos trabalhos seguintes trabalhos.

- Charão, A. S., Neto, A. F. K, Stein, B. O., Barcelos, P. P. A. *Hall of Fame*/*Shame*: um Padrão Pedagógico para o Ensino de Programação. Em: Sociedade Brasileira De Computação (Ed.). Anais do XXXVI Congresso da Sociedade Brasileira de Computação. Porto Alegre, Rio Grande do Sul, 2016. p. 2166–2175.

- Charão, A. S., Neto, A. F. K, Stein, B. O., Barcelos, P. P. A. *Hall of Shame/Fame: a Pedagogical Pattern for Computer Programming Classes.* Em: *Writers' Workshop of 23rd PLoP*. Monticello, Illinois, USA, 2016. Disponível em <http://www-usr.inf.ufsm.br/~alberto/dissertacao-ufsm2017.php?key=plop2016-ww>.

6.2 Trabalhos Futuros

Os experimentos conduzidos apontaram a necessidade de melhoria da metodologia e da implementação desenvolvida. Em todos os experimentos verificou-se a falta de controle sobre os tipos de análise executadas, como por exemplo a ausência de um filtro para controle de severidade dos problemas reportados. Além disso, a verificação dinâmica das aplicações mostrou-se uma necessidade, e que pode ser implementada usando-se instrumentação automática de forma a coletar métricas sobre o funcionamento das aplicações frente a de entrada conhecidas e saídas esperadas.

A implementação do suporte à transferência de recursos via *OAuth* pode ser melhorado de forma a permitir o acesso aos repositórios de código remoto de maneira automática. Atualmente, a implementação do protocolo pelo serviço de hospedagem *Bitbucket* dificultou o acesso aos dados protegidos por demandar do programador a implementação de um servidor de autorização para uso da própria API do serviço de hospedagem.

Uma vez garantida a automatização da comunicação com os serviços de hospedagem, o relatório de detecções pode ser atualizado automaticamente cada vez que um aluno envia uma nova versão de sua solução para o repositório remoto. Isso pode ser feito por meio do mecanismo de gatilhos suportado pela ferramenta *git*, reduzindo a necessidade de manutenção pelo instrutor da disciplina (CHACON; STRAUB, 2014).

APÊNDICE A – ARTIGO APRESENTADO NO WORKSHOP SOBRE EDUCAÇÃO EM COMPUTAÇÃO DO XXXVI CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO

Hall of Fame/Shame: um Padrão Pedagógico para o Ensino de Programação

Andrea S. Charão[1,2], Alberto F. Kummer Neto[2]
Benhur de O. Stein[1,2], Patrícia P. de A. Barcelos[1]

[1] Departamento de Linguagens e Sistemas de Computação

[2]Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria
Santa Maria, RS, Brasil

{andrea, alberto, benhur, pitthan}@inf.ufsm.br

Resumo. *Padrões pedagógicos visam aproveitar o conhecimento especializado em práticas de ensino e aprendizagem, em um formato que favoreça sua reutilização. Neste trabalho, propõe-se um padrão pedagógico voltado ao ensino de programação, em cursos de nível superior na área de Computação. Este padrão é centrado na exposição e discussão de bons e maus exemplos de códigos produzidos pelos alunos, formando o que denominou-se respectivamente de "Hall of Fame" e "Hall of Shame". Ao longo do texto, apresenta-se uma caracterização do padrão, relacionando-o com outros padrões pedagógicos descritos na literatura, e descreve-se sua aplicação no ensino de programação orientada a objetos, em uma instituição de ensino superior. Os resultados indicam uma avaliação positiva do padrão pelos alunos e revelam sua eficácia em incentivar boas práticas de programação.*

Abstract. *Pedagogical patterns propose to take advantage of the expertise in teaching and learning practices, in an organized manner that can be easily reused. In this paper, we propose a pedagogical pattern targeted to teaching programming in higher level courses in Computer Science. This pattern is centered on the presentation and discussion of good and bad examples of codes produced by the students, forming what was named respectively "Hall of Fame" and "Hall of Shame". Throughout the text, we presents a characterization of HoFS pattern, relating it to other pedagogical patterns described in the literature. We also report its application in teaching object oriented programming in a higher education institution. The results indicate a positive evaluation by students and reveal the pattern helps to encourage good programming practices.*

Introdução

Os processos de ensino-aprendizagem de programação de computadores são alvo de discussões recorrentes em comunidades de pesquisadores e professores da área de Computação [Pears et al. 2007, Aureliano e Tedesco 2012]. É consenso que a programação impõe várias dificuldades a alunos iniciantes e, aos que já possuem alguma experiência prévia, apresenta-se também o desafio de aprimorar suas habilidades e aprender novos paradigmas [Jenkins 2002, Robins et al. 2003, Lahtinen et al. 2005, Tan et al. 2009]. Aos professores e instituições de ensino, cabe a responsabilidade de permanentemente refletir

sobre as metodologias e as práticas de ensino-aprendizagem de programação, que envolve também a tomada de decisões com impacto em todo o processo [Denning 1989]. Questões frequentes neste contexto incluem, por exemplo: Que linguagens de programação adotar para iniciantes? Que estratégias de ensino-aprendizagem são mais eficazes?

Em buscas de respostas para estas e outras questões, há exemplos de pesquisadores que têm se dedicado a compreender melhor os processos cognitivos envolvidos na aprendizagem de programação [Soloway e Spohrer 1988, Winslow 1996], enquanto outros investigam e propõem formas de lidar com as dificuldades recorrentes [Wilson e Shrock 2001, Caspersen e Kolling 2009]. Nessa segunda direção, há diversos autores que sustentam uma abordagem centrada nos chamados "padrões pedagógicos" (*pedagogical patterns*) [Sharp et al. 1996], originários de uma comunidade interessada no ensino-aprendizagem de programação orientada a objetos, e gradativamente estendidos para um escopo mais amplo [Larson et al. 2008, Köppe 2015]. Essa abordagem propõe-se a capturar boas práticas de ensino-aprendizagem em domínios específicos, sob uma forma compacta e organizada, para que possa ser facilmente comunicada e reutilizada. Seu foco principal é, portanto, facilitar a disseminação de boas práticas.

Desde a proposta inicial, em 1996, vários foram os padrões pedagógicos descritos, catalogados e publicados, com diferentes intenções. No domínio específico de aprendizagem de programação, há padrões variados, como por exemplo: *Mistake* [Bergin 2000], que propõe que os alunos criem programas (ou outros artefatos) com erros; e *Show Programming*, que sugere mostrar programação em tempo real aos alunos, e não somente slides sobre programação [Schmolitzky 2007]. Conforme autores da área [Bergin et al. 2012], alguns padrões podem parecer triviais para educadores experientes, mas se tornam referências úteis para quem tem menos experiência ou busca diversificar suas práticas. A forma sucinta de apresentação dos padrões facilita este compartilhamento; por outro lado, a ausência de dados complementares sobre avaliação de certos padrões pode suscitar dúvidas quanto à sua efetividade.

Neste artigo, apresenta-se um padrão pedagógico voltado ao ensino de programação, em cursos de nível superior na área de Computação. Este padrão, que denominou-se "Hall of Fame/Shame" (HoFS), é centrado na exposição e discussão de bons e maus exemplos de códigos produzidos pelos alunos. Sua motivação é principalmente estimular os alunos a aprimorarem suas habilidades em programação. Além de descrever o padrão de forma sucinta, num formato que facilite o reuso, apresenta-se também resultados experimentais sobre sua aplicação prática numa instituição de ensino superior.

Este artigo encontra-se assim organizado: na seção 2 apresenta-se sucintamente uma visão histórica de padrões pedagógicos no ensino de programação, indicando as principais fontes de referência sobre o assunto, até os dias atuais; na seção 3 apresenta-se o padrão HoFS num formato tipicamente utilizado pela comunidade da área; na seção 4 descreve-se a aplicação e a avaliação do padrão em uma instituição de ensino superior; na seção 5, por fim, apresentam-se considerações finais sobre o trabalho.

Padrões Pedagógicos no Ensino de Programação

Padrões pedagógicos (*pedagogical patterns*) [Sharp et al. 1996] surgiram no contexto de discussões sobre ensino de programação orientada a objetos e suas tecnologias. A inspiração para isso veio dos chamados padrões de projeto (*design patterns*) [Gamma et al. 1993],

que constituem soluções bem sucedidas e reusáveis para problemas no contexto da orientação a objetos. A ideia inicial dos padrões pedagógicos era, portanto, usar essa mesma abordagem para catalogar soluções reusáveis a problemas comumente encontrados no ensino desse paradigma.

Gradativamente, a ideia dos padrões pedagógicos estendeu-se a um escopo mais amplo, capturando soluções para problemas encontrados em diferentes situações e domínios além da orientação a objetos. Pesquisadores e educadores passaram a propor padrões em conferências tais como PLoP (Pattern Languages of Programs) e EuroPLoP (European Conference on Pattern Languages of Programs), chegando a conferências tradicionais sobre ensino de computação, tais como SIGCSE (ACM Technical Symposium on Computing Science Education) e ITiCSE (ACM Conference on Innovation and Technology in Computer Science Education). Os padrões foram sendo catalogados em sites[1][2] e, mais recentemente, vários deles forem reunidos em livros [Bergin et al. 2012, Mor et al. 2014].

No cenário brasileiro, um dos primeiros trabalhos que encontramos sobre padrões pedagógicos foi uma dissertação de mestrado [de Oliveira Neto 2000], seguida mais adiante por publicações no Simpósio Brasileiro de Informática na Educação [de Barros et al. 2004, Medeiros et al. 2007]. No Workshop de Educação em Computação, os padrões pedagógicos também são abordados em alguns trabalhos [de Barros e Delgado 2006, dos Santos Júnior et al. 2009]. De forma geral, nota-se que esses trabalhos em âmbito nacional não propõem ou avaliam padrões, mas sim ferramentas ou estratégias de apoio ao aprendizado com base em padrões pedagógicos.

Um aspecto importante sobre os padrões pedagógicos é seu formato de descrição, que é sempre sucinto e organizado para facilitar o reuso. Esse formato não é rígido, mas comumente inclui as seções: nome do padrão, objetivo/motivação, aplicabilidade, estrutura, consequências, implementação, recursos necessários, exemplos e padrões relacionados [Sharp et al. 1996]. Alguns autores organizam a descrição em menos seções, porém mantendo essas informações básicas [Schmolitzky 2007]. Para facilitar a comunicação, a redação do padrão geralmente usa o pronome "você" para dirigir-se diretamente ao professor.

O Padrão "Hall of Fame/Shame"

O padrão proposto é descrito a seguir, usando um formato semelhante ao utilizado por [Bergin 2000].

- **Nome:** HALL OF FAME/SHAME
- **Problema:** Alunos que vencem barreiras iniciais no aprendizado de programação ficam satisfeitos quando conseguem resolver novos problemas e seus programas funcionam. No entanto, esses alunos não costumam ser críticos quanto a seus códigos, ou julgam-se incapazes de produzir algo melhor no tempo disponível. Muitas vezes, seus programas são resultado de tentativas e erros, revelando más práticas de programação que são repetidas sucessivamente. Os alunos recebem orientações sobre boas práticas, porém não as relacionam com os problemas que devem resolver. O *feedback* fornecido para esses alunos ocorre muitas vezes sob

[1] `http://www.pedagogicalpatterns.org`
[2] `http://educationalpatterns.org`

forma de uma nota, um comentário ou um gabarito, que não estimulam o aluno a vislumbrar diferentes alternativas de soluções, algumas melhores que outras.

- **Contexto:** Você está ensinando um novo tópico de programação a alunos que já adquiriram alguma experiência com uma linguagem (tipicamente, alunos de segundo ano de cursos de graduação em Computação). Por exemplo, podem ser novas metodologias e técnicas de programação, um novo paradigma ou uma nova linguagem. Você requer que os alunos produzam código, de diferentes tamanhos, para resolver problemas propostos. Seu grupo de alunos tem de 20 a 40 indivíduos (não é uma turma muito pequena, nem muito grande). Você deseja estimulá-los a conhecer e adotar boas práticas, aproveitando também suas experiências anteriores em programação.
- **Solução:** Depois que o grupo de alunos entregar suas soluções para um dado problema, analise os códigos buscando identificar boas e más práticas de programação. Preferencialmente, permita que os alunos entreguem soluções parciais antes da entrega final. A cada entrega, escolha trechos de códigos para formar uma exposição, separada em duas "galerias": *Hall of Fame* (boas práticas) e *Hall of Shame* (más práticas). Em aula, para cada aspecto da solução, apresente e comente os trechos em ambas as galerias, sem revelar os autores, discutindo com os alunos os motivos para classificar cada trecho em uma ou outra galeria. Apresente também trechos de código sem classificação e discuta com os alunos o enquadramento como *Fame* ou *Shame*. Use recursos visuais contrastantes para identificar ambas as galerias. Ressalte que, num mesmo programa, é possível encontrar exemplos de boas e más práticas, reforçando que a galeria não é de bons ou maus programadores, mas sim de trechos de código com bons e maus exemplos. Faça com que a galeria fique disponível para consultas futuras. Repita este padrão sempre que possível.
- **Discussão:** Este padrão provê *feedback* aos alunos com base em problemas que eles se dedicaram a resolver. Ao contrário de recomendações gerais e abstratas sobre boas práticas, o padrão garante exemplos reais, provenientes do próprio grupo de alunos. Com a separação clara e repetitiva entre *Fame/Shame*, a intenção é estimular processos cognitivos que, nas próximas atividades de programação, resultem na adoção de uma boa prática ou, no mínimo, alimentem o senso crítico dos alunos frente a seus programas. Este padrão não visa substituir uma avaliação individual dos programas (que, por outro lado, nem sempre é possível), mas constitui uma alternativa para prover *feedback* de forma rápida e proveitosa para o grupo. O padrão tem também limitações, a saber: (i) é necessário um tempo considerável para preparar e expor/comentar as galerias, imediatamente após a entrega das soluções pelos alunos; (ii) em grupos de alunos com experiências muito homogêneas, a diversidade de exemplos produzidos pode ser pequena.
- **Recursos necessários:** Para exibição das galerias, é necessário projetor multimídia para uso em aula, além de infraestrutura para disponibilizar o material para consulta futura. Os problemas propostos aos alunos devem ser especificados de tal forma a dar margem suficiente para que se produzam soluções diversificadas. O apoio de um monitor ou assistente é recomendável na preparação das galerias, para que a exibição ocorra o mais rápido possível, idealmente em um encontro subsequente à entrega das soluções.
- **Exemplos:** Um exemplo de aplicação do HoFS é no ensino de programação procedimental em linguagem C, depois que os alunos já souberem estruturar o

código em funções e procedimentos. Neste ponto, é comum que os alunos tenham que programar um jogo simples, ou um editor de textos. A galeria *Shame*, por exemplo, possivelmente irá revelar trechos de código com os seguintes problemas: divisão do código em poucos subprogramas, procedimentos que realizam mais ou menos instruções do que o nome sugere, procedimentos que acessam variáveis globais quando deveriam se ater a seus argumentos, código mal endentado, etc. Na galeria *Fame*, ficarão os trechos de código com características positivas, opostas a essas.

Outro exemplo de aplicação é no ensino de programação funcional, em linguagens Haskell ou Lisp. Habitualmente, propõem-se que os alunos resolvam vários tipos de problemas com listas, envolvendo percursos recursivos ou via funções de ordem superior (*higher order functions*). Quando os alunos já conhecem ambas as alternativas, a galeria *Shame* poderá apontar, por exemplo, trechos de código que usaram recursão explícita para implementar o que seria mais facilmente escrito e reusável usando funções de ordem superior, como `map` ou `filter`.

- **Padrões relacionados:** A aplicação do padrão HoFS supõe que os alunos devam produzir código para implementar algum software, mesmo que parcialmente. Assim, padrões relacionados ao ensino de desenvolvimento de software podem ser aliados no processo de ensino-aprendizagem [Schmolitzky 2007]. Além disso, o padrão HoFS é centrado na ideia de que os alunos se beneficiam de *feedback* frequente para progredirem e desenvolverem suas habilidades em programação. Desta forma, outros padrões pedagógicos que também focam em *feedback* contínuo [Larson et al. 2008, Köppe et al. 2015] podem complementar a aplicação do HoFS.

Aplicação e Avaliação do HoFS

O padrão aqui apresentado foi aplicado em 4 semestres de uma disciplina que trata de Programação Orientada a Objetos (POO), oferecida na Universidade Federal de Santa Maria, para alunos cursando a partir do terceiro semestre dos cursos de graduação em Paradigmas de Programação. Essa disciplina proporciona um primeiro contato com POO aos alunos e, para isso, adota a linguagem Java. Observou-se, nas primeiras instâncias de aplicação, que o padrão pareceu ter aceitação positiva por parte dos alunos, evidenciada principalmente por: (i) uma maior atenção e participação dos alunos em sala de aula durante a exposição dos códigos, em contraste com exposições de conceitos ou trechos de código do professor e (ii) interesse de alguns alunos em saber se seus códigos entrariam ou não para as galerias.

Diante dessas observações, na última oferta da disciplina buscou-se realizar uma avaliação do padrão HoFS com critérios e procedimentos previamente estabelecidos, visando coletar mais dados sobre seu impacto no aprendizado de POO. O restante desta seção se refere à avaliação do HoFS aplicado à última turma.

Na literatura sobre padrões pedagógicos, não se encontram critérios e procedimentos sistematicamente utilizados para avaliar padrões propostos. Assim, decidiu-se avaliar 2 aspectos complementares que, em nossa experiência, poderiam fornecer indicadores sobre a eficácia (ou não) do padrão. Um desses aspectos foi a **aceitação** da abordagem por parte dos alunos, avaliada por meio de um questionário respondido ao final da disciplina. Outro aspecto foi a **produção** de código pelos alunos, contendo bons e/ou maus exemplos de práticas de programação, avaliada pela análise continuada do código produzido para

resolver problemas propostos. Entende-se que esses aspectos são complementares pois, por exemplo, os alunos poderiam avaliar positivamente a experiência, mas suas produções de código não revelarem melhoria de suas práticas.

Avaliação da aceitação: método e resultados

Para avaliar a aceitação, elaborou-se um questionário dividido em 2 partes, cada parte apresentada sob forma de itens da escala de Likert de 5 pontos, variando de -2 (discordo totalmente) a +2 (concordo totalmente). A primeira parte continha 5 questões relacionadas à percepção geral do padrão pelos alunos, relacionando-o com outras experiências de aprendizado. A segunda parte apresentava 4 questões sobre o formato utilizado para exibição e discussão das galerias, que compreendeu aulas expositivas e participativas usando *slides* com trechos de códigos, seguida de publicação do material *online* para consultas futuras (conforme apresentado na seção 3). Nessa segunda parte, buscou-se levantar a necessidade de possíveis ajustes na aplicação do padrão.

O questionário foi respondido anonimamente por 15 alunos, num total de 22 que cursaram a disciplina até o final. Esses alunos tiveram aulas teóricas e práticas, realizaram exercícios e desenvolveram 2 trabalhos/projetos individuais de POO. O padrão HoFS foi aplicado em 3 oportunidades: (1) após a entrega do primeiro trabalho, (2) após uma entrega parcial do segundo trabalho (50% dos requisitos implementados) e (3) após a entrega final do segundo trabalho. A avaliação dos alunos na disciplina considerou seu desempenho nos trabalhos e em uma avaliação escrita. O questionário foi apresentado após a divulgação de notas, ao fim do semestre.

A figura 1 apresenta os resultados obtidos na primeira parte do questionário. Pode-se notar que uma das questões continha uma proposição com viés negativo ("Achei ruim ver meus erros expostos, mesmo anonimamente"). De forma geral, as respostas dos alunos reforçaram a hipótese inicial sobre a boa aceitação da abordagem. Além disso, os alunos perceberam um caráter original na abordagem, ao não concordarem com a afirmação "Já tive aulas assim em outras disciplinas".

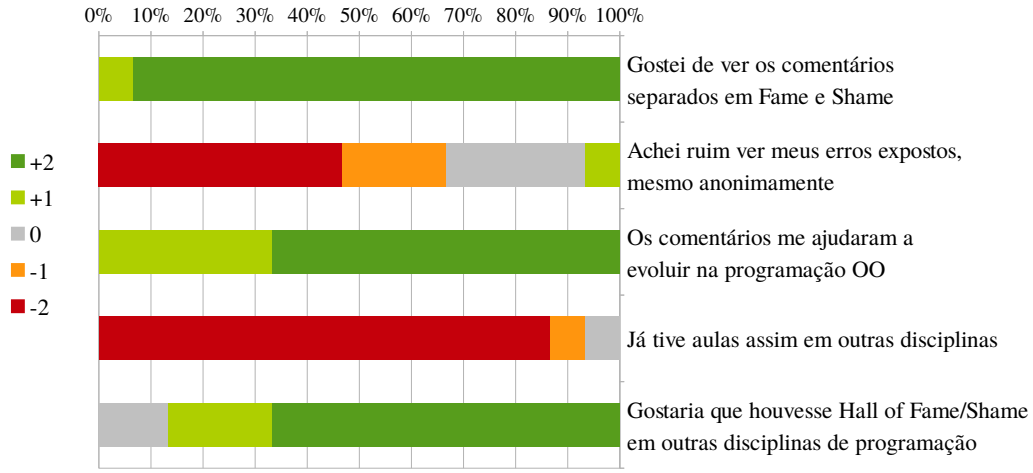


Figura 1. Avaliação da percepção geral do padrão pelos alunos

Na figura 2, apresenta-se as respostas para as questões sobre o formato de aplicação do padrão. Neste quesito, de forma geral, nota-se também que houve boa aceitação do

formato, porém alguns itens merecem mais atenção: (1) os comentários *Fame/Shame* **antes** da entrega final de um trabalho foram entendidos como mais úteis do que aqueles apresentados **após** a entrega; (2) muitos alunos responderam "Concordo parcialmente" ao item "O material disponibilizado para consultas posteriores foi suficiente". O item (1) sugere que, quando possível, o padrão seja aplicado enquanto os alunos ainda estão programando suas soluções. O item (2), por sua vez, revela um possível descompasso entre o aproveitamento das galerias durante e após as aulas. Os motivos disso podem ser investigados em novas experiências mas, como alternativa imediata, estima-se que a disponibilização das galerias em formato de vídeo poderia reduzir esse descompasso.

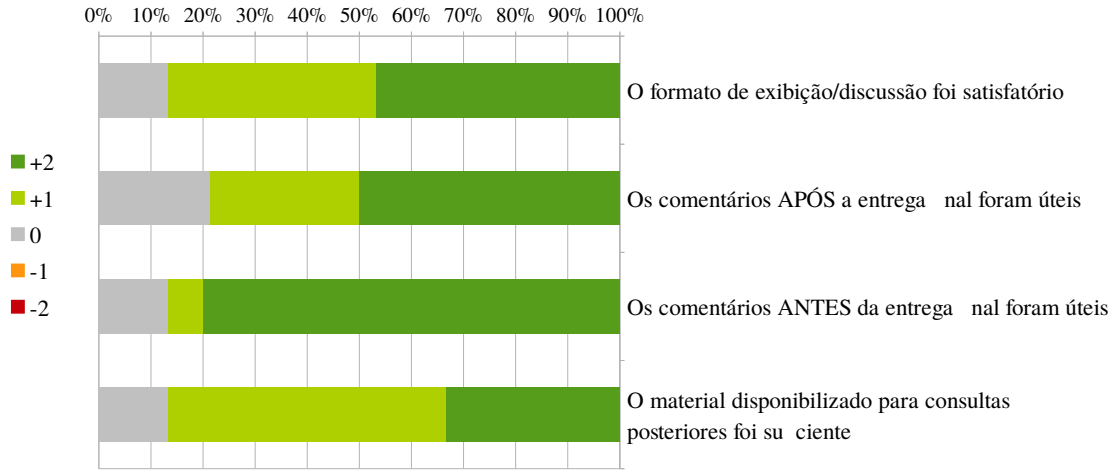


Figura 2. Avaliação do formato adotado para as aulas que seguiram o padrão HoFS

Avaliação da produção: método e resultados

Para avaliar o impacto do HoFS na produção dos alunos, considerou-se 2 trabalhos/projetos de POO desenvolvidos pelos alunos. Os programas referentes a esses projetos foram criados pelos alunos no prazo de 1 e 3 semanas, respectivamente, e entregues ao professor em 3 oportunidades, conforme explicado na seção anterior.

O primeiro projeto tinha como tema a simulação de entradas e saídas de veículos num estacionamento, compreendendo o cálculo do valor a ser cobrado para cada veículo. Esse projeto não explicitava requisitos quanto à interface com o usuário, enfatizando principalmente o projeto e implementação orientados a objetos para o domínio em questão. O padrão HoFS foi aplicado após a entrega final deste trabalho.

O segundo projeto, por sua vez, consistiu em desenvolver uma aplicação com interface gráfica para *desktop*, em Java, visando gerenciar um cadastro de postos de combustíveis em um município, com seus respectivos históricos de preços. O projeto deveria utilizar o padrão MVC (*Model-View-Controller*) e persistir os dados em arquivo e/ou banco de dados. Para este trabalho, aplicou-se o padrão HoFS antes e após a entrega final.

Para análise da produção dos alunos, considerou-se todos os códigos entregues pelos alunos nos 3 momentos descritos anteriormente, num total de 17 e 20 projetos entregues, respectivamente, para o primeiro e segundo trabalhos. Para cada exposição

Fame e *Shame* em aula, escolheu-se em média 15 trechos de código para discussão, entre bons e maus exemplos, versando sobre diversos aspectos da POO.

Pelos temas dos projetos, há boas práticas de POO que não eram necessárias nos 2 trabalhos (por exemplo, classes de acesso aos dados persistidos em arquivo ou banco de dados). Por outro lado, algumas práticas relacionadas aos fundamentos da POO deveriam estar presentes em todos os trabalhos. Nessa situação, a hipótese era de que o padrão HoFS poderia estimular os alunos a corrigir suas más práticas, entre um trabalho e outro. Assim, dentre todas as boas e más práticas identificadas nos trabalhos, escolheu-se 3 más práticas cujas ocorrências foram contabilizadas nos códigos **antes** do primeiro HoFS (isto é, no primeiro trabalho entregue) e **depois** do último HoFS (isto é, no último trabalho entregue). Essas práticas foram: (1) duplicação de código ao invés de polimorfismo, (2) violação do princípio da responsabilidade única na implementação de uma classe e (3) visibilidade pública para atributos de instância.

Na tabela 1, apresenta-se as ocorrências identificadas, sendo que um aluno pode ser responsável por mais de uma ocorrência. Pode-se notar que as ocorrências destas más práticas diminuíram, como esperado. Sabe-se que os processos de ensino-aprendizagem são por natureza complexos e, por isso, não se pode afirmar com certeza que a diminuição de más práticas seja consequência unicamente da aplicação do HoFS. Mesmo assim, interpreta-se estes resultados como indícios de que o padrão pode ser eficaz. Mais importante do que isso é o evidente progresso demonstrado pelos alunos e estimulado pelo HoFS.

Má prática	Ocorrências antes	Ocorrências depois
Duplicação de código	5	1
Violação do princípio da responsabilidade única	10	2
Visibilidade pública para atributos de instância	3	0

Tabela 1. Ocorrências de más práticas nos códigos antes e depois da aplicação do HoFS

Considerações Finais

Neste trabalho, buscou-se caracterizar um padrão pedagógico focado no ensino de programação. Esse padrão relaciona-se com outros voltados ao *feedback* contínuo, porém difere desses à medida em que se fundamenta essencialmente na análise e exibição de trechos de código que representam boas e más práticas de programação.

O padrão foi aplicado numa instituição de ensino superior e, na última instância de aplicação, realizou-se uma avaliação baseada na aceitação do padrão pelos alunos e na análise de códigos por eles produzidos. Os resultados apontam para a eficácia da abordagem e destacam seu caráter original, embora seja necessário repetir o experimento para aferir tais afirmações.

Tem-se consciência de que, isoladamente, um padrão pedagógico não pode garantir o sucesso do processo de ensino-aprendizagem. No entanto, padrões documentados, aplicados e discutidos podem constituir recursos úteis professores que procuram diversificar e aprimorar suas práticas. Assim, como perspectivas futuras relacionadas a este trabalho, pode-se vislumbrar que mais professores também compartilhem padrões originais e/ou relatem experiências de aplicações de padrões pedagógicos em Computação.

Visando facilitar a reprodução do padrão em outros tempos e espaços, as galerias produzidas e outros materiais utilizados encontram-se disponíveis em: `http://www-usr.inf.ufsm.br/~andrea/elc117-2015b/`.

Referências

Aureliano, V. e Tedesco, P. (2012). Ensino-aprendizagem de programação para iniciantes: uma revisão sistemática da literatura focada no SBIE e WIE. In *Anais do Simpósio Brasileiro de Informática na Educação (SBIE)*.

Bergin, J. (2000). Fourteen pedagogical patterns. In *Proceedings of the Fifth European Conference on Pattern Languages of Programs*, Irsee, Germany.

Bergin, J., Eckstein, J., Volter, M., Sipos, M., Wallingford, E., Marquardt, K., Chandler, J., Sharp, H., e Manns, M. L. (2012). *Pedagogical Patterns: Advice For Educators*. Joseph Bergin Software Tools.

Caspersen, M. E. e Kolling, M. (2009). Stream: A first programming process. *Trans. Comput. Educ.*, 9(1):4:1–4:29.

de Barros, L. N. e Delgado, K. V. (2006). Aprendizado de Programação. In *Anais do Workshop sobre Educação em Computação (WEI)*, pages 31 – 40.

de Barros, L. N., Delgado, K. V., e Machion, A. C. G. (2004). An ITS for programming to explore practical reasoning. In *Anais do Simpósio Brasileiro de Informática na Educação (SBIE)*.

de Oliveira Neto, J. A. (2000). Suporte de ferramenta de software para o padrão pedagógico aula em mapa de conceitos. Master's thesis, Universidade Federal da Paraíba, Campina Grande.

Denning, P. J. (1989). A debate on teaching computing science. *Commun. ACM*, 32(12):1397–1414.

dos Santos Júnior, G. P., Fechine, J. M., e de Barros Costa, E. (2009). Analogus: Um ambiente para auxílio ao ensino de programação orientado pelo raciocínio por analogia. In *Anais do Workshop sobre Educação em Computação (WEI)*, pages 499 – 508.

Gamma, E., Helm, R., Johnson, R., e Vlissides, J. (1993). Design patterns: Abstraction and reuse in object-oriented designs. In Nierstrasz, O., editor, *Proceedings of ECOOP'93*, Berlin. Springer-Verlag.

Jenkins, T. (2002). On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, pages 53–58.

Köppe, C. (2015). Towards a pattern language for lecture design: An inventory and categorization of existing lecture-relevant patterns. In *Proceedings of the 18th European Conference on Pattern Languages of Program*, EuroPLoP '13, pages 3:1–3:17, New York, NY, USA. ACM.

Köppe, C., Portier, M., Bakker, R., e Hoppenbrouwers, S. (2015). Lecture design patterns: More interactivity improvement patterns. In *Preprints of the 22nd Pattern Languages of Programs conference, PLoP*, volume 15.

Lahtinen, E., Ala-Mutka, K., e Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 14–18, New York, NY, USA. ACM.

Larson, K. A., Trees, F. P., e Weaver, D. S. (2008). Continuous feedback pedagogical patterns. In *Proceedings of the 15th Conference on Pattern Languages of Programs*, PLoP '08, pages 12:1–12:14, New York, NY, USA. ACM.

Medeiros, F. M., Hernández-Domínguez, A., de Medeiros, F. N., e da Silva, A. G. (2007). Um sistema de ensino na web baseado no padrão pedagógico exposição teórica-exemplos-atividade-apresentação-avaliação. In *Anais do Simpósio Brasileiro de Informática na Educação (SBIE)*.

Mor, Y., Mellar, H., Warburton, S., e Winters, N. (2014). *Practical Design Patterns for Teaching and Learning with Technology*. Sense Publishers.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., e Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '07, pages 204–223, New York, NY, USA. ACM.

Robins, A., Rountree, J., e Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13:137–172.

Schmolitzky, A. (2007). Patterns for teaching software in classroom. In *Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007)*, EuroPLoP '07, pages B5:1–B5:10, Irsee, Germany. Hillside Europe.

Sharp, H., Manns, M. L., McLaughlin, P., Prieto, M., e Dodani, M. (1996). Pedagogical patterns – successes in teaching object technology: A workshop from OOPSLA '96. *SIGPLAN Not.*, 31(12):18–21.

Soloway, E. e Spohrer, J. C. (1988). *Studying the Novice Programmer*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Tan, P. H., Ting, C. Y., e Ling, S. W. (2009). Learning difficulties in programming courses: Undergraduates' perspective and perception. In *Computer Technology and Development, 2009. ICCTD '09. International Conference on*, volume 1, pages 42–46.

Wilson, B. C. e Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, pages 184–188, New York, NY, USA. ACM.

Winslow, L. E. (1996). Programming pedagogy – a psychological overview. *SIGCSE Bulletin*, 28(3):17–22.

APÊNDICE B – ARTIGO APRESENTADO NO PATTERN LANGUAGES OF PROGRAMS CONFERENCE 2016

Hall of Shame & Fame: a pedagogical pattern for computer programming classes

Andrea S. Charão, Alberto F. Kummer Neto, Benhur de O. Stein, Patrícia Pitthan A. Barcelos
Federal University of Santa Maria

Pedagogical patterns propose to take advantage of the expertise in teaching and learning practices, in an organized manner that can be easily reused by educators. In this paper, we propose a pedagogical pattern targeted to teaching programming in higher level courses in Computer Science. This pattern is centered on the presentation and discussion of good and bad examples of code produced by the students, forming what was named respectively "Hall of Fame" and "Hall of Shame". Throughout the text, we present a characterization of HOFS pattern, relating it to other pedagogical patterns described in the literature. We also report its application in teaching object oriented programming in a higher education institution. The results indicate a positive evaluation by students and reveal the pattern helps to encourage good programming practices.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computers and Information Science Education—*Computer science education*

General Terms: Languages, Education

Additional Key Words and Phrases: Educational Patterns, Language Patterns

ACM Reference Format:

Andrea S. Charão. 2016. Hall of Shame/Fame: a pedagogical pattern for computer programming classes – HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. (October 2016), 8 pages.

1. INTRODUCTION

Computer programming teaching and learning processes are a recurring subject of research [Pears et al. 2007; Robins 2010]. It is consensus that programming brings several difficulties to beginning students and even for those who have some previous experience, as there remain challenges as improving skills and learning new paradigms[Jenkins 2002; Robins et al. 2003; Lahtinen et al. 2005; Tan et al. 2009]. Faculty also face some challenges in this context, which includes decision making that impact on the entire process [Denning 1989]. Some frequent issues in this context are, for example: 'Which programming languages adopt to teach beginners?' and 'Which teaching and learning strategies are most effective?'

Looking for answers to this kind of questions, there are researchers that have dedicated to improve the understanding of cognitive processes involved in programming learning [Soloway and Spohrer 2013; Winslow 1996], while others investigate and propose ways to handle recurrent difficulties [Wilson and Shrock 2001; Caspersen and Kolling 2009]. To the latter, there are several authors that support an approach centered in "pedagogical patterns" [Sharp et al. 1996]. Pedagogical patterns were developed by a community interested in object oriented computer programming and gradually extended to a larger scope [Larson et al. 2008; Köppe 2015].

Author's addresses: {andrea, alberto, benhur, pitthan}@inf.ufsm.br, Federal University of Santa Maria, Rio Grande do Sul, Brazil.

This approach suggests the capture of good teaching and learning practices of specific domains in a organized and compact way that can be easily reused. Therefore, its main objective is to promote dissemination of good practices.

Since the initial propose, in 1996, several pedagogical patterns were described, cataloged and published, with several intentions. Specifically to programming learning domain, there are varied patterns, for example: *Mistake* [Bergin 2000], which proposes that the students develop programs (or other artifacts) with errors; and *Show Programming*, which suggests exposure of real time programming to the students–instead of slides about programming [Schmolitzky 2007]. According to authors of area [Bergin et al. 2012], some patterns may sound trivial to experienced educators, but they show up as useful references to less experienced ones, as well as a way to expand teaching skills. The succinct format used to present patterns ease its sharing; in other hand, the absence of complementary data about the evaluation of certain patterns can evoke questions about its effectiveness.

In this work, we present a pedagogical pattern to programming teaching in a higher course of Computation. This pattern–called HALL OF FAME & SHAME (HOFS)–is concerned on exposure and discussion of good and bad programming examples produced by the students. Its main objective is motivate the students to improve their programming skills. In addition to describe the pattern in succinct way, in a format that facilitates reuse, we also present a experimental evaluation of its application in a higher educational institution.

The text is organized as following. In the Section (1) we do a short historical review of pedagogical patterns for programming teaching, pointing the most relevant references about the subject nowadays; in Section (2) we present HoFS pattern in a common format used by community of the area; in Section (3) we describe the application and evaluation of the pattern on a higher educational institution; Lastly, we present our final considerations about this work in Section (4).

Pedagogical patterns [Sharp et al. 1996] emerged from discussions in object oriented teaching context and its technologies. The inspiration comes from so-called design patterns [Gamma et al. 1993], which consist of well established solutions for recurring problems in object oriented context. The initial idea of pedagogical patterns is, therefore, use a similar approach to catalog reusable solutions for common problems found on teaching of aforementioned paradigm.

Gradually, the idea of pedagogical pattern was extended to a larger scope, capturing solutions to problems found in different situations rather object orientation. Researchers and educators started to propose patterns in conferences, as PLoP (Pattern Languages of Programs) and EuroPLoP (European Conference on Pattern Language of Programs) and then, in more traditional conferences about computation teaching, such as SIGCSE (ACM Technical Symposium on Computing Science Education) and ITiCSE (ACM Conference on Innovation and Technology Science Education). The patterns were gathered in sites[1][2] and, more recently, several of them were presented in books [Bergin et al. 2012; Mor et al. 2014].

An important aspect of pedagogical patterns is its format of description, which is always succinct and organized to ease reuse. Although the format is not strict, it usually includes the following sections: pattern name, objective/motivation, applicability, structure, consequences, implementation, resources needed, examples and related patterns [Sharp et al. 1996]. Some author present the description in less sections, keeping more basic informations only [Schmolitzky 2007]. We choose to present the HOFS pattern within fewer sections to keep its presentation less fragmented. To ease communication, we use the pronoun “you” as a direct reference to teacher in Section (2).

2. THE HALL OF FAME & SHAME PATTERN (HOFS)

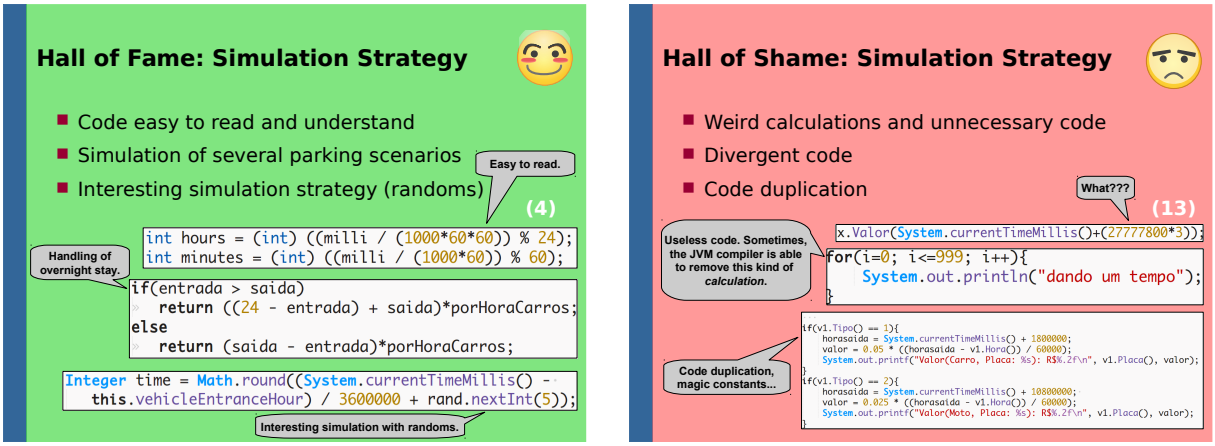
The proposed patterns is described as follows, using a similar format of presentation of [Bergin 2000].

- **Name:** HALL OF FAME & SHAME

[1] http://www.pedagogicalpatterns.org
[2] http://educationalpatterns.org

- **Problem:** The pattern is applicable to activities that involves design and decision making. In classroom, students might have difficulties to use or apply recently learned subjects to solve teacher assignments. In most of cases, they have not enough information to criticize their work-in-progress solutions, which can lead to adoption of 'trial-and-error' approaches. This is bad by itself, and such way of thinking mostly results in bad methodologies and habits. Such problems may happen again every time that a student face a new problem. The most common feedback procedures used by teachers are numbers (score), general comments, or standard solutions, which does not encourage students to think about how to improve their approaches of solutions development.
- **Context:** You are teaching a subject related to design or decision making, and your students have little skill with such topic. Suppose you are a teacher of programming class. For example, you may ask to your students to produce a software to solve a set of problems using a recently learned new programming technique, a new programming paradigm or a new programming language. Your objective is encourage the active thinking of students about how to solve such problems using the recently learned subject. You should be careful to craft the problem set to raise relevant questions about the study topic. Your classroom is not too big (it has 20 to 40 individuals). You want to encourage them to learn and adopt good problem solving practices, using you own expertise.
- **Solution:** After the students delivered their solutions to their assignments, do an analysis looking for good and bad solution practices. Desirably, allow students to deliver partial solutions before deadline set out to make them more active and involved with the activity [Larson et al. 2008; Köppe et al. 2015]. After each partial delivery, select some fragments of students solutions to compile an exposition separated in two "galleries", as shown in Figure 1: *Hall of Fame* (good practices) and *Hall of Shame* (bad practices). In classroom, show and comment about the selected solution snippets, asking for students to think and categorize the snippets as "Fame" or "Shame". You may use contrasting visual resources to identify both galleries. Note that, for the same problem, it is possible to find both good and bad solution approaches, which proposes that the gallery is not about good and bad programmers, but is about good and bad coding practices. You should also make the snippets anonymous. Whenever is possible, repeat this pattern. This approach stimulates the critical thinking of students, which is common habit between experienced programmers. The pattern provides feedback based in problems that the students dedicated themselves to solve. Instead of generic and abstract recommendations about problem solving methodologies, the pattern gives real examples from the own group of students. With a clear and reinforced distinction between Fame and Shame, it is expected a stimulus of cognitive processes, resulting in adoption of good practices in following assignments. It is not a objective of this pattern act as replacement to individual evaluation of programs (even in situations that it is not possible), but it is an option to promote quick and useful feedback to the group.
- **Side effects:** The pattern has its own limitations: (i) it demands a considerably amount of time to compile, expose and comment the galleries after each partial delivery, (ii) a homogeneity of experience of group tends to reduce diversity of solutions and (iii) the teacher should take care when referencing to snippets of solutions to not "attack" the students: making solution snippets anonymous could not be not enough since the author of snippet could recognize their own work. In this case, several occurrences of their solution in "shame" galleries can make the student lost his/her interest to follow up with the assignment.
- **Resource requirements:** It is recommended to expose the galleries with a multimedia projector, but any other multimedia resource could be employed to improve the discussion. Teacher may publish the material (in the course website, by e-mail, etc) for use outside of classroom. The problems proposed in the assignment should be specified to promote the variety of solutions. It could be useful some assistance from a monitor to prepare galleries and perform its exhibition–ideally from few days to up to a week after the delivery. It is a good idea to keep a repository of solution snippets of previous applications of pattern to enrich the discussions when students solutions are too much similar, for example.

- **Examples:** An example of application of HoFS is in teaching of procedural language C, after the students learned how to structure code in functions and procedures. At this point, it is common that the students have to implement a simple game or a text editor. The *Shame* gallery, then, will reveal code snippets with following problems: poor division of code into small subprograms, procedures that perform more or less instructions than the name suggests, procedures that access global variables instead of use parameters, bad code indentation, etc. In *Fame* gallery, there will be good code snippets, pointing to solutions that overcome bad solutions of *Shame*. Other example is teaching of functional programming with languages like Haskell and Lisp. Usually, it proposes to students to solve several kind of problems with lists, using recursive approaches and higher order functions. When students already known both alternatives, the *Shame* gallery could point, for example, code snippets that use explicit recursion instead of higher order functions as `map` and `filter` (since higher order functions enable use of lambda expression in several modern languages [Pierce 2002]).
- **Related patterns:** Application of HoFS supposes that students should produce code to implement some software, even an incomplete one. So, patterns related to teaching-learning can be used [Schmolitzky 2007]. In addition, the HoFS pattern is centered on the idea that students can benefit themselves of frequent feedback to advance and improve their programming skills. In this way, it can be complemented with other pedagogical patterns that focus in *continuum feedback* [Larson et al. 2008]. HoFS share some objectives of [Köppe et al. 2015] patterns to mitigate common issues in CS courses, like in Activating Delivery Forms and Discussion Statements. Other related patterns are Active Student, Set the Stage and Suitable Delivery Form Selection [Board 2012; Köppe et al. 2015]. HoFS also have several aspects in common with USE STUDENT'S SOLUTIONS [KÖPPE et al. 2015], since both patterns suggest the active discussion of snippets from students solutions. Additionally, HoFS explores the use of partial deliveries of assignments before its deadline to strenghen the engagement of students into the discussions.



(a) (b)

Fig. 1. Sample slides from the "Fame" (a) and "Shame" (b) galleries. Note the use of contrasting visual elements.

3. APPLICATION AND EVALUATION OF HOFS

We applied the proposed pattern to fourth semester students of a course which attend to Object Oriented Programming (OOP), offer by Federal University of Santa Maria, Brazil, to students of third and subsequent semesters of undergraduate course in Computer Science. The first contact of students with OOP–usually with Java

language–happens in this course. In the first application of HoFS, we perceive a good acceptance by students since: (i) students become involved during code exposition, paying more attention to teacher's talks and (ii) some students were concerned about their code been shown (or not) into some gallery.

Given these observations, we put the pattern in practice on the last course offer, following previously established procedures and criteria, aiming to collect data about patterns' impact on OOP learning. The remaining of this section refers to evaluation of application of HoFS.

We did not found any criteria or procedure to systematically evaluate pedagogical patterns in literature. Then, we decide to evaluate 2 aspects that, in our experience, are potential indicators about the efficiency (or not) of pattern. One of those is the students' **acceptance** of approach, measured by a survey that has been answered in course ending. The second is students' code **production**, which may contains good and/or bad programming practices, which was evaluated after each delivery. We understand that both aspects complement themselves. As an example of this complementation, a student can positively evaluate the experience, but the experience does not lead to any practical improvement in produced code.

3.1 Acceptance evaluation: methodology and results

To evaluate the acceptance of HoFS pattern, we prepare a 2 step survey, each one presented in form of items weighed in 5-point Likert scale, from -2 (totally disagree) to 2 (totally agree). The first step has 5 questions related to students' general perception about the pattern, relating it with their previous learning experiences. The second step has 4 questions about format adopted in exhibition and discussion of galleries, which comprises of interactive lectures using slides to expose code snippets, followed by on-line publication of a brief material for future reference (as presented in Section 2). Also, answers for second step of survey pointed some adjustments needed in pattern application.

The survey was anonymously answered by 15 students, in total of 22 students that attended the course until the end of semester. These students had practice and theoretical lectures, performed assignments and developed 2 individual projects with OOP. The HoFS patterns was applied in 3 times: (1) after delivery of first project, (2) in a partial delivery of second project (50% of requirements implemented) and (3) at deadline of second project. Students were evaluated by their performance to develop the software and by delivered report quality. The survey was applied at end of semester, after publication of scores.

We present the results of first step of survey in Figure (2). Note that one question of this part of survey has a negative proposition ("I did not liked to see my errors exposed, even anonymously"). In general, collected answers strengthen the initial hypothesis of good approach acceptance. Moreover, students perceived the originality of approach to disagree with "I already have lectures like this in other disciplines".

We present the results of second step of survey in Figure (3). In this regard, we observed a general acceptance of format, but some items need more attention: (1) Fame and Shame comments presented before assignment deadline were seen as more useful than those presented after deadline; (2) several students answered "Partially agree" to item "The published material for use outside of classroom was sufficient". The item (1) suggests that, when possible, the pattern should be applied while students are coding their solutions. The item (2) reveal a potential gap between effectiveness of galleries during and after lectures. The reasons of this can be investigated in new essays but, as a immediate alternative, we estimate that publication of videos about galleries' presentation can overcome those gap.

3.2 Production evaluation: methodology and results

To evaluate the impact of HoFS pattern in quality of students' productions, we considered 2 OOP projects assigned to students. These projects were developed in 1 and 3 weeks, respectively, and were delivered to teacher in two opportunities, as stated in previous section.

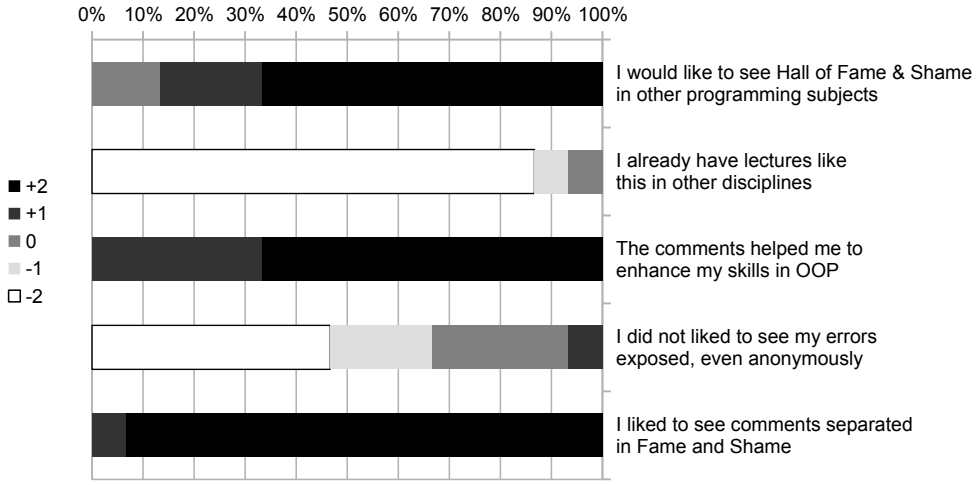


Fig. 2. Evaluation of general perception of pattern by students.

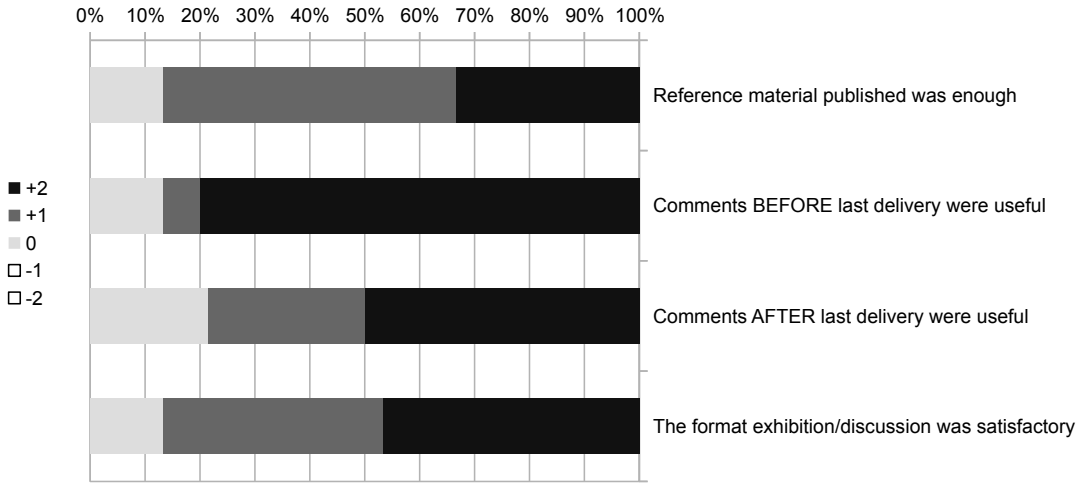


Fig. 3. Evaluation of format used no lectures following HoFS pattern.

The first project was a parking lot management software with a simulation of arrival and departure of vehicles. This project has no requirements about user interface, but emphasizes a object oriented implementation of domain in question. HoFS was applied after the deadline of this project.

The second project was about the development of a desktop graphical user interface, in Java, aiming to manage the records of fuel stations of a city with their respective pricing history. The project should use the MVC (Model-View-Controller) pattern and persist data in file or database. To this assignment, we applied HoFS pattern before and after project deadline.

For analysis of students' production, we consider all codes delivered by them on 3 previously mentioned moments, in a total of 17 and 20 projects delivered for the first and second assignments, respectively. For each

Fame and Shame exposition, we choose an average of 15 code snippets to present and discuss, all of them about good and bad practices of several aspects of OOP. The students could access the galleries in course web-page.

The subject of two projects allows use of optional good OOP approaches (as example, Data Access Objects for data persistence). By other side, some practices related to basement of OOP should be present in all works. In this situation, the hypothesis was that HoFS pattern can stimulate the students to correct their bad practices between a assignment to other. Thus, among all good and bad practices identified in projects, we choose 3 bad practices that their occurrences were counted before of first HoFS (before deadline of first project) and after of last HoFS (after deadline of last project). These practices were (1) code duplication rather than polymorphism, (2) violation of principle of single responsibility on class implementation and (3) public visibility of instance's attributes.

In Table (I), we present identified occurrences, being that a student can be responsible by more than one occurrence. We also can note that occurrences of bad practices decreased, as expected. It is well known that the nature of teaching-learning is very complex and, for this, we can not say undoubtedly that reduction of bad practices was only due to HoFS application. Nevertheless, we interpret these results as evidence about effectiveness of pattern. More important than this is the advances presented by the students and stimulated by HoFS.

Table I. Occurrences of bad and good coding practices after and before HoFS application.

Bad practice	Occurrences	
	Before	After
Code duplication	5	1
Single responsibility principle violations	10	2
Public visibility of instances attributes	3	0

4. FINAL REMARKS

In this work, we characterized a pedagogical pattern focused in programming teaching. This pattern is related to others of continuous feedback emphasis, but it differs since it is based essentially on analysis and exhibition of code snippets that represents good and bad programming practices.

The pattern was applied in a higher education institution and, on its last application, we performed an evaluation based in students acceptance of pattern and into analysis of code produced by themselves. The results show to efficiency and originality of pattern. Aiming to ease the reproduction of pattern in other time, galleries were published on-line and are available in `http://www-usr.inf.ufsm.br/~andrea/hofs-plop2016`.

We know that, a single pedagogical pattern can not guarantee the success of teaching-learning process. However, pedagogical patterns are an useful resource to teachers that want to improve their practices, as more patterns were documented, applied and discussed. Thus, as future prospects related to this work, we expect that more teachers share original patterns and/or report their application experiences of pedagogical patterns in Computation.

Lastly, the main idea behind some pedagogical patterns are generic and useful in several knowledge areas. As an example, the core of HoFS pattern is the analysis of trade-offs related to decision making, which can be extended to any design process. As a future work, the code snippets repository of previous applications of HoFS might be published together to the *fame* and *shame* galleries, extending our original approach to a "system of patterns".

Acknowledgement

The second author would like to thank CAPES foundation for his masters scholarship.

We would also to thank our shepherd Hironori Washizaki for his valuable comments. We are grateful for his patience during our (several) interactions and we have no doubt that his observations brought a lot of improvement to our work.

REFERENCES

Joseph Bergin. 2000. Fourteen Pedagogical Patterns. In *EuroPLoP*. 1–49.

Joseph Bergin, Jutta Eckstein, Markus Volter, Marianna Sipos, Eugene Wallingford, Klaus Marquardt, Jane Chandler, Helen Sharp, and Mary Lynn Manns. 2012. *Pedagogical patterns: advice for educators*. Joseph Bergin Software Tools.

Pedagogical Patterns Advisory Board. 2012. Pedagogical Patterns: Advice for Educators. *Joseph Bergin Software Tools* (2012).

Michael E Caspersen and Michael Kolling. 2009. STREAM: A first programming process. *ACM Transactions on Computing Education (TOCE)* 9, 1 (2009), 4.

Peter J. Denning. 1989. A Debate on Teaching Computing Science. *Commun. ACM* 32, 12 (Dec. 1989), 1397–1414. DOI:http://dx.doi.org/10.1145/76380.76381

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. *Design patterns: Abstraction and reuse of object-oriented design*. Springer.

Tony Jenkins. 2002. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, Vol. 4. 53–58.

Christian Köppe. 2015. Towards a Pattern Language for Lecture Design: An inventory and categorization of existing lecture-relevant patterns. In *Proceedings of the 18th European Conference on Pattern Languages of Program*. ACM, 3.

CHRISTIAN KÖPPE, RALPH NIELS, ROBERT HOLWERDA, LARS TIJSMA, NIEK VAN DIEPEN, K Van Turnhout, and R Bakker. 2015. Flipped Classroom Patterns-Using Student Solutions. In *Preprints of the 22nd Pattern Languages of Programs conference, PLoP*, Vol. 15.

Christian Köppe, Michel Portier, René Bakker, and Stijn Hoppenbrouwers. 2015. Lecture Design Patterns: More Interactivity Improvement Patterns. In *Preprints of the 22nd Pattern Languages of Programs conference, PLoP*, Vol. 15.

Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. In *ACM SIGCSE Bulletin*, Vol. 37. ACM, 14–18.

Kathleen A Larson, Frances P Trees, and D Scott Weaver. 2008. Continuous feedback pedagogical patterns. In *Proceedings of the 15th Conference on Pattern Languages of Programs*. ACM, 12.

Yishay Mor, Harvey Mellar, Steven Warburton, and Niall Winters. 2014. *Practical design patterns for teaching and learning with technology*. Springer.

Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin* 39, 4 (2007), 204–223.

Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

Anthony Robins. 2010. Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education* 20, 1 (2010), 37–71.

Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172.

Axel Schmolitzky. 2007. Patterns for Teaching Software in Classroom.. In *EuroPLoP*. 37–52.

Helen Sharp, Mary Lynn Manns, Phil McLaughlin, Maximo Prieto, and Mahesh Dodani. 1996. Pedagogical patterns—successes in teaching object technology: a workshop from OOPSLA'96. *ACM SIGPLAN Notices* 31, 12 (1996), 18–21.

Elliot Soloway and James C Spohrer. 2013. *Studying the novice programmer*. Psychology Press.

Phit-Huan Tan, Choo-Yee Ting, and Siew-Woei Ling. 2009. Learning difficulties in programming courses: undergraduates' perspective and perception. In *Computer Technology and Development, 2009. ICCTD'09. International Conference on*, Vol. 1. IEEE, 42–46.

Brenda Cantwell Wilson and Sharon Shrock. 2001. Contributing to success in an introductory computer science course: a study of twelve factors. In *ACM SIGCSE Bulletin*, Vol. 33. ACM, 184–188.

Leon E Winslow. 1996. Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin* 28, 3 (1996), 17–22.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers, Principles, Techniques**. [S.l.]: Addison wesley Boston, 1986.

BEAUBOUEF, T.; MASON, J. Why the high attrition rate for computer science students: some thoughts and observations. **ACM SIGCSE Bulletin**, ACM, v. 37, n. 2, p. 103–106, 2005.

BECK, K. **Test-driven development: by example**. [S.l.]: Addison-Wesley Professional, 2003.

BEIZER, B. **Black-box testing: techniques for functional testing of software and systems**. [S.l.]: John Wiley & Sons, Inc., 1995.

BERGIN, J. et al. **Pedagogical Patterns: Advice For Educators**. Joseph Bergin Software Tools, 2012. Disponível em: <http://oro.open.ac.uk/34138/>.

BRAY, T. et al. Extensible markup language (xml). **World Wide Web Journal**, v. 2, n. 4, p. 27–66, 1997.

BROWN, M. H. Algorithm animation. Brown University, 1987.

BRUSILOVSKY, P. et al. Mini-languages: a way to learn programming principles. **Education and Information Technologies**, Springer, v. 2, n. 1, p. 65–83, 1997.

CAMPOS, C. P. de; FERREIRA, C. E. Boca: um sistema de apoio a competições de programação. In: **Workshop de Educação em Computação**. [S.l.: s.n.], 2004. p. 885–895.

CASPERSEN, M. E.; KOLLING, M. Stream: A first programming process. **Trans. Comput. Educ.**, ACM, New York, NY, USA, v. 9, n. 1, p. 4:1–4:29, mar. 2009. ISSN 1946-6226. Disponível em: <http://doi.acm.org/10.1145/1513593.1513597>.

CHACON, S.; STRAUB, B. **Pro git**. [S.l.]: Apress, 2014.

CHARÃO, A. S. et al. Hall of fame/shame: um padrão pedagógico para o ensino de programação. In: SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (Ed.). **Anais do XXXVI Congresso da Sociedade Brasileira de Computação**. Porto Alegre, Rio Grande do Sul, 2016. p. 2166–2175.

CHARãO, A. S. et al. Hall of fame/shame: um padrão pedagógico para o ensino de programação. In: SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (Ed.). **Anais do XXXVI Congresso da Sociedade Brasileira de Computação**. Porto Alegre, Rio Grande do Sul, 2016. p. 2166–2175.

CLIFTON, C.; KACZMARCZYK, L. C.; MROZEK, M. Subverting the fundamentals sequence: using version control to enhance course management. In: ACM. **ACM SIGCSE Bulletin**. [S.l.], 2007. v. 39, n. 1, p. 86–90.

CROCKFORD, D. The application/json media type for javascript object notation (json). 2006.

DALE, N. B. Most difficult topics in cs1: results of an online survey of educators. **ACM SIGCSE Bulletin**, ACM, v. 38, n. 2, p. 49–53, 2006.

DARWIN, I. F. **Checking C Programs with lint**. [S.l.]: "O'Reilly Media, Inc.", 1991.

DELINE, R.; ROWAN, K. Code canvas: zooming towards better development environments. In: ACM. **Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2**. [S.l.], 2010. p. 207–210.

DENNING, P. J. A debate on teaching computing science. **Commun. ACM**, ACM, New York, NY, USA, v. 32, n. 12, p. 1397–1414, dez. 1989. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/76380.76381>.

EGAN, M. H.; MCDONALD, C. Program visualization and explanation for novice c programmers. In: AUSTRALIAN COMPUTER SOCIETY, INC. **Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148**. [S.l.], 2014. p. 51–57.

EMDEN, E. V.; MOONEN, L. Java quality assurance by detecting code smells. In: IEEE. **Reverse Engineering, 2002. Proceedings. Ninth Working Conference on**. [S.l.], 2002. p. 97–106.

ERLIKH, L. Leveraging legacy system dollars for e-business. **IT professional**, IEEE, v. 2, n. 3, p. 17–23, 2000.

FINCHER, S.; UTTING, I. Pedagogical patterns: their place in the genre. In: ACM. **ACM SIGCSE Bulletin**. [S.l.], 2002. v. 34, n. 3, p. 199–202.

FOWLER, M.; BECK, K.; BRANT, J. Refactoring: improving the design of existing code. **Refactoring: Improving the Design of Existing Code**, Addison-Wesley, 1999.

GITHUB. **Github Pages, websites for you and your projects**. 2017. Disponível em <https://pages.github.com/>.

GOOGLE TRENDS. **git, Apache Subversion, Mercurial, Perforce Helix, Concurrent Versions System**. 2017. Disponível em: <https://goo.gl/IFr0X5>.

HAARANEN, L.; LEHTINEN, T. Teaching git on the side: Version control system as a course platform. In: ACM. **Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education**. [S.l.], 2015. p. 87–92.

HARDT, D. The oauth 2.0 authorization framework. 2012.

HENSON, V.; GARZIK, J. Bitkeeper for kernel developers. In: **Ottawa Linux Symposium**. [S.l.: s.n.], 2002. p. 197–212.

JENKINS, T. On the difficulty of learning to program. In: **Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences**. [S.l.: s.n.], 2002. p. 53–58.

JOHNSON, S. C. Lint, a c program checker. In: **COMP. SCI. TECH. REP**. [S.l.: s.n.], 1978. p. 78–1273.

KELLEHER, J. Employing git in the classroom. In: IEEE. **Computer Applications and Information Systems (WCCAIS), 2014 World Congress on**. [S.l.], 2014. p. 1–4.

KERNIGHAN, B. W.; RITCHIE, D. M. **The C programming language**. [S.l.: s.n.], 2006.

KO, A. J. et al. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. **IEEE Transactions on software engineering**, IEEE, v. 32, n. 12, 2006.

KÖPPE, C. et al. Lecture design patterns: More interactivity improvement patterns. In: **Preprints of the 22nd Pattern Languages of Programs conference, PLoP**. [S.l.: s.n.], 2015. v. 15.

KRAMER, D. Api documentation from source code comments: a case study of javadoc. In: ACM. **Proceedings of the 17th annual international conference on Computer documentation**. [S.l.], 1999. p. 147–153.

LAADAN, O.; NIEH, J.; VIENNOT, N. Teaching operating systems using virtual appliances and distributed version control. In: ACM. **Proceedings of the 41st ACM technical symposium on Computer science education**. [S.l.], 2010. p. 480–484.

LAHTINEN, E.; ALA-MUTKA, K.; JÄRVINEN, H.-M. A study of the difficulties of novice programmers. In: ACM. **ACM SIGCSE Bulletin**. [S.l.], 2005. v. 37, n. 3, p. 14–18.

LANZA, M.; MARINESCU, R. **Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems**. [S.l.]: Springer Science & Business Media, 2007.

LARSON, K. A.; TREES, F. P.; WEAVER, D. S. Continuous feedback pedagogical patterns. In: **Proceedings of the 15th Conference on Pattern Languages of Programs**. New York, NY, USA: ACM, 2008. (PLoP '08), p. 12:1–12:14. ISBN 978-1-60558-151-4. Disponível em: <http://doi.acm.org/10.1145/1753196.1753211>.

LAWRANCE, J.; JUNG, S.; WISEMAN, C. Git on the cloud in the classroom. In: ACM. **Proceeding of the 44th ACM technical symposium on Computer science education**. [S.l.], 2013. p. 639–644.

LEE, J. W.; KESTER, M. S.; SCHULZRINNE, H. Follow the river and you will find the c. In: ACM. **Proceedings of the 42nd ACM technical symposium on Computer science education**. [S.l.], 2011. p. 411–416.

LINUX.COM. **10 Years of Git: An Interview with Git Creator Linus Torvalds**. 2017. Disponível em: <https://goo.gl/9O6242>.

MARJAMÄKI, D. Cppcheck: a tool for static c/c++ code analysis. **available at:< a href="http://cppcheck. sourceforge. net/» http://cppcheck. sourceforge. net/(last access: 30 May 2014)**, 2013.

MARTINS, L. C.; LOPES, D. A.; RAABE, A. Um assistente de predição de evasão aplicado a uma disciplina introdutória do curso de ciência da computação. **Anais do** 23^o **Simpósio Brasileiro de Informática na Educação**, nov 2012.

MCCONNELL, S. **Code Complete**. Microsoft Press, 2009. (DV-Professional). ISBN 9780735636972. Disponível em: <https://books.google.com.br/books?id=3JfE7TGUwvgC>.

MORENO, A. et al. Visualizing programs with jeliot 3. In: ACM. **Proceedings of the working conference on Advanced visual interfaces**. [S.l.], 2004. p. 373–376.

MYERS, B. A. Visual programming, programming by example, and program visualization: a taxonomy. In: ACM. **ACM SIGCHI Bulletin**. [S.l.], 1986. v. 17, n. 4, p. 59–66.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011.

NETHERCOTE, N.; SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM. **ACM Sigplan notices**. [S.l.], 2007. v. 42, n. 6, p. 89–100.

NETO, V. dos S. M. A utilização da ferramenta scratch como auxílio na aprendizagem de lógica de programação. **II Congresso Brasileiro de Informática na Educação**, 2013).

NGUYEN, A. et al. Codewebs: scalable homework search for massive open online programming courses. In: ACM. **Proceedings of the 23rd international conference on World wide web**. [S.l.], 2014. p. 491–502.

OLBRICH, S. et al. The evolution and impact of code smells: A case study of two open source systems. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement**. [S.l.], 2009. p. 390–400.

PEARS, A. et al. A survey of literature on the teaching of introductory programming. In: **Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2007. (ITiCSE-WGR '07), p. 204–223. Disponível em: <http://doi.acm.org/10.1145/1345443.1345441>.

PIERCE, B. C. **Types and programming languages**. [S.l.]: MIT press, 2002.

PILATO, C. M.; COLLINS-SUSSMAN, B.; FITZPATRICK, B. **Version control with subversion**. [S.l.]: "O'Reilly Media, Inc.", 2008.

PLUMLEE, M. D.; WARE, C. Zooming versus multiple window interfaces: Cognitive costs of visual comparisons. **ACM Transactions on Computer-Human Interaction (TOCHI)**, ACM, v. 13, n. 2, p. 179–209, 2006.

PRECHELT, L.; MALPOHL, G.; PHILIPPSEN, M. Finding plagiarisms among a set of programs with jplag. **J. UCS**, v. 8, n. 11, p. 1016, 2002.

RAABE, A. L. A. Adquirindo experiência na construção de ferramentas de apoio a aprendizagem de algoritmos. In: **Workshop de Ambientes de Apoio a Aprendizagem de Algoritmos e Programação. Simpósio Brasileiro de Informática na Educação**. [S.l.: s.n.], 2007.

ROBERTS, D.; BRANT, J.; JOHNSON, R. A refactoring tool for smalltalk. **Urbana**, v. 51, p. 61801, 1997.

ROCHKIND, M. J. The source code control system. **IEEE Transactions on Software Engineering**, IEEE, n. 4, p. 364–370, 1975.

ROSSUM, G. V.; DRAKE, F. L. **Python library reference**. [S.l.]: Centrum voor Wiskunde en Informatica, 1995.

ROSSUM, G. V. et al. Python programming language. In: **USENIX Annual Technical Conference**. [S.l.: s.n.], 2007. v. 41, p. 36.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: local algorithms for document fingerprinting. In: ACM. **Proceedings of the 2003 ACM SIGMOD international conference on Management of data**. [S.l.], 2003. p. 76–85.

SCHMOLITZKY, A. Patterns for teaching software in classroom. In: **Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007)**. Irsee, Germany: Hillside Europe, 2007. (EuroPLoP '07), p. B5:1–B5:10. Disponível em: <http://doi.acm.org/10.1145/2739011.2739014>.

SHARP, H.; MANNS, M. L.; ECKSTEIN, J. Evolving pedagogical patterns: The work of the pedagogical patterns project. **Computer Science Education**, Taylor & Francis, v. 13, n. 4, p. 315–330, 2003.

SHARP, H. et al. Pedagogical patterns – successes in teaching object technology: A workshop from OOPSLA '96. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 31, n. 12, p. 18–21, dez. 1996. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/242604.242610>.

SHEARD, J. et al. Analysis of research into the teaching and learning of programming. In: ACM. **Proceedings of the fifth international workshop on Computing education research workshop**. [S.l.], 2009. p. 93–104.

SOLOWAY, E.; SPOHRER, J. C. **Studying the Novice Programmer**. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1988. ISBN 0805800026.

SONARSOURCE. **SonarLint: an extension to your favorite IDE that provides on-the-fly feedback to developers on new bugs and quality issues injected into their code**. 2017. Disponível em: <http://www.sonarlint.org/>.

SOUZA, C. M. de. Visualg-ferramenta de apoio ao ensino de programação. **Revista Eletrônica TECCEN**, v. 2, n. 2, 2016.

STALLMAN, R. M. et al. **Using GCC: the GNU compiler collection reference manual**. [S.l.]: Gnu Press, 2003.

TAN, P. H.; TING, C. Y.; LING, S. W. Learning difficulties in programming courses: Undergraduates' perspective and perception. In: **Computer Technology and Development, 2009. ICCTD '09. International Conference on**. [S.l.: s.n.], 2009. v. 1, p. 42–46.

THENAULT, S. **Code analysis for Python.** 2017. Disponível em <pylint.org>.

TICHY, W. F. Rcs—a system for version control. **Software: Practice and Experience**, Wiley Online Library, v. 15, n. 7, p. 637–654, 1985.

WALDRON, R. et al. **JSHint, A Static Code Analysis Tool for JavaScript**. 2016. Disponível em: <https://github.com/jshint/jshint>.

WEXELBLAT, R. L. **History of programming languages**. [S.l.]: Academic Press, 2014.

WILSON, B. C.; SHROCK, S. Contributing to success in an introductory computer science course: A study of twelve factors. In: **Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2001. (SIGCSE '01), p. 184–188. ISBN 1-58113-329-4. Disponível em: <http://doi.acm.org/10.1145/364447.364581>.

WINSLOW, L. E. Programming pedagogy – a psychological overview. **SIGCSE Bulletin**, ACM, New York, NY, USA, v. 28, n. 3, p. 17–22, set. 1996. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/234867.234872>.