Minerando Código Comentado

Lucas Grijó[1], **Andre Hora**[1]

[1] Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)

rksgrijo@gmail.com, hora@facom.ufms.br

Abstract. *As software evolves, programmers often comment code snippets as a mean of removing them (a practice known as comment-out code). However, this bad practice may cause problems to software maintainers, such as readability reduction, distraction, and waste of time. In this context, this paper presents an empirical study to evaluate the presence of commented-out code on open source systems. We analyze 100 relevant software systems and 300 releases. As a result, we detect a rate of 4,17% commented-out code. However, we verify that this rate tends to decrease over time. Finally, based on our results, we discuss improvements to software quality tools and development practices.*

Resumo. *Durante a evolução de software, programadores comumente comentam trechos de código como forma de remoção (prática conhecida como comment-out code). Entretanto, essa má prática pode causar problemas para mantenedores de software, como redução de legibilidade, distração e perda de tempo. Nesse contexto, este artigo apresenta um estudo empírico para avaliar a presença de códigos comentado em sistemas open source. Analisa-se 100 sistemas de software relevantes e 300 releases desses projetos. Como resultado, detecta-se que 4,17% dos comentários são compostos por códigos comentado. No entanto, verifica-se que, na mediana, essa taxa tende a diminuir ao longo do tempo. Por fim, com base nos nossos resultados, são discutidas melhorias para ferramentas de qualidade de software e práticas de desenvolvimento.*

1. Introdução

Sistemas de software estão em constante evolução para acomodar necessidades de negócio. Durante o desenvolvimento, programadores utilizam diversos artifícios para lidar com a evolução de software. Uma prática comum quando programadores detectam trechos de código que precisam ser deletados é "removê-los" através de comentários de código. Essa má prática de programação, que deriva do termo em inglês *comment-out code*, se refere ao ato de comentar um trecho de código ao invés de simplesmente removê-lo [Martin 2009]. Por exemplo, o código apresentado na Figura 1 contém trechos funcionais intercalados com trechos comentados.[1]

A adição de código comentado pode causar diversos problemas para mantenedores de software, tais como redução de legibilidade, distração e perda de tempo [Letouzey 2012, Letouzey and Coq 2010, Martin 2009]. De forma geral, essa

[1] Exemplo extraído de https://westminstercollege.edu

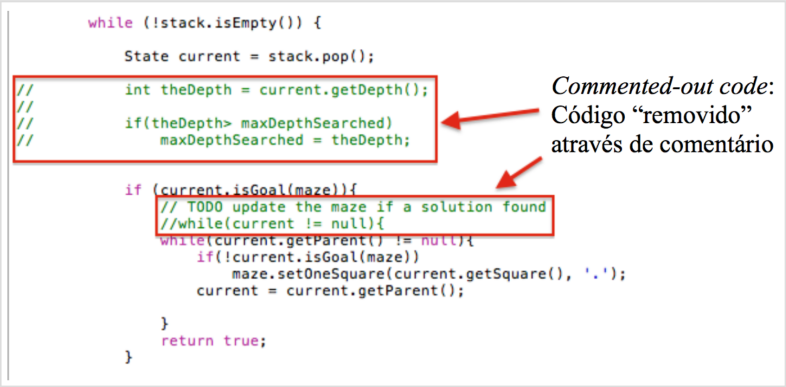

Figure 1. Exemplo de código "removido" através de comentário.

prática é muito discutida por profissionais[2], mas, no melhor do nosso conhecimento, ainda não foi devidamente investigada por pesquisadores.

Este artigo apresenta um estudo empírico para avaliar a presença de código comentado em sistemas *open source*. Especificamente, foca-se em responder duas questões de pesquisa: (QP1) Qual a taxa de código comentado atualmente? e (QP2) Como a taxa de código comentado evolui ao longo do tempo? Para responder as questões de pesquisa, analisa-se 300 *releases* fornecidas por 100 sistemas de software hospedados no GitHub. Logo, as principais contribuições desse trabalho são: (1) uma análise em larga escala sobre a presença da má prática de programação código comentado; (2) uma avaliação da ocorrência de código comentado ao longo do tempo; e (3) um conjunto de lições aprendidas através da análise de sistemas *open source*.

2. Por que Comentar Trechos de Código é uma Má Prática?

Diversos problemas podem surgir a partir da utilização de código comentado como forma de remoção. Em seu livro clássico e influente sobre técnicas para escrever código limpo, Clean Code [Martin 2009], Robert Martin sugere: "*Few practices are as odious as commenting-out code. Don't do this!*". O autor argumenta que outros mantenedores do código nunca terão a coragem para deletar o código comentado, decaindo a qualidade do software. A empresa de qualidade de software SonarQube[3] também aponta algumas desvantagens da inserção de código comentado:

1. Código comentado é uma distração e pode confundir o programador;
2. Código comentado sempre levanta mais questões do que respostas;
3. Programadores esquecerão rapidamente o quão relevante é o código comentado;
4. Código comentado é uma forma errada de controle de versão, uma vez que ferramentas como Git e SVN são indiscutivelmente a solução apropriada;
5. O simples fato do programador buscar entender a razão por trás do código comentado pode tomar muito tempo.

[2]Por exemplo: https://softwareengineering.stackexchange.com/questions/45378/is-commented-out-code-really-always-bad, http://www.markhneedham.com/blog/2009/01/17/the-danger-of-commenting-out-code, https://blog.sonarsource.com/commented-out-code-eradication-with-sonar

[3]https://www.sonarqube.org

Além desses problemas, é importante notar que o código comentado torna-se defasado ao longo do tempo uma vez que o sistema continua a evoluir normalmente enquanto o trecho de código comentado permanece estático. Por exemplo, código comentado pode referenciar classes e variáveis que foram deletadas ou invocar métodos já renomeados. Portanto, a simples remoção do comentário do código comentado não necessariamente garante o seu correto funcionamento nem sua compilação.

3. Metodologia

3.1. Seleção de Sistemas

Para responder as questões de pesquisa, analisa-se um conjunto de sistemas *open source* relevantes e populares armazenados no repositório GitHub e escritos na linguagem Java. Especificamente, selecionou-se os 100 primeiros projetos Java (ordenados pelo número de estrelas[4]) que atendem aos seguintes critérios: (i) possuir 3 ou mais releases e (ii) ser um sistema real. O primeiro critério foi adotado para realização de análise evolucionária. O segundo critério foi adotado pois o GitHub possui diversos projetos populares que não são sistemas reais, mas sim tutoriais, exemplos de códigos, guias, etc. Para tal, selecionou-se os projetos com página no GitHub em inglês ou português, e identificou-se manualmente se o projeto é um sistema real, como bibliotecas, frameworks, aplicativos móveis, etc.

A Figura 2 apresenta a distribuição do número de estrelas, releases, contribuidores e commits dos 100 sistemas selecionados. As medianas desses indicadores são: 7.450 estrelas, 31,5 releases, 57,5 contribuidores e 1.366 commits, comprovando a popularidade e relevância dos mesmos. Entre esses sistemas, destacam-se: ReactiveX/RxJava, elastic/elasticsearch, square/retrofit, square/okhttp e spring-projects/string-framework, todos com mais de 20.000 estrelas e pelo menos 40 releases.

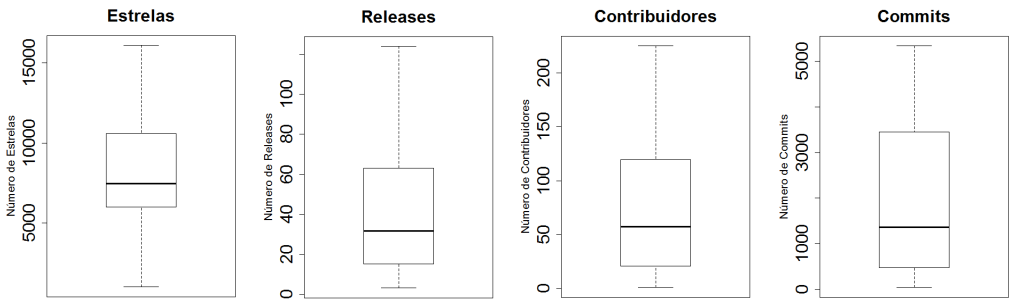

Figure 2. Caracterização dos sistemas analisados.

3.2. Extração de Código Comentado (Commented-out Code)

A abordagem proposta possui duas etapas principais: (1) extração de comentários e (2) detecção de código comentado:

1. Extração de comentários: primeiramente, extraiu-se os comentários dos arquivos Java presentes nos projetos selecionados;
2. Detecção de código comentado: em seguida, foram analisados os comentários para identificar ocorrências de código comentado.

[4]Métrica de popularidade nos projetos GitHub, similar ao *like*.

Extração de Comentários. Apesar de existirem algumas ferramentas capazes de extrair comentários em Java, tais como a Eclipse Java Development Tools (JDT) e outras similares que utilizam bibliotecas de expressões regulares (como a java.util.regex)[5], nenhuma ferramenta encontrada satisfez as nossas necessidades com relação a detecção de todos os tipos de comentários de forma eficiente. A JDT captura apenas um dos três tipos de comentários em Java (ie, Javadoc). Com outras ferramentas, a leitura de comentários suficientemente grandes resultava em um estouro no buffer, devido ao uso excessivo de recursões. Portanto, desenvolveu-se uma ferramenta com base na biblioteca de expressões regulares robusta *pcregrep*[6], para capturar os três tipos de comentários em Java:

- *Comentários multi-linha:* Comentários de uma ou mais linhas. Se inicia com `/*` e termina com `*/`. Exemplo: `/* this is a multi-line comment*/`
- *Comentários Javadoc:* Facilita a criação de documentação automática. Pode ser utilizado em uma ou mais linhas. Se inicia com `/**` e termina com `*/`. Exemplo: `/** this is a Javadoc comment */`
- *Comentários de linha única:* Comentários que terminam no final da linha e são iniciados com `//`. Exemplo: `//this is line comment`

Um desafio ao utilizar expressões regulares para encontrar padrões em códigos-fonte é que strings podem conter qualquer combinação de caracteres. Logo, é possível que uma string contenha um comentário dentro de seus delimitadores. Para evitar esses falsos positivos, tal cenário foi também tratado em nossa implementação.

Acurácia. A detecção de comentários depende da corretude da expressão regular utilizada e sua capacidade de capturar os tipos de comentários. Realizou-se centenas de testes com o auxílio da plataforma web regex 101[7], levando em consideração os mais variados cenários típicos e atípicos, em especial (i) os casos esperados encontrados em dezenas de sistemas reais e (ii) os casos excepcionais listados em *Finding Comments in Source Code Using Regular Expressions*.[8] Desse modo, em nossa avaliação, a ferramenta desenvolvida para detectar comentários Java obteve 100% de precisão e 100% de revocação.

Detecção de Código Comentado. Uma segunda ferramenta foi desenvolvida para a detecção de código comentado. Essa solução utiliza a biblioteca c2nes/javalang[9], que inclui um parser para códigos em Java capaz de tokenizar e reconhecer trechos de códigos bem formados. Nessa etapa, os conteúdos dos comentários extraídos no passo anterior são analisados. Durante a tokenização dos códigos dentro dos comentários, pode-se obter sucesso ou falha. As tokenizações que são bem sucedidas têm a possibilidade de serem códigos válidos, pois possuem uma sintaxe compatível com a linguagem Java (por exemplo: `//private int[] ranks;`). Os trechos que falham durante a tokenização não são códigos válidos (exemplo: `/* this is not a valid code */`).

Diversos casos particulares foram detectados e tratados para evitar falsos positivos. Abaixo são apresentados outros exemplos de códigos comentados detectados:

```
// throw new IllegalArgumentException();
// i = (args.length == 0) ?  1 :  Integer.parseInt(args[0]);
```

[5]https://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html
[6]https://www.pcre.org/original/doc/html/pcregrep.html
[7]https://regex101.com
[8]https://blog.ostermiller.org/find-comment
[9]https://github.com/c2nes/javalang

Acurácia. Para analisar a precisão e a revocação da ferramenta desenvolvida, selecionou-se aleatoriamente 2 mil comentários dentre 20 dos 100 sistemas estudados (100 comentários por sistema). Em seguida, analisou-se manualmente esses comentários: 107 continham código comentado e 1.893 continham documentação convencional. Ao analisar essa base com a ferramenta proposta, foi detectado 98 *true positives* (ie, classificado como código comentado e correto), 19 *false positives* (ie, classificado como código comentado e incorreto) e 9 *false negatives* (ie, não classificado como código comentado e incorreto). Desse modo, pode-se computar acurácia da detecção de código comentado: Precisão = TP/(TP+FP) = 98/117 = 83,8%; Revocação = TP/(TP+FN) = 98/107 = 91,6%. Desse modo, a precisão acima de 83% e a revocação acima de 91% mostra que a ferramenta desenvolvida pode ser utilizada com bom nível de confiança.

4. Resultados

QP1 - Qual a taxa de código comentado?

A taxa de código comentado foi calculada a partir da análise das últimas releases dos 100 sistemas selecionados. A Figura 3 (esquerda) apresenta a distribuição do número total de comentários e de código comentado. Na mediana, detectou-se 1.316 comentários por sistema (1o quartil 413 e 3o quartil 5.890). Dentre esses, verificou-se, na mediana, 51 códigos comentados por sistema (1o quartil 14 e 3o quartil 236). A Figura 3 (direita) apresenta a taxa de código comentado por sistema (razão "código comentado" por "total de comentários"): na mediana, 4.17% dos comentários são compostos por códigos comentado (1o quartil 1.93% e 3o quartil 8.31%).

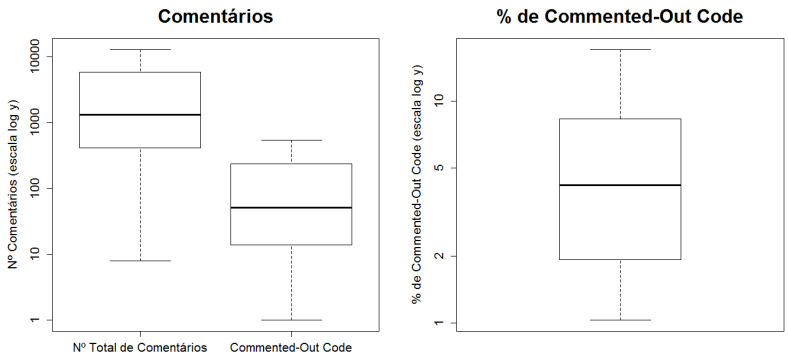

Figure 3. Taxa de código comentado.

Dentre os 100 sistemas, 9 não continham uma única ocorrência de código comentado. Por outro lado, alguns possuem uma alta taxa de código comentado; os 5 sistemas com a maior taxa são: cymcsg/UltimateRecyclerView (28,78%), navasmdc/MaterialDesignLibrary (23,89%), Blankj/AndroidUtilCode (19,93%), libgdx/libgdx (19,19%) e alibaba/fastjson (19,04%).

4.1. QP2 - Como a taxa de código comentado evolui ao longo do tempo?

Para analisar a taxa de código comentado ao longo do tempo, foram coletadas 3 releases de cada sistema: a inicial, uma intermediária e a final (totalizando 300 versões). Especificamente, as releases foram coletadas com suporte do comando `git tag` (que apresenta

a lista de releases de um dado projeto), dos quais foram selecionadas a primeira (mais antiga), a intermediária (metade do total de releases) e a última (mais recente). Após a seleção das releases, o comando `git checkout` foi utilizado para baixar a versão correspondente, que, por sua vez, teve seus comentários extraídos e analisados.

A Figura 4 apresenta a distribuição do número total de comentários e de código comentado assim como a taxa de código comentado nas três releases analisadas. As medianas do total de comentários são: 451, 1.064 e 1.316, para as releases iniciais, intermediárias e finais, respectivamente. As medianas de códigos comentados são: 28, 47 e 51. Por fim, as medianas das taxas de códigos comentados são: 6,57%, 4,52% e 4,17%.

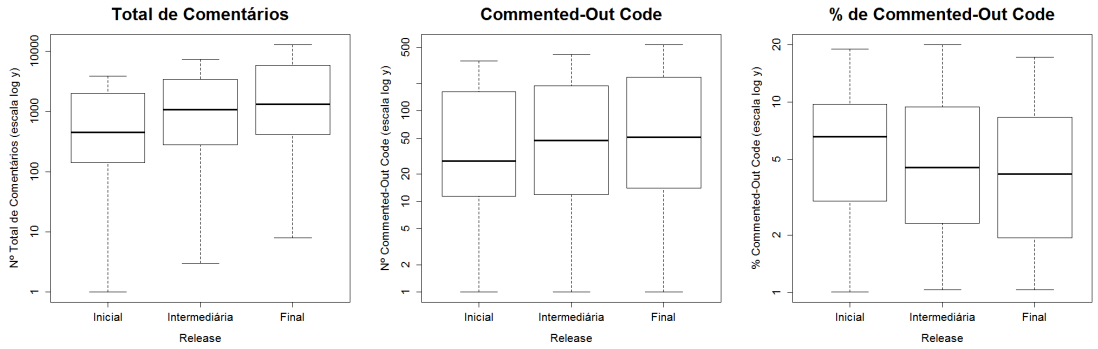

Figure 4. Evolução da taxa de código comentado.

Observa-se algumas tendências no decorrer dos projetos de software analisados. À medida que a quantidade total de comentários aumenta, a quantidade de código comentado code também aumenta; esse resultado é esperado, pois os sistemas tendem a crescer no decorrer do tempo. Entretanto, na mediana, a taxa de código comentado tende a diminuir ao longo do tempo. A mediana da taxa no início dos projetos é de 6,57%, decaindo para 4,17% em suas últimas versões.

Dentre os 100 sistemas analisados, 45 diminuíram suas taxas de código comentado, 46 aumentaram e 9 permaneceram estáveis. Os três sistemas com maior diminuição de código comentado são: iBotPeaches/Apktool (-33,71%), perwendel/spark (-27,95%) e SeleniumHQ/selenium (-24,08%). Por outro lado, os três sistemas com maior aumento de código comentado são: navasmdc/MaterialDesignLibrary (+14,88%), afollestad/material-dialogs (+13,35%) e wasabeef/recyclerview-animators (+13,33%).

5. Discussão

A taxa de código comentado é baixa, mas não deve ser ignorada. 91% dos sistemas analisados possuem código comentado. Portanto, somente 9% dos sistemas estão livres dessa má prática. Na mediana, a taxa de código comentado é de 4,17%. Porém, em alguns casos, essa porcentagem sobe para quase 30%. Logo, é crucial contar com apoio ferramental para identificar e eliminar essa prática. Entretanto, dentre as ferramentas de análise de qualidade do código mais populares do mercado, somente a SonarQube detecta trechos de código comentados. Outras ferramentas, como JArchitect, PMD e Checkstyle, não diferenciam comentários convencionais de código comentado. Desse modo, existe uma escassez de ferramentas para detectar essa má prática.

A taxa de código comentado tende a diminuir ao longo do tempo. De forma geral, a taxa de código comentado tende a diminuir ao longo do tempo, de 6,57% para 4,17%; um decréscimo de 36,53%. Isso indica que as equipes de desenvolvimento zelam pela manutenibilidade do código, particularmente evitando código comentado. Para manter qualidade, portanto, é fundamental que continuem vigilantes e sigam as melhores práticas. Entretanto, em equipes de porte médio ou grande, isso pode se mostrar uma tarefa de difícil gerenciamento e controle, que necessita de automação. Desse modo, sugere-se que a detecção de código comentado seja incluído em processos de integração contínua para que a verificação seja constante.

Em casos particulares, a taxa de código comentado pode aumentar. É importante notar que 45 dos 100 sistemas analisados aumentaram suas taxas de código comentado. Isso é preocupante, pois esses projetos são extremamente populares e relevantes, sendo usados por milhões de desenvolvedores. Além disso, esses sistemas utilizam o controle de versão Git. De fato, um das motivações para comentar código e mantê-los ao longo do tempo é justamente para "lembrar" o trecho de código caso ele seja necessário no futuro, evitando re-implementação [Spinellis 2005]. Portanto, é contraditório que os sistemas analisados incluam essa má prática, pois manter o histórico de estados anteriores é justamente a principal função do controle de versão. Comentar trechos de código pode ser uma forma rápida de realizar testes durante uma seção de programação, mas desenvolvedores não deveriam fazer commits contendo esses trechos.

6. Ameaças à Validade

Detecção de Código Comentado. Neste trabalho foi desenvolvida uma ferramenta para detectar código comentado. Para avaliar a qualidade dessa ferramenta, foi mensurada sua acurácia em detectar corretamente código comentado (ver Seção 3). Em uma avaliação com 2 mil comentários rotulados manualmente, obteve-se precisão de 83,8% e revocação de 91,6%. Desse modo, considera-se o risco dessa ameaça baixo.

Generalização dos Resultados. Este estudo limitou-se à análise de 100 bibliotecas *open source* implementadas em Java. Portanto, os resultados não podem ser generalizados para outras linguagens ou para sistemas comerciais.

7. Trabalhos Relacionados

Apesar das críticas da literatura contra comentar trechos de código [Martin 2009], nosso estudo revelou a presença dessa má prática. No melhor do nosso conhecimento, poucos artigos acadêmicos abordam esse tema, e grande parte de forma secundária ou superficial. Por exemplo, um estudo exploratório sobre estratégias de *backtracking* utilizada por desenvolvedores indica uma prevalência de trechos de código comentados ao invés da deleção [Yoon and Myers 2012]. Além disso, o estudo aponta motivações comuns que levam os desenvolvedores a optar por essa prática. O modelo de qualidade de software SQALE (*Software Quality Assessment Based on Lifecycle Expectations*), que se propõe a avaliar a qualidade de código-fonte, mostra que comentá-los apenas aumenta o esforço de leitura sem agregar qualquer valor ao software [Letouzey 2012, Letouzey and Coq 2010]. Apesar das ferramentas de análise estática terem um papel importante para garantir qualidade, atualmente, somente o SonarQube pode detectar trechos de código comentado [Campbell and Papapetrou 2013]. A pesquisa proposta também se relaciona com

estudos sobre violações (*bad smells*) de código. Nesse contexto, diversas abordagens são propostas, por exemplo, para priorizar violações [Kim and Ernst 2007, Allier et al. 2012], avaliar a relevância dessas violações [Araujo et al. 2011] e detectar violações específicas por sistema [Hora et al. 2013, Hora et al. 2015]; tais abordagens, entretanto, não focam na análise de código comentado.

8. Conclusão

Este artigo apresentou um estudo empírico para quantificar a ocorrência de código comentado em 100 sistemas reais e 300 releases. Observou-se que, na mediana, 4,17% dos comentários são compostos por trechos de código e que essa taxa tende a diminuir durante a evolução dos projetos. Entretanto, em casos particulares, a taxa de código comentado é bastante significava, chegando a 30%. Como trabalhos futuros planeja-se (i) aumentar a quantidade de sistemas analisados para melhor caracterizar esse fenômeno, (ii) aumentar a acurácia da ferramenta e (iii) entender melhor quais trechos de código estão sendo comentados (eg, assinaturas, declarações, parâmetros, entre outros).

Agradecimentos: Esta pesquisa é financiada pela UFMS.

References

Allier, S., Anquetil, N., Hora, A., and Ducasse, S. (2012). A framework to compare alert ranking algorithms. In *Working Conference on Reverse Engineering (WCRE)*.

Araujo, J. E., Souza, S., and Valente, M. T. (2011). A study on the relevance of the warnings reported by Java bug finding tools. *IET Software*, 5(4).

Campbell, G. A. and Papapetrou, P. P. (2013). *SonarQube in Action*. Manning Publications Co., 1st edition.

Hora, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2013). Mining system specific rules from change patterns. In *Working Conference on Reverse Engineering (WCRE)*.

Hora, A., Anquetil, N., Etien, A., Ducasse, S., and Valente, M. T. (2015). Automatic detection of system-specific conventions unknown to developers. *Journal of Systems and Software*, 109.

Kim, S. and Ernst, M. D. (2007). Which warnings should i fix first? In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Letouzey, J.-L. (2012). The SQALE method for evaluating technical debt. In *International Workshop on Managing Technical Debt*.

Letouzey, J.-L. and Coq, T. (2010). The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. In *International Conference on Advances in System Testing and Validation Lifecycle*.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Spinellis, D. (2005). Version control systems. *IEEE Software*, 22(5).

Yoon, Y. and Myers, B. A. (2012). An exploratory study of backtracking strategies used by developers. In *International Workshop on Co-operative and Human Aspects of Software Engineering*.