**Audios**

**Conversa**

* Audio Google Meet: https://drive.google.com/file/d/1Vgvp02FswkhV7Ec1cijHx5MS\_oOuvA7O/view?usp=sharing
* Audio local Adolfo: https://drive.google.com/file/d/1epECjxs5QAV2sUuZO1cw8Tgyf4pLkpZm/view?usp=sharing
* Audio local Gustavo: https://www.dropbox.com/s/ua89fow3y13avyp/fes-marcelo-ufu.mp3?dl=0

**Encerramento:**

* **https://drive.google.com/file/d/1Hd5f0WOCiQg5OkmaKx-4S0uZC244p3d3/view?usp=sharing**

**Fronteiras da Engenharia de Software - Podcast**

Host: Adolfo Neto (UTFPR)

Co-host: Gustavo Pinto (UFPA)

Equipe: Marco Tulio Valente (UFMG), Danilo Ribeiro (Zup), Leonardo Fernandes (IFAL), Fabio Petrillo (Univ. Quebec), Marcela dos Santos (Univ. Quebec)

**Introdução ao Episódio**

[ADOLFO] Olá ouvintes, eu sou Adolfo Neto, professor da Universidade Tecnológica Federal do Paraná.

[GUSTAVO] E eu sou Gustavo Pinto, professor da UFPA, e este é o Fronteiras da Engenharia de Software, um podcast feito para refletir sobre o presente e futuro da engenharia de software

[ADOLFO] Fronteiras da Engenharia de Software é um podcast de divulgação científica que discute mensalmente diversificados temas da Engenharia de Software. Os episódios abordam desde questões fundamentais da engenharia de software até os mais recentes avanços na área.

[GUSTAVO] Para o nosso quarto episódio, entrevistamos Marcelo Maia, que é professor titular da Universidade Federal de Uberlândia, onde lidera o Laboratório de Engenharia de Software Inteligente.

[ADOLFO] Nos últimos anos, Marcelo tem trabalhado na área de Mineração de Repositórios de Software, frequentemente interagindo com questões relacionadas a como extrair o conhecimento embutido nos dados de repositórios de diversas naturezas e como transformar este conhecimento em ferramentas ou práticas que facilitem a vida do desenvolvedor de software. Neste episódio, vamos conversar sobre *bad smells*. Vem com a gente!

**Episódio 04: Code smells, com Marcelo Maia (UFU)**

**Parte 1: Apresentação [5 min, estimativa]**

[ADOLFO] Olá ouvintes, para este quarto episódio, contamos com a presença de Marcelo Maia, que é professor titular na UFU.

[ADOLFO] Oi Marcelo, tudo bem? Você pode se apresentar para nossos ouvintes?

[MARCELO] Olá Adolfo, Gustavo. Olá ouvintes do Fronteiras de Engenharia de Software. Antes de mais nada, gostaria de parabenizar a iniciativa de criar o Fronteiras da Engenharia de Software e é um prazer e uma honra poder estar aqui com vocês. Eu comecei minha carreira ainda na graduação em 1989 trabalhando como Programador/Analista desenvolvendo sistemas de informação para uma atacadista em C apoiado por uma biblioteca de B-trees. Era impressionante o que dava para rodar em um “supermicro” Edisa. Mas ao formar, decidi seguir para o mestrado trabalhando na área de linguagens de programação com o prof. Bigonha de UFMG. Entrei como professor assistente na UFOP e decidi seguir carreira acadêmica, seguindo para o doutorado na área de Métodos Formais. Ao retornar para Ouro Preto, após concluir o doutorado ainda no século passado :), percebi que seria difícil convencer os alunos a fazer pesquisa em Métodos Formais e decidi mudar minha linha de pesquisa. Como meu background era em linguagens de programação, o livro de Refactoring do Martin Fowler já estava dando o que falar, e tinha visto um trabalho do Paulo Borba da UFPE em uma linguagem de transformação de programas, decidi abrir um trabalho nesta linha. Criamos um ambiente de metaprogramação MetaJ que permitir escrever refactorings de maneira programática, usando basicamente um Yacc para Java, sem JDT (o que teria facilitado enormente a vida naquela época). Na época o prof. Eduardo Figueiredo da UFMG, trabalhava conosco em um projeto ambicioso de refatorar automaticamente programas usando Algoritmos Genéticos usando como fitness functions, métricas estruturais. A ideia apesar de ambiciosa demais para um trabalho de graduação em 2003, se mostrou viável em reparo automático de software com o Genprog (que foi a base da tese de doutorado da Claire Le Goues) aparecendo na ICSE’2009 e impulsionando a área de reparo automático de software. Ao me transferir para UFU e ingressar como docente no PPGCO, decidi novamente repensar a linha de pesquisa por volta de 2010 e com o boom dos repositórios de software (Sourceforge, github, …) e com nosso background de pesquisa vindo de ferramentas para análise/manipulação de código, decidi analisar repositórios software em larga escala para extrair dos dados brutos, informação que possa ser transferida como conhecimento prático para o desenvolvedor que é basicamente a linha com a qual venho atuando hoje dia.

[GUSTAVO] Você atua em engenharia de software há alguma longa data. Olhando seus trabalhos recentes, você parece ter um interesse bem diversificado. Em que parte da engenharia de software você mais concentra seus esforços hoje?

Definir temas para se pesquisar é uma tarefa desafiadora e muitas vezes a escolha do tema vai definir o sucesso ou não da pesquisa. É importante achar o timing certo para começar a trabalhar nos problemas. Muitos dos trabalhos recentes que desenvolvemos ao longo dos últimos 5 anos envolveu estudar repositórios em larga escala com métodos quantitativos ou alguns estudos mais qualitativos em menor escala de dados. Como exemplo do primeiro cito, um trabalho que desenvolvemos com o Marco Túlio, analisando co-changes em larga escala para encontrar padrões de co-mudança em classes e como isto poderia impactar a qualidade do software. Um outro estudo em menor escala, qualitativo, que teve uma resposta bastante positiva da comunidade foi dissecar um dataset de bugs para reparo automático (o Defects4J). Este estudo ao mostrar os detalhes e características do Defects4J permite entender melhor a performance das ferramentas de reparo de forma a direcionar a evolução das mesmas.

Ao abordar estes temas, inevitavelmente acabei percebendo que os métodos quantitativos mais poderosos para “entender” os dados brutos de repositórios acabam sendo os métodos de Aprendizado de Máquina (Machine Learning). Este tema por si só já é bastante abrangente, dado a comunidade tem aplicado Machine Learning nas mais diversas áreas de Engenharia de Software. Particularmente vamos nos concentrar em tarefas relacionadas a código-fonte, “autocompletion”, “localização de bugs”, information retrieval em documentação de software para prover soluções de problemas, … Acho que vale a pena investigar isto porque ao mesmo tempo que existe uma certa “moda” em fazer trabalho do tipo “Deep alguma coisa” em referência a Deep Learning, por outro lado, não necessariamente estes modelos são uma panaceia que oferecerem melhores resultados para tudo.

**Parte 2 - Code smells [20 min, estimativa]**

[GUSTAVO] Marcelo, você parece ter um interesse recente sobre code smells? O que seria code smell? Ou deveria chamar bad smell? Quando foi que isso tudo começou? <AGUARDAR RESPONDER> E hoje em dia? Com a modernização de linguagens e ambientes de desenvolvimento, mudou o entendimento do que seria um smell?

Interessante este ponto. Lembre que nos inícios dos anos 2000 com o livro de Refatoração do Fowler recentemente lançado, eu comecei a investigar ferramentas para refactoring, o tema Bad smells já se aproximava de mim. O Kent Beck que é co-autor do capítulo de Bad Smells no livro do Fowler, já tinha proposto esta metáfora pouco antes do livro em um artigo na sua wiki (o conceito de wiki, inclusive havia sido proposto há pouco tempo então). Isto quer dizer que a terminologia de bad smells só apareceu no final da década de 90. Mas na verdade, acho que o primeiro programa escrito já deve ter sido escrito com algum tipo bad smell. :) Apesar da metáfora, ter aparecido no final dos anos 90, já em meados dos anos 70, Guy Steele já usava o termo código spaghetti em um artigo. Isto sem dizer que 1968, um artigo do Dijkstra que ficou super-famoso foi o “Goto statement considered harmful” publicado na CACM. Na verdade o que o Dijkstra naquela época quis dizer era que Goto era um bad smell e como refatorar código espaguete cheio de gotos não deve ser uma das tarefas mais agradáveis de se fazer, ele simplesmente pregava que se abolisse o goto das linguagens para evitar a tentação de comer a maçã chamada “goto”. A história deu razão a ele.

Hoje em dia, como não temos goto em nossas linguagens, o código espaguete que existia nas décadas de 60/70, talvez não existam mais hoje. Mas não significa que hoje não tenhamos maçãs do pecado da programação. Quem nunca ouviu dizer nos termos “copy-and-paste oriented programming” ou “Stack-overflow oriented programming”? Na verdade, se por um lado a ampla disponibilidade de código na Web é um recurso fantástico para os desenvolvedores, por outro lado o uso das teclas Ctrl-C e Ctrl-V pode trazer problemas de duplicação de código que “podem” ter consequências danosas para um sistema. Código duplicado, também chamados de clones, é um outro exemplo de smell que foi caracterizado muito antes de Fowler e Beck escrevessem sobre “bad smells”. Os artigos científicos sobre código duplicado começaram a surgir no início da década de 90, e é o tipo de smell mais estudado na literatura.

O interessante do trabalho de Fowler e Beck foi criar uma metáfora poderosa (poderosa porque é ao mesmo tempo engraçada, sugestiva, e trata de um problema potencialmente relevante) que abriu espaço para que uma série de “potenciais problemas” de código pudessem ser catalogados e publicados de uma maneira sistemática. Em um recente estudo que fizemos, de 1990 a 2017 foram publicados mais de 100 tipos diferentes de code smells, sem contar aqueles que são muito similares mas que levam nomes diferentes.

[ADOLFO] Quais são os três principais *code smells* que você poderia citar? Por que eles são importantes, e quais são os problemas que eles podem causar?

Escolher três smells em mais de 100 não é uma tarefa fácil… Muito provavelmente, cada especialista em smells poderia escolher diferentes smells dependendo do olhar ou do critério adotado. Mas é interessante este desafio, e vou citar alguns smells que julgo especialmente interessantes:

1. O primeiro é o “código duplicado”, primeiro porque duplicação de código é uma prática reconhecida de longa data, um smell bastante estudado, com diversas ferramentas de detecção.
2. Depois citaria os bloaters (smells de inchaço), em particular, Large Class e Long Method que também são smells bastante estudados, prevalentes, e que em qualquer disciplina de POO é um dos primeiros pontos que precisam ser enfatizados… Quando o paradigma de programação orientada a objetos começou a se tornar popular, logo as pessoas perceberam que não ia ser uma panaceia para resolver a crise do Software, termo cunhado no início dos anos 80, porque você pode muito bem usar uma linguagem OO e criar um programa que não implemente adequamente os conceitos de OO. Implementar adequadamente OO significa seguir princípios que foram estabelecidos ao longo da prática de OO. Um destes princípios é o princípio da responsabilidade única que está relacionado à coesão entre módulos. Quando sua classe fere este princípio, ou seja, implementa várias responsabilidades, ela tem a chance de se tornar grande e estas condições de tamanho e várias responsabilidades implica possivelmente em impacto na manutenção e evolução da classe. Sobre o Long Method é um smell que nos é ensinado a evitar, assim que começamos a programar. Se não fizermos uma decomposição funcional adequada dos métodos, existe a chance de eles se tornarem grandes e possivelmente gerar impactos de manutenção e evolução também.
3. Em terceiro, eu citaria um smell relacionado a distribuição de responsabilidades entre as classes. Em disciplinas onde ensino POO, eu percebo que uma grande dificuldade de iniciantes é colocar as responsabilidades nas classes mais apropriadas. Existe um smell chamado de Feature Envy (Inveja de característica/responsabilidade), onde um método de uma classe ao invés de manipular suas próprias informações, recebe um objeto como parâmetro e passa a manipular as informações daquele objeto.

Estes smells que citei são os 4 smells mais estudados segundo nossa revisão da literatura publicada no final de 2018.

[ADOLFO] De que forma code smells (ou bad smells, depende do que ele responder) se manifestam em um código? Como alguém pode encontrar um bad smell?

Como se manifestam:

Veja que os smells são uma metáfora para um problema potencial. O código de um sistema pode ser comparado a um organismo vivo. Muitas vezes os problemas de design são sistêmicos. Não é um simples cano rachado que você conserta localmente. O grupo do Alessandro Garcia da PUC-Rio tem um trabalho interessante que mostra que os problemas de design (mais alto nível) podem estar associados a vários smells de código que se relacionam localmente. Isto acontece porque a metáfora de bad code smell originalmente proposta por Fowler/Beck são sintomas atômicos que podem ser percebidos em entidades mais localizadas, como um método ou uma classe, ou algumas entidades relacionadas (como o Divergent Change ou Shotgun surgery). Obviamente este olhar mais localizado facilita apontar estas possíveis anomalias, mas por outro lado pode esconder problemas mais sistêmicos.

Como encontrar:

Mas voltando ao problema de como localizar um code smell, existe uma certa área cinzenta na identificação de um smell que vai ter um impacto importante nos métodos de detecção de smells.

Vamos tomar o Large Class como exemplo. Fowler/Beck dizem que uma Large Class é uma classe que tenta fazer demais e que possivelmente por conta disto terá muitas variáveis de instância. Mas o quanto é “demais”? A partir de quantas variáveis, consideramos que são “muitas”? Obviamente, não há resposta definitiva para estas perguntas. Por exemplo, uma ferramenta chamada Decor, apesar de ser configurável, define por padrão que um Blob é também uma Large Class com ALTO número de atributos e métodos, onde ALTO é definido pelo boxplot da métrica em questão com mais um outro parâmetro multiplicador de fuzziness para não deixar os limiares rígidos, enquanto o livro Antipatterns de Brown et al, define que Blobs tem mais de 60 atributos/métodos. Isto é para dar apenas uma ideia da incerteza que é definir estes limiares. Inclusive alguns estudos estão justamente relacionados a diferentes percepções de smells. De certa forma, falar sobre os extremos é mais fácil… Recentemente usei um código de um aluno meu, cujo projeto foi se tornando mais complexo ao longo do desenvolvimento. No final ele tinha um método gigantesco que controlava toda a operação do seu projeto. Lá ninguém iria duvidar que o método era gigantesco e que uma refatoração iria melhorar bastante a legibilidade do código.

[ADOLFO] Como é possível identificar smells? Sei que existe muito de experiência do desenvolvedor, mas existem métodos/técnicas que auxiliam para se encontrar smells?

Certamente, nós da Ciência da Computação adoramos algoritmizar tudo. Na verdade existem várias soluções sejam acadêmicas ou industriais que se propuseram a criar regras que podem ser executadas algoritmicamente para apontar se uma determinada entidade do código possue ou não um smell. Só para se ter uma ideia, no caso de clones (código duplicado), foi criada uma caracterização de clones (tipo 1, tipo 2, etc) onde os clones são definidos desde aquele mais simples, cujo o código é exatamente igual, aqueles onde só nomes de identificadores são modificados, até os mais complexos onde a sintaxe é diferente mas a semântica é similar. No caso geral, a equivalência de programas é até um problema indecidível. Mas nos casos mais simples, existem um série de detectores automáticos de clone que funcionam muito bem.

Para smells, como Large Class, Large Method, Feature Envy, etc, também foram desenvolvidas um número extenso de ferramentas, em geral especializadas em um subconjunto de smells. Em nosso estudo que mencionei anteriormente identificamos mais de 80 ferramentas e/ou técnicas para detecção de smells que foram usadas em estudos científicos.

[GUSTAVO] Todos os code smells são nocivos a um sistema? Ou seja, é importante remover *todos* os code smells? Seria ingênuo da minha parte imaginar que todos os smells serão removidos, ou posso me confortar em saber que há code smells que eu nem preciso me preocupar? <<AGUARDAR RESPONDER>> Existem smells que são mais importantes que outros? Quais smells deveríamos priorizar durante sessões de refactoring?

Esta é uma pergunta importante e difícil de ser respondida. Em 2008, eu estava em uma conferência e em uma das discussões sobre clones, um dos autores mostrou que os clones existem em todo software, no kernel do Linux por exemplo, e não necessariamente os desenvolvedores vão se preocupar com eles. Mesmo não trabalhando com clones naquela época, aquela discussão me chamou a atenção. Para mim fazia parte do “senso comum” que clones (ou qualquer outro smell) são maléficos e DEVERIAM SER REMOVIDOS. Uma justificativa muito simples me levou a entender que as coisas não são necessariamente assim: ao reestruturar código, você pode inadvertidamente inserir erros que podem não ser descobertos em uma fase de teste. Isto é especialmente arriscado se estamos falando de código que funciona perfeitamente há um bom tempo. Faz sentido correr o risco? Para muitos desenvolvedores não e isto pode explicar porque estudos mostram que muitos sistemas apresentam smells e os desenvolvedores não se preocupam em removê-los.

Voltando a sua pergunta, então como podemos (se é que conseguimos) priorizar os code smells que podem ser mais maléficos ao sistema? Este é um tópico de pesquisa razoavelmente quente, onde já foram propostas várias soluções, mas até onde conheço não existe uma solução matadora, até por conta da dificuldade de se estabelecer o grau de malefício que um smell pode trazer.

Nossa revisão sistemática inclusive apontou diversos achados, aparentemente contraditórios na literatura, por exemplo, apontando que bad smells interferem na produtividade da manutenção e outro estudo apontando que bad smells não necessariamente interferem na produtividade de manutenção. Alguns estudos apontavam que smells tornavam o código mais propenso a defeitos, mas outros estudos não encontraram associação de smells com propensão a defeitos dependendo do sistema.

Um coisa que é importante lembrar que os estudos científicos têm suas limitações/ameaças à validade, seus contextos e que avaliar os resultados destes trabalhos requer uma cautela adicional para entender o quanto os resultados são generalizáveis.

No final das contas, a experiência dos desenvolvedores no seu contexto específico vão trazer reflexões melhores do que receitas pré-fabricadas de alguma prática a ser imposta. Obviamente, a prática e experiência dos outros são ótimas ferramentas para compor nosso arcabouço de decisão, mas precisam de ser aplicados de forma consciente.

Mas enfim, a linha de pesquisa em priorização de smells continua sendo bastante ativa e de interesse.

[GUSTAVO] refatoração é uma forma comumente utilizada para se remover bad smells, mas, como podemos evitá-los para que eles nem apareçam? Quais são os grupos de práticas e ferramentas que podem ajudar desenvolvedores a passarem longe de um bad smell?

Um estudo interessante publicado em 2015 mostrou que boa parte dos smells, exceto alguns bloaters, ao contrário do que o senso comum indica, já nascem com a entidade ao invés de serem adicionados ao longo do tempo.

Em outras palavras, isto indica que uma parcela significativa dos smells já nascem com o design. Considerando esta premissa, a aplicação de princípios e padrões de projeto para o design inicial das classes pode ajudar.

Por outro lado, a partir do momento em que desenvolvedores adotam metodologias derivadas dos valores e princípios ágeis, o refactoring é assumido como atividade inerente ao desenvolvimento. Por exemplo, quando se opta por adiar uma decisão de projeto por eventualmente não se ter as melhores informações para tomar a decisão em um dado momento, e ao mesmo tempo se opta por fazer uma entrega rápida e continuada, parece fazer sentido que o desenvolvedor não terá todos os elementos para fazer o melhor design e que futuramente o design deverá ser repensado e portanto algum tipo de refatoração possa acontecer.

O estudo que mencionei acima ainda mostra dados até certo ponto surpreendentes. Supostamente seria de se esperar que equipes de desenvolvimento trabalhassem em remover se não todos, mas uma boa parte dos smells do código, certo?

Eles estudaram 200 sistemas nos ecosistemas, Android, Apache e Eclipse, por um tempo médio de 6 anos e descobriram que 80% dos smells encontrados (procuraram por Blob Class, Class Data should be private, Complex Class, Functional Decomposition, Spaghetti Code) não são removidos do sistema. Dos 20% que são removidos, apenas 9% são removidos com refactoring.

Uma das mensagens importantes deste trabalho é que de fato os desenvolvedores por diferentes razões vão aceitar conviver com os smells.

Um exemplo que eu gosto muito de citar é o caso da classe IndexWriter do Lucene (Search engine da Apache). Esta é indubitavelmente uma Large Class, inclusive auto-admitida pelos próprios desenvolvedores do Lucene no Issue Tracker. Os desenvolvedores sabem que ela é uma Large Class, até fizeram uma refatoração de longo prazo (eu explico) mas continuava Large Class. A refatoração de longo a qual me referi foi um Extract Class, mas que não ocorreu nos moldes tradicionais. Ao invés disso, a suposta classe “extraída” continha parâmetros de configuração da IndexWriter e foi chamada de IndexWriterConfig. Esta classe ficou no sistema junto com as mesmas variáveis de configuração dentro da própria IndexWriter. Dai colocou-se os métodos que acessavam as variáveis de configuração como “deprecated” para que os clientes se adaptassem à mudança. Finalmente algumas releases depois a refatoração foi concluída tirando os campos de configuração com respectivos getters/setters. Moral da história: refatoração nem sempre é um processo atômico, nem sempre resolve o problema do smell. Até onde eu acompanhei IndexWriter continuava como LargeClass e os desenvolvedores não tinham uma melhor solução para ela.

Concluíndo a resposta para sua pergunta Gustavo, infelizmente me parece que não tem como ficar livre de smells, assim como não conseguimos ficar livre dos vírus e bactérias… o que podemos fazer é criar ferramentas e métodos (e vacinas) para conseguir conviver em paz com eles e fazer com que os danos fiquem em padrões aceitáveis.

[ADOLFO] qual é o status atual das ferramentas de detecção de bad smell? Funcionam? Somente um grupo muito específico de smells? Smell triviais? <AGUARDAR> Quais as vantagens/desvantagens da (FERRAMENTA MENCIONADA).

De novo, uma ótima pergunta que já adianto que não vou dar uma resposta satisfatória. Nosso estudo detectou uma carência importante de ground-truth para estabelecer benchmarks que possam comparar as ferramentas por exemplo em precisão e revocação. Uma ferramenta com baixa precisão vai nos apontar smells em muitas entidades que de fato não tem o smell. Uma ferramenta de baixa revocação não consegue detectar smells existentes. Na verdade, existe iniciativa chamada de Landfill para estabelecer um ground-truth de code smells para que as ferramentas possam ser comparadas. Eles tinham smells apenas para Long Method, Blob e Feature Envy. Em agosto, participei de uma banca de mestrado na UFMG de um aluno da Mariza Bigonha e da Kécia Ferreira, onde ele propôs um benchmark curado para avaliar ferramentas de bad smells, dentre elas JDeodorant, JSpirit, FindSmells, … o melhor resultado foi para Large Class/Large Method com F-Measure 0.6, no JSpirit, mas o qual ficou com F-Measure abaixo de 0.35 para Refused Bequest, Data Class e Feature Envy.

Em resumo, as ferramentas são pouco testadas por conta da falta de benchmarks e para os benchmarks que existem os resultados não são dos melhores. Mas não implica que estas ferramentas não seja úteis. O desenvolvedor tem que ter em mente que a acurácia delas é limitada, mas estas ferramentas podem ser usadas como uma ferramenta de triagem que vai separar candidatos a bad smells que podem ser avaliados qualitativamente pelo desenvolvedor.

Na nossa revisão sistemática, já havíamos relatado que a falta destes benchmarks é um importante desafio para o avanço da fronteira em bad smells.

[GUSTAVO] Smells podem levar a bugs? Por exemplo, a ocorrência de vários smells em conjunto pode tornar um sistema de software mais propenso a falhas? <AGUARDAR> Isso é uma pergunta de pesquisa? Ou é algo que já foi bem explorado?

Esta pergunta foi razoavelmente explorada na literatura. Existem pelo menos 11 trabalhos que estudaram esta relação, e a grande maioria aponta algum tipo de maior risco de bugs associados a bad smells.

Mas é importante notar que isto não é uma regra universal. Um ponto interessante é que alguns trabalhos acharam uma variabilidade de intensidade de associação em alguns casos ou até mesmo nenhuma associação, dependendendo da release ou do smell estudados.

Vou mencionar alguns trabalhos que considero interessantes. Um dos principais grupos de pesquisa em bad smells está sediado na Escola Politécnica de Montreal. Eles desenvolveram DECOR que é a ferramenta mais utilizada em estudos de bad smells, até porque muitos dos estudos foram desenvolvidos por eles próprios ou por parceiros de pesquisa. DECOR também é uma ferramenta capaz de detectar um grande número de tipos diferentes de smells. Em um estudo de 2012, eles observaram que no Eclipse classes com bad smells (que não fossem Large Class e nem Spaghetti code, o que é curioso) tinham risco mais alto (2x) de apresentar issues que outras classes. Depois em 2015 eles ainda virão que as classes com dependência estática a classes com bad smells eram também mais propensas a issues. E de maneira interessante, classes com smells mas com dependências a classes com design pattern tinham menos propensão a bugs que outras classes com smell. Um outro estudo publicado na Transactions de Eng. Soft da ACM com 5 smells menos estudados (Data Clumps, Switch Statements, Generalidade especulativa, Message chains, e middle man) em 3 sistemas (ArgoUML, Eclipse e Apache Commons) mostrou que alguns destes smells aumentavam a propensão a bugs em alguns sistemas, outros não tinham nenhuma associação com bugs, e mais surpreendente alguns smells associavam negativamente com bugs em alguns sistemas. Concluíram então sobre a importância da priorização que falamos anteriormente e que refatorar arbitrariamente poderia inclusive aumentar a propensão a bugs.

~~[ADOLFO] Bad smells são intrínsecos a linguagem de programação? Por exemplo, existem bad smells em Java que não existem em Ruby? Ou existem bad smells onipresentes em todas (ou várias) linguagens de programação? <AGUARDAR> Existem bad smells culturais? Por ex, comunidades ao redor de um ecossistema podem ser mais sensíveis a um grupo de bad smells?~~

[ADOLFO] Os smells de código também se aplicam a código de teste? Ou são outros grupos de smells?

Sim… a área de smells de teste é uma área a parte para a qual foram definidos os respectivos smells próprios para eles. Afinal de contas, código de teste também é código :)

Este foi inclusive um desdobramento da pesquisa em smells: a proposta de smells em contextos específicos. Por exemplo, existe alguns smells que foram propostos especificamente para o ecosistema Android. Enfim, a metafóra de smells pode ser aplicada onde quer que haja “design” … sempre é possível elaborar um design sub-ótimo e portanto com possibilidade de enxergar um padrão recorrente e caracterizar um smell.

**Parte 3 - Pesquisa em Code smells [20 min, estimativa]**

[GUSTAVO] Você teve um trabalho recente sobre uma revisão da literatura sobre code smells, com um título bem sugestivo (5 W’s: which, when, what, who, where). Você pode explicar pra gente um pouco sobre esse trabalho? Qual metodologia que você usou?

<Talvez esta pergunta pudesse aparecer mais cedo, porque em várias vezes na minha fala acima eu referencio este trabalho>

Este trabalho foi desenvolvido no contexto da tese de doutorado de um aluno meu, o Elder Sobrinho e em parceria com o prof. Andrea de Lucia da Italia.

O objetivo inicialmente era modesto, apenas para ajudar a estabelecer um problema de pesquisa para o Elder e mapear as pessoas que trabalhavam na área. Como o Elder foi muito detalhista, acabou virando um “Large Paper” se é que isto é um Paper Smell (rs rs).

Este artigo tem um diferencial metodológico interessante porque estávamos encontrando diversas limitações para aplicar metodologias de busca e filtragem tradicionais de revisões sistemáticas. Então resolvemos adotar uma metodologia “hardcore” e coletar todos os papers de venues importantes de Engenharia de Software e fazer filtragem manual do artigos, ao invés por exemplo de formular queries de busca nas principais bases de dados. Depois de um snowballing e etc chegamos a 351 artigos dos quais ⅔ eram sobre Duplicate Code, para terem uma ideia de que clones de fato é uma linha de pesquisa a parte. No final o artigo tem uma lista de 417 referências.

Uma das perguntas era entender se os estudos tentavam estudar inter-relações de smells no código e acabamos vendo que esta é uma área que pode ser melhor explorada.

~~[GUSTAVO] Há alguns trabalhos que sugerem para~~ *~~não~~* ~~fazer deploy, vocês encontraram algo similar para os code smells? Por exemplo, há algum dia/horário durante a semana que seria melhor nem tocar no código pq se não eu posso inserir um smell? Faz sentido isso?~~

~~Não analisamos este aspecto. Se existir indício em outros contextos que o dia/horário pode estar associados a problemas de desenvolvimento, por que não investigar se poderiam estar associados a introdução de smells no código?~~

~~[ADOLFO] Quais são as partes de um sistema que são mais propensas a terem smells? Tem como ter algum indício de onde vão aparecer smells? Por ex, nas partes do código que são mudadas com mais frequência?~~

Acho que o estudo que apontei anteriormente ajuda a responder esta pergunta. Considerando que o número de commits para um smell se manifestar é muito baixo, a mediana é zero, exceto para o Blob no Android (que é 3) e Complex Class no Eclipse (que é 1)... Isto significa que em regra geral, o smell ou nasce na classe, ou aparece muito cedo. E quando é removido também tende a ser removido rapidamente. Enfim, se ele já passou por um processo de revisão, o PR foi aceito, estabilizou vários commits, então o pessoal parece estar “conformado” com ele.

Obviamente há casos extremos que precisam de mais commits para se manifestar, que talvez merecessem uma análise qualitativa melhor, mas isto não é a regra geral.

Sobre partes do código serem mudadas com maior frequência pode até ser um bom fator para ela não ter smells porque o desenvolvedor não deve ter maiores problemas em mexer nelas.

Eu talvez diria que as classes mais estáveis talvez fossem as mais complicadas porque deve ter alguma razão para não mexer muito nela. Talvez as dependências e estabilidade da classe ajudem a explicar a dificuldade de remover smells.

~~[GUSTAVO] O que mais dá pra explorar em pesquisa em smells? O que esse estudo de vocês não responde… ?~~

~~Na verdade este paper nosso termina com 7 desafios e possíveis futuras pesquisas, o que não quer absolutamente dizer que sejam as únicas, mas foram lacunas que em 2018 identificamos no estado-da-arte.~~

* ~~Melhoria da acurácia das ferramentas de detecção com possíveis novos tipos de métricas: métricas de ambiente (p.ex. CI), métricas sociais~~
* ~~Monitoramento de smells com métricas de evolução: entender tendências que uma entidade tem pode ajudar a tomar decisão sobre uma ação mais efetiva~~
* ~~Incorporação da percepção de usuário sobre o smells nas ferramentas de detecção de smells, oferecendo um relatório mais informado sobre as possíveis consequências dos mesmos.~~
* ~~Construção de benchmarks representativos de smells para evolução desta área como um todo.~~
* ~~Melhor desenvolvimento do conceito de linhagem de smells, como diferentes smells podem co-ocorrer em uma entidade, ou como (e se) um tipo de smell pode se transformar em outro.~~
* ~~Estudo de smells em diferentes contextos, como p.ex., a proposta de smells para o contexto de Android (Limbo, Internal Getter/Setter)~~
* ~~Enfim, como tantas novas possibilidades em smells sendo propostas, novas taxonomias para entender e explicar melhor smells vão ser bem vindas.~~

**Parte 4: Indicações [3 min, estimativa]**

[GUSTAVO] Alguma mensagem para nossos ouvintes? Quem estiver interessado em começar a estudar/pesquisar code smells, por onde deve começar? Recomendação de livro ou algum outro material?

**Parte 6: Conclusão [3 min, estimativa]**

[ADOLFO] Para você, qual é a próxima fronteira da engenharia de software?

Eu assisti os episódios anteriores deste podcast e confesso que achei muito legal quando fizeram esta pergunta. É muito interessante porque nós como seres humanos temos uma curiosidade incrível de poder saber de antemão como será o futuro. Qual será a tecnologia dominante em 10 anos? Os carros autônomos vão ficar disponível quando?

Hoje acho que uma das perguntas que os desenvolvedores de software mais se fazem é::

- A profissão de programador vai acabar?

Eu confesso que eu pesquiso para isto (rs rs rs). Mas provavelmente minha pesquisa não vai chegar lá.

Outro dia me postaram uma foto do Turbo Pascal (kkk) da década de 80 e vejo que o IDEs de hoje, obviamente com muito mais recursos, seguem a mesma estrutura. Ou seja, não houve nada que eu considere disruptivo em desenvolvimento de software nas últimas décadas. Continuamos falando em Spaghetti Code 50 anos depois. Desde a década de 80, já se tinha a ideia de criar sistemas de transformação que gerassem código automaticamente… o DRACO era um deles e não se pode dizer que foi um sucesso, … depois apareceu MDA que era uma repaginação do DRACO… talvez as tecnologias de reuso, APIs são quem de fato tem dado um ganho de produtividade em software.

Se por um lado não me parece que chegamos no estado atual de uma maneira disruptiva, por outro lado, o Facebook em 2019 no ICSE, apresentou um artigo mostrando que tinha colocado uma ferramenta de Reparo Automático!!!! de Software em produção. Mais recente, houve um burburinho com o GPT-3, uma API/ferramenta de processamento de linguagem de natural ter sido aplicada para gerar alguns toy examples de código de maneira que eu consideraria disruptiva (se funcionar de verdade).

Vocês já devem ter assistido aqueles filmes de super-heróis, onde o vilão entra dentro de um exo-esqueleto aumentando enormemente suas capacidades … ou então o próprio Homem-de-ferro com sua armadura, certo?

Para mim na próxima fronteira da Engenharia de Software, os algoritmos de Aprendizado de Máquina devidamente treinados vão permitir a criação de um “exo-esqueleto” para o programador, o qual vai aumentar significativamente a sua capacidade de produzir artefatos de software de qualidade, mas por outro lado os “novos desenvolvedores” continuarão a ter que exercer sua condição humana de criação e de interação com outras pessoas e terão que aprender a fazer isto em um novo contexto onde estes robôs vão ter cada vez mais relevância.

**Para compartilhar**

* Aaa
* aaaa

**Texto para divulgação**

Na entrevista de hoje falamos com Marcelo Maia, que é professor titular da UFU. Marcelo tem ampla experiência em engenharia de software, tanto que a nossa conversa sobre code smells rendeu tanto que dividimos em duas partes. Nessa primeira parte, falamos sobre o que é um code smell, como identificar, e se é possível priorizar. Na segunda parte falamos como remover, sobre o status atual das ferramentas de identificação de smells (eles funcionam?), se code smells podem gerar bugs, além de um trabalho de pesquisa que Marcelo conduziu com seu aluno de doutorado para encontrar como, onde, ou quem são os responsável por adicionar smells num código.

Interessou? Escute o podcast e nos avise o que acho pelas redes sociais!

Sites do Marcelo

* <https://dblp.uni-trier.de/pid/97/4668.html>
* <http://www.portal.facom.ufu.br/pessoas/docentes/marcelo-de-almeida-maia>
* <http://lascam.facom.ufu.br/cms/>
* <https://scholar.google.com/citations?user=AkhgR1AAAAAJ&hl=en>
* <http://lattes.cnpq.br/4915659948263445>
* <http://isel.ufu.br>