UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MATHEUS KRAUSE

Detecção de Anomalias de Orientação a Objetos em C#

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof[a]. Dra. Érika Fernandes Cota

Porto Alegre
2019

“Viver é diferente de estar vivo.
Viver é ser feliz, sendo um cara extrovertido.”
— AZIZ “ZYZZ” SHAVERSHIAN

AGRADECIMENTOS

Agradeço a toda minha família pelos ensinamentos.

Agradeço ao meu colega e amigo Lucas por todo o suporte.

Agradeço a Iggy por toda a motivação dada.

E por fim gostaria de agradecer a minha orientadora, Érika Fernandes Cota, que me acompanhou nessa jornada.

RESUMO

O teste é uma prática importante de garantia de qualidade do processo moderno de desenvolvimento de software. Como não podemos dar ao luxo de testar todos os aspectos de um programa, é essencial definir o escopo desses testes (*isto é*, quais aspectos do software o teste está exercitando) a fim de tornar a técnica viável. A delimitação desse escopo é dificultada à medida que mais e mais camadas de abstração são incorporadas às linguagens de programação. Embora esses novos paradigmas e construções de programação possam aumentar a funcionalidade do código, eles também introduzem novas classes de falhas específicas de linguagem, algumas das quais são sutis e difíceis de testar computacionalmente. Isso deixa os testadores de software com a tarefa adicional de estabelecer criteriosamente se a gravidade dessas falhas específicas justifica o custo de elencar novos testes exclusivamente destinados a sua cobertura. Neste trabalho, analisamos um caso específico de anomalias introduzidas pela programação orientada a objetos, que são conhecidas como anomalias de código orientadas a objetos. Primeiro, analisamos a possibilidade dessas anomalias ocorrerem no ambiente C#, considerando as salvaguardas que o compilador e a sintaxe do idioma já fornecem contra essas falhas. Em seguida, avaliamos, com base em nossa análise, se essas falhas são impactantes o suficiente para serem dignas de metodologias específicas de teste.

Palavras-chave: Teste de Software. Analise estática. C#.

Object Oriented Anomalies analysis in C#

ABSTRACT

Testing is an important quality assurance practice of the modern software development process. Since we cannot afford to test every aspect of a program, defining which aspects will be exercised is essential in order to make testing feasible for integration with the development process. This definition process is made even harder as more and more abstraction layers are incorporated into programming languages. While these new programming paradigms and constructs may augment the functionality of code, they also introduce new classes of language-specific faults, some of which are subtle and computationally difficult to test for. This leaves software testers with the additional task of judiciously establish if the severity of these new language specific faults substantiates the overhead of testing for them. In this work, we aim for a specific case of anomalies introduced by object-oriented programming, which are object oriented code anomalies. We first discover the likelihood of these anomalies occurring on the C# environment, considering the safeguards that both the compiler and language syntax already place against these faults. We then assess, based on our analysis, if these faults are impactful enough to be worthy of having their own testing methodology. We argue that our practical study of these faults can serve as guidelines for C# software testers, ellucidating the circumstances surrounding each fault, as well as their probability of occurence and possible countermeasures.

Keywords: Software testing. Static analysis. C#.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Há algumas décadas atrás iniciou-se uma rápida transformação digital globalizada com sistemas digitais se tornando a essência do mundo moderno. Mundialmente mais de um terço das organizações (44.7%) já implementam uma abordagem *digital-first* para os processos de negócio, operações e engajamento com clientes (IDG, 2018). O objetivo é claro, busca-se automatizar todos os processos manuais visando menores custos, maior escalabilidade, menor tempo de reação e maior satisfação de consumidores (IVANOV et al., 2017).

Essa transformação acelerada trouxe uma série de questionamentos sobre a qualidade dos sistemas sendo desenvolvidos. A complexidade dos sistemas aumentava continuamente e rapidamente enquanto que o processo de qualidade ainda permanecia praticamente manual, não escalável. Esse descompasso fez com que a aplicação da técnica de teste fosse limitada, dado que aspectos muito complexos do software exigem muito esforço e o teste manual não consegue acompanhar o crescente volume de mudanças. O cenário citado culminou em um aumento colossal no prejuízo por parte de falhas de software (NATO, 1968). O contínuo aumento do número de problemas relacionado a softwares fez com que esta categoria de problemas se tornasse uma das maiores preocupações econômicas do presente, sendo esperado um prejuízo de 1.7 trilhões de dólares em 2017 (FRISCHKNECHT, 2018).

Em resposta a esse cenário caótico a área de garantia de qualidade teve uma ascensão de popularidade se tornando uma parte essencial do processo de desenvolvimento de Sofware. Teste de software se tornou a base do processo de garantia de qualidade em consequência do seu baixo custo relativo a técnicas alternativas e alta confiança nos resultados derivados da execução real do sistema (técnicas como análise estática não exercitam o sistema). A possibilidade ainda de se automatizar a tarefa viabilizou a sua adoção no contexto moderno ágil de desenvolvimento de software onde mudanças e entregas contínuas são o estado da arte (TECHNOLOGIESCIGNITI; SERVICES, 2018). A eficiência, flexibilidade e agilidade propiciadas pela automação de teste permitiram que a prática de teste de software se tornasse uma atividade escalável, em que é possível acompanhar o crescimento da complexidade e quantidade dos sistemas.

Apesar de todos os benefícios, testar software ainda envolve custo. Com a maturidade do processo diversas técnicas surgiram para buscar uma redução no impacto econômico do teste. A limitação do escopo a ser abrangido pelo teste é um dos artifícios

utilizados. Ao evitar, por exemplo, o exercício de certos caminhos de negócio no software se diminui o custo total agregado. É evidente porém que se esta limitação de escopo for aplicada de maneira não criteriosa há a possibilidade no aumento de falhas não capturadas pelos testes. O nível de exigência do teste, o custo/tempo e risco devem sempre ser ponderados.

Este trabalho busca dar suporte a essa atividade de teste de software, mais especificamente a limitação do escopo de teste, no ambiente .Net C# ao analisar a relevância das anomalias de orientação a objetos neste ambiente. As anomalias, que são pouco conhecidas tanto na indústria quanto na academia, são uma possível causa de bugs originados do mau uso dos recursos de herança e polimorfismo.

Neste trabalho inicialmente analisamos a possibilidade de ocorrência das anomalias no ambiente. Como C# é uma linguagem moderna, o compilador e a própria sintaxe da linguagem podem já ter mecanismos que impeçam a ocorrência das anomalias. Seguiu-se depois para a implementação de um analisador estático de código fonte da linguagem, com o objetivo de identificar as anomalias em um conjunto razoável de classes para se ter um panorama sobre as ocorrências das anomalias em código legado. Todos esses dados permitiriam formar uma opinião sobre a relevância das anomalias e a necessidade de definir estratégias de teste específicas para estas falhas.

Os resultados obtidos podem justificar o desconhecimento generalizado sobre as anomalias. O número de detecções foi baixíssimo e mesmo as poucas ocorrências não acarretavam em erros no sistema.

1.1 Organização do trabalho

O trabalho segue esta estrutura: o Capítulo 2 apresenta os conceitos básicos necessários para a compreensão do trabalho. No Capítulo 3, os trabalhos relacionados são apresentados. O quarto capítulo analisa, no ambiente C#, a possibilidade de ocorrência das anomalias. Em seguida os detalhes da implementação do analisador e as estratégias utilizadas para análise automática do código-fonte são apresentados. No quinto capítulo, os experimentos realizados e os resultados são apresentados e analisados. E por fim o último capítulo é destinado a conclusões e possibilidades de trabalhos futuros.

2 TECNOLOGIAS E CONCEITOS

2.1 Teste de Software

Teste de software é a prática de execução de programas a fim de buscar defeitos visando uma qualidade maior ao artefato sendo desenvolvido. É um dos principais métodos da área de garantia de qualidade na área de computação servindo para identificação prévia de problemas antes da entrega final do software ao cliente e para a redução do custo relacionado às consequências dos defeitos (MYERS; SANDLER; BADGETT, 2012)

A aplicação da técnica é flexível, tradicionalmente seguindo um processo com as seguintes etapas: planejamento dos casos de testes, implementação dos testes (no caso de testes automatizados), execução dos testes e análise dos resultados obtidos. Porém detalhes de sua implementação são específicos do testador variando de acordo com o conhecimento próprio e tradição da organização na qual o testador trabalha.

São duas questões que orientam o esforço de planejamento do teste visando limitar o escopo abrangido e o planejamento sistemático dos dados de entrada (MYERS; SANDLER; BADGETT, 2012):

1. Como definir que um programa foi testado o suficiente.

2. Como definir as entradas do teste.

Em 1 busca-se definir formalmente requisitos de teste que serão utilizados para o desenvolvimento de casos de teste, um acordo com as partes interessadas, como clientes, *product owners* e arquitetos, deve ser feito para concretizar o que será testado e se a cobertura está adequada.

Segundo Howden (HOWDEN, 1978), a questão 2 é tomada por duas ramificações principais que contêm diversas técnicas auxiliares que podem ser utilizadas para planejamento das entradas, elas são:

- Teste caixa branca - utilizar o conhecimento estrutural do programa a ser testado para definir entradas que exercitem os caminhos desejados;

- Teste caixa preta - evitar o código fonte e o conhecimento estrutural focando no conhecimento do domínio da aplicação e na especificação, ou seja, visando no que o programa deveria fazer em vez de como é feito.

Os testes podem ser categorizados quanto ao nível de abstração que foram arquitetados para cobrir (AMMANN; OFFUTT, 2008). É comum haver ao menos três classificações:

1. Teste unitário - testa a menor unidade do programa, evita dependência entre unidades a fim de ter uma rápida execução e avaliar puramente a lógica do módulo que está sendo testado.

2. Teste de integração - testa a agregação das unidades, utiliza as dependências reais para avaliar se a interface das unidades está sendo corretamente utilizada e/ou desenvolvida.

3. teste de sistema - todo o sistema é agregado e exercitado, buscando avaliar o comportamento no ponto de vista do usuário.

Segundo Jorgensen (JORGENSEN, 2014), o processo inicia pelos testes unitários que são responsabilidade do próprio desenvolvedor, seguindo para os de integração e por fim chegando aos de sistema cuja responsabilidade tipicamente assumida por uma equipe distinta da equipe de desenvolvimento. Cada etapa é mais custosa e lenta que a anterior, sendo esperado que a maior concentração de testes seja do tipo unitário seguido de integração e sistema. O ciclo de vida desse processo é flexível e depende fortemente do que está sendo testado. Encontramos fundamentalmente as fases a seguir:

1. Procedimentos inicias - Nesta etapa deverá ser aprofundado um estudo dos requisitos do negócio que dará origem ao sistema de informação a ser testado, de modo a garantir que o mesmo esteja completo e sem nenhuma ambiguidade.

2. Planejamento - Consiste em elaborar a Estratégia de Teste e o Plano de Teste a serem utilizados de modo a minimizar os principais riscos do negócio e fornecer os caminhos para as próximas etapas. O Plano de Teste pode ser mais ou menos formal e detalhado, dependendo da organização e do processo de desenvolvimento utilizado.

3. Preparação - Nesta etapa, o objetivo básico é preparar o ambiente de teste (equipamentos, pessoal, ferramentas de automação, hardware e software), para que os testes sejam executados corretamente.

4. Especificação - São elaborados e revisados os casos de teste e os roteiros teste. Os casos de teste e os roteiros de teste devem ser elaborados dinamicamente durante o decorrer do projeto de teste.

5. Execução - Os testes deverão ser executados de acordo com os casos de teste e os roteiros de teste. Devem ser usados scripts de teste, caso seja empregada alguma ferramenta de automação de testes.

6. Entrega - Esta fase é onde acontece a entrega do software depois de ter passado por todas as fases, com objetivo de minimizar todas as chances possíveis de erros (bugs).

2.2 Orientação a objetos

Orientação a Objetos é um paradigma de programação que define a organização de um software em volta de uma coleção de objetos discretos que incorporam tanto estrutura de dados como comportamento (MEYER, 2016).

O termo objeto é definido nesse contexto de desenvolvimento como uma unidade abstrata e limitada que detém significado claro no domínio de aplicação. Todo objeto possui um identificador único para distinção contra outros objetos, estado e comportamentos (BOOCH, 1991).

O estado de um objeto é o valor de suas propriedades. O comportamento são as ações que o objeto tem capacidade de realizar, normalmente são concretizados como métodos em linguagens de programação e comumente influenciam no estado do objeto.

Uma classe é um modelo abstrato que consiste em métodos, que descrevem o comportamento, e de atributos, que definem a estrutura de dados do estado, de um conjunto de objetos. Todos os objetos são instâncias de uma classe e honram as definições impostas por ela (RUMBAUGH, 1998).

O paradigma de Orientação a Objetos utiliza de uma grande gama de características que empoderam o desenvolvimento, entre os mais relevantes a este trabalho destacamos:

2.2.1 Encapsulamento

O processo de esconder a implementação concreta do objeto revelando ao exterior apenas o modelo abstrato do mesmo. Através da interface do objeto e do conhecimento da abstração o usuário deve ser capaz de obter o comportamento desejado. O benefício consiste na flexibilidade de poder mudar a sua implementação sem afetar as aplicações que o utilizem.

2.2.2 Hierarquia de Classes

Uma hierarquia de classe é resultado de uma relação entre classes. Esse relacionamento tem duas formas, herança e composição.

Na herança todos os atributos e métodos são compartilhados entre os membros do relacionamento. As subclasses herdam todas propriedades da superclasse e adicionam suas próprias propriedades. Uma classe pode portanto definir de modo bem abstrato um conceito e ser refinada sucessivamente por subclasses mais detalhadas que conterão todas as propriedades previamente definidas. A reutilização originada deste mecanismo é um dos grandes ganhos da Orientação a Objetos. A herança pode ser simples, cada subclasse tem apenas uma superclasse, ou múltipla onde uma subclasse pode ter múltiplas superclasses.

Composição define o relacionamento entre uma classe agregadora e outras classes componentes que a compõe. As classes componentes podem ou não existir independentemente da agregada. A classe agregada detém em seus atributos um objeto de cada tipo de componente e pode acessar o comportamento visível deles. Também é um recurso essencial de reúso visto que, ao invés de implementar novamente um comportamento desejado, uma classe pode ter em sua definição uma classe componente que já o implementa.

2.3 C#

C# é uma linguagem moderna de propósito geral, estaticamente tipada e multiparadigma desenvolvida pela Microsoft como um dos principais componentes da plataforma .Net (DRAYTON; ALBAHARI; NEWARD, 2003). É um membro da família de linguagens de programação C, possuindo fortes similaridades e herdando aspectos de outros

membros, como Java, de quem herdou o mecanismo de coleta de lixo, e C++, na qual se baseou para escolha da sintaxe atribuída aos mecanismos de herança (uso da palavra reservada *virtual* para permitir sobrecarga por exemplo).

A linguagem é mista no quesito compilação, sendo compilada para a Microsoft Intermediary Language e interpretada pelo *runtime* do .Net. Ela tem também sua especificação aberta, a sintaxe e regras semânticas são padronizadas pelo ECMA-334 e ISO/IEC 23270 (ISO/IEC. . . , 2006).

Um tipo em C# é composto por campos, métodos, propriedades e construtores. Cada tipo tem sempre ao menos um construtor que pode ou não ter argumentos. No caso da não existência explicita de ao menos um construtor, o compilador gera automaticamente um construtor padrão com corpo vazio. Os construtores somente são invocados no momento da instanciação pela palavra reservada *new*. As propriedades são um novo recurso da linguagem, são equivalentes a métodos *setters* e *getters* e tem como objetivo fornecer uma interface de acesso indireta aos campos. Um campo é um membro de classe que possui um tipo e nome, seu estado ou valor define o estado do objeto que o detém. Todos os campos e propriedades tem sempre um valor padrão consistente mesmo que não inicializados no construtor do tipo. Os métodos são usualmente responsáveis pelo comportamento ou ação do objeto, tem uma assinatura, constituída pelo nome e pelos argumentos, que o identifica de forma única na classe. Os métodos não guardam estado.

2.4 Orientação a Objetos em C#

Como uma linguagem orientada a objetos, C# inclui todos os mecanismos tradicionais do paradigma de Orientação a Objetos (OO), incluindo herança (não múltipla), polimorfismo e vínculo dinâmico. Diferentemente de JAVA, C# possui um sistema de tipo centralizado onde todos os tipos herdam do tipo raiz *Object*. Todos os tipos são uma classe(tipo referência), herdam de *class*, ou estrutura (tipo valor), herdam de *struct*.

O mecanismo de herança em C# é uma poderosa ferramenta de reúso, abstração e principalmente polimorfismo. Quando um tipo *A* herda de outro tipo *B* utilizando o operador de herança ":", *A* se torna uma subclasse de *B* ganhando acesso a todos os membros definidos em *B* com exceção dos que contenham o modificador *private*. O subtipo *A* também pode ser usado em todos os locais que *B* é esperado. Uma variável declarada como tipo *B* pode receber um valor do tipo *A* em uma atribuição diretamente, o contrário não é verdade.

Para haver a sobrescrita de um membro herdado, método ou propriedade, a definição original deve estar marcada com o modificador *virtual* e a subclasse deve então declarar um método de mesma assinatura porém com o modificador *override*. Caso a definição original não contenha o modificador *virtual*, a nova definição ocultará a definição original na hierarquia de herança, tornando o membro da superclasse escondido. Um membro escondido não é acessível na hierarquia de herança após o ponto de redeclaração pela subclasse porém continua existindo e sendo utilizado nas classes que antecedem esse evento. A diferença entre o membro escondido e sobrescrito é que naquele o tipo da variável na declaração, ou seja o tipo declarado, dita a versão usada independentemente da atribuição enquanto que neste a versão utilizada é definida pelo tipo atribuído a variável. O comportamento polimórfico ideal é obtido quando uma subclasse sobrescreve os métodos que herdou (que tenham o modificador *virtual*) para que seu comportamento reflita a nova implementação. Isso permite expandir as funcionalidades de código existente sem haver alteração de tipo.

No momento da invocação do construtor da classe, caso não haja uma invocação explícita de um construtor da superclasse, o compilador adiciona implicitamente uma chamada ao construtor sem argumentos da superclasse. Caso essa versão do construtor não exista um erro de compilação é informado e o programador deve invocar um construtor existente explicitamente.

No Código 2.1 há um exemplo dos elementos de uma classe, de herança, variável escondida e sobrescrita, construtor e invocação do construtor da superclasse.

Listing 2.1: C# code

```
1  class A : B
2  {
3    public A(): base(2){} //invoca construtor da superclasse
4    int campo1;
5    public override int metodo2(){} //metodo sobreescrito
6  }
7
8  class B
9  {
10   public B(){}
11   public B(int a){campo1 = a;}
12   int campo1; //variavel escondida
13   int propriedade1 {get;set;}
14   string campo2;
```

```
15    void metodo1(){}
16    public virtual int methodB2(){}
17  }
```

Como visto previamente uma das maiores preocupações da Orientação a Objetos é o encapsulamento e ocultamento de implementação. Em C# há quatro modificadores base que, quando combinados, permitem seis níveis válidos de acessibilidade, ver Tabela 2.1.

Tabela 2.1: Modificadores de acesso

Access Modifiers	***Accessibility Levels***
Public	Sem Restrição
Protected	Restrito a Classe e Subclasse
Private	Restrito a Classe
Internal	Restrito ao Assembly de Definição
Protected Internal	Restrito ao Assembly de Definição e à Subclasse
Private Protected	Restrito a Classe e Subclasse desde que pertencentes ao Assembly de Definição

2.5 Anomalias de orientação a objetos

Ammann *et al* introduzem em (AMMANN; OFFUTT, 2008) a ideia de anomalias de orientação a objetos como o resultado do mau uso dos recursos de herança e polimorfismo disponíveis ao desenvolvedor em linguagens orientadas a objetos.

Há no total nove anomalias cuja existência é definida como independente de linguagem de programação. Segundo os autores, o que muda de uma linguagem para outra é a forma de aparição das anomalias e suas consequências. A Tabela 2.2 lista todas as anomalias indicando seus acrônimos e significados.

Na Anomalia de Uso Inconsistente de Tipo (ITU) uma subclasse não faz a sobrecarga dos métodos herdados via mecanismo de herança resultando na ausência de comportamento polimórfico. Um objeto de uma classe A que estende uma classe B em um contexto que espera B somente poderá agir como originalmente definido em B, todos os métodos adicionais definidos em A permanecerão ocultos. O comportamento anômalo é originado do uso de uma dessas classes como A em múltiplos contextos intercaladamente (utilizar o objeto de A como B e então logo após utilizá-lo como A e depois ainda utilizá-lo novamente como B). Uma possibilidade é que os novos métodos definidos em A utilizem os métodos ou as variáveis de estado herdados de B resultando em um estado

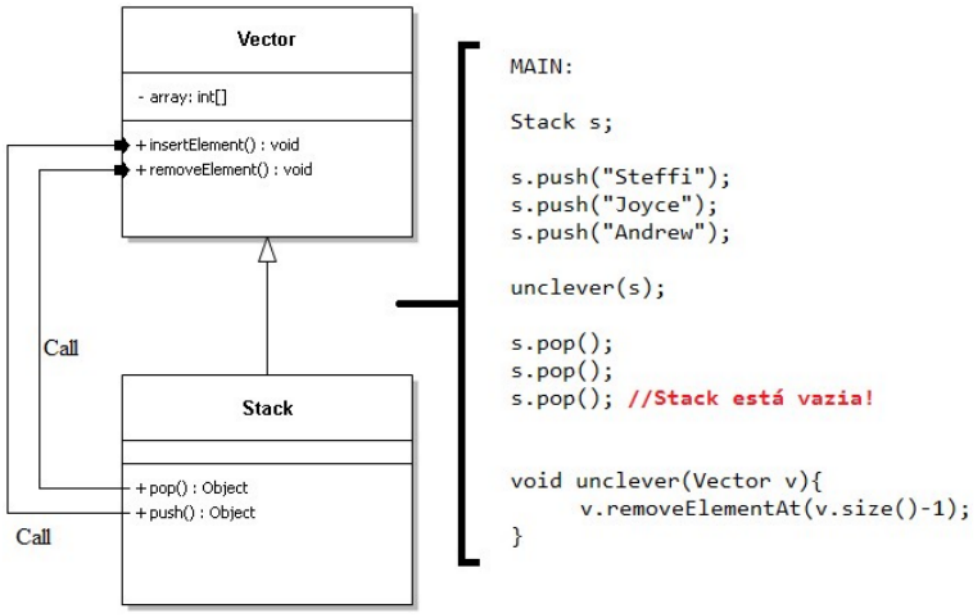
Tabela 2.2: Anomalias em C#

Acronym	***Anomaly***
ITU	Inconsistent type use / Uso inconsistente de tipo
SDA	State definition anomaly / Anomalia de definição de estado
SDIH	State definition inconsistency / Inconsistência de definição de estado
SDI	State defined incorrectly / Estado definido incorretamente
IISD	Indirect inconsistent state definition / Estado indiretamente definido incorretamente
ACB1	Anomalous constructor behavior / Comportamento anômalo de construtor
ACB2	Anomalous constructor behavior / Comportamento anômalo de construtor
IC	Incomplete construction / Construção incompleta
SVA	State visibility anomaly / Anomalia de visibilidade de estado

Fonte: Adaptado de AMMANN & OFFUTT (2008)

anômalo caso a semântica da ação seja incompatível entre a superclasse e subclasse. O exemplo clássico é advindo da classe *Stack* e Vector de JAVA como mostrado na Figura 2.1. *Stack* estende Vector e a implementação de seus métodos utiliza as funções herdadas de manipulação do vetor interno.

Figura 2.1: ITU



Fonte: Jayne Guerra Ceconello (2016)

Três elementos são adicionados à pilha e em seguida há a invocação do método *unclever* que espera um elemento Vector como entrada. Como *Stack* estende Vector o

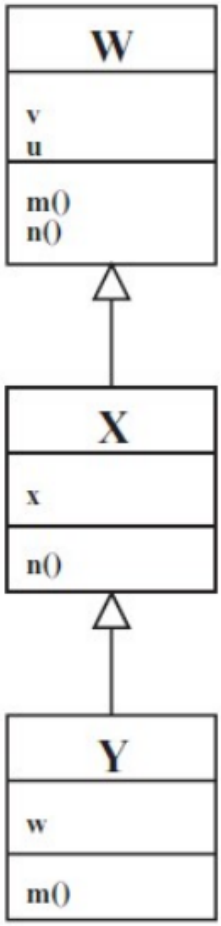
objeto do tipo *Stack* pode ser utilizado em seu lugar. Porém, como a implementação desse método quebra a semântica da classe *Stack*, já que remove um elemento do meio do vetor, uma anomalia ocorre após as invocações do método pop.

Na Anomalia de Definição de Estado (SDA) as interações entre estados da subclasse não são compatíveis com as interações de estado da superclasse. Os métodos sobrecarregados na subclasse, ao serem invocados, devem deixar o estado do objeto de forma coerente respeitando a implementação original dos métodos. Sob o ponto de vista do fluxo de dados, as mesmas definições de estado devem ocorrer na implementação original e na sobrecarga. A Figura 2.2 mostra uma situação em que existe a anomalia no fluxo de dados. O método *m* sobrecarregado na classe *Y* não respeita as definições originais, a variável *v* só tem seu valor definido por *m* quando a versão de *W* é invocada. No caso da chamada sequencial de *m* e *n* o comportamento será anômalo na perspectiva de *n* caso um objeto do tipo *Y* seja usado, uma vez que não haverá a definição de *v* esperada por esse método.

A Inconsistência de Definição de Estado (SDIH) é causada pela declaração de uma nova variável de estado na subclasse. Se a nova variável *v* é introduzida na subclasse e já existe uma variável *v* que foi herdada da superclasse (mesmo nome), a versão herdada será ocultada da cadeia de herança. Todas as referências a *v* a partir daquele nível da hierarquia irão referenciar a versão da subclasse. Isso pode não ser um problema no caso de todos os métodos serem sobrecarregados ocultando de fato todo o uso da variável herdada. Isso, porém, acaba não sendo o cenário comum havendo a existência da anomalia no fluxo de dados conforme o exemplo da Figura 2.2, note que a tabela lista as definições e usos de campos por método. A classe *Y* introduz a mesma variável *v* que herdou de *W*, no caso da chamada sequencial dos métodos *m* e *n* haverá uma anomalia se o objeto de *Y* for utilizado já que *m* irá definir o valor da variável *v* de *Y* mas não de *W* e *n* irá utilizar uma variável não inicializada.

A Anomalia de Estado Definido Incorretamente (SDI) ocorre quando um método sobrecarregado define uma mesma variável de estado *v* que a definição original do método. Caso a computação efetuada no método sobrecarregado seja semanticamente diferente da versão original, a respeito da variável *v*, os estados subsequentes a chamada desse método, seguindo o fluxo de execução na perspectiva de uso da superclasse, podem ser afetados causando uma mudança de comportamento. Não se trata de uma anomalia no fluxo de dados uma vez que há as mesmas definições mas uma potencial anomalia de comportamento.

Figura 2.2: SDA e SDIH



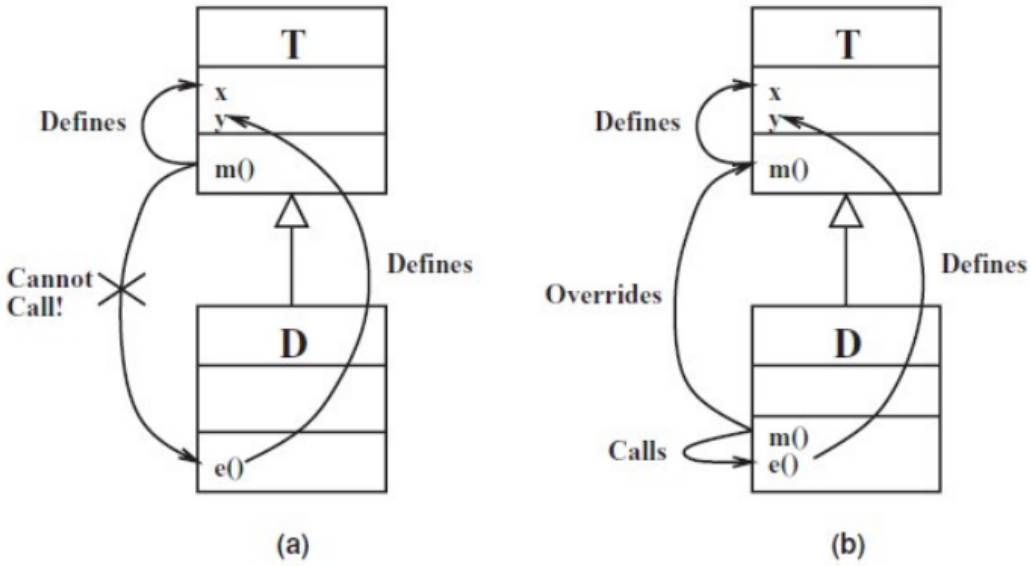
	W::v	*W::u*	*X::x*	*Y::w*
W::l		*def*		
W::m	*def*			
W::n	*use*	*use*		
X::n	*use*		*def*	
Y::l		*def*		
Y::m				*def*

Fonte: AMMANN & OFFUTT (2008)

Na Anomalia de Estado Indiretamente Definido Incorretamente (IISD) um descendente declara um método que define o valor de uma variável de estado herdada. O exemplo visível na Figura 2.3 demonstra a situação. *T* tem duas variáveis de estado *x* e *y* e o método *m* que a define *x*. *D*, que estende *T*, declara o método *e* que modifica o estado de *y*. O método *e* não é visível para a classe *T* já que é declarado em seu descendente. Porém no caso de haver uma sobrecarga de método, nesse exemplo a sobrecarga de *m*, o método sobrecarregado pode fazer a invocação desse novo método e assim introduzir uma anomalia de fluxo de dado no caso de adicionar novas definições que não existiam na implementação original. No exemplo *m* originalmente não modifica *y* porém na implementação de *D m* faz essa definição indiretamente.

O Comportamento Anômalo de Construtor 1 (ACB1) consiste no construtor de

Figura 2.3: IISD



Fonte: AMMANN & OFFUTT (2008)

uma superclasse qualquer conter chamadas a métodos polimórficos. Duas características do construtor são relevantes para essa anomalia:

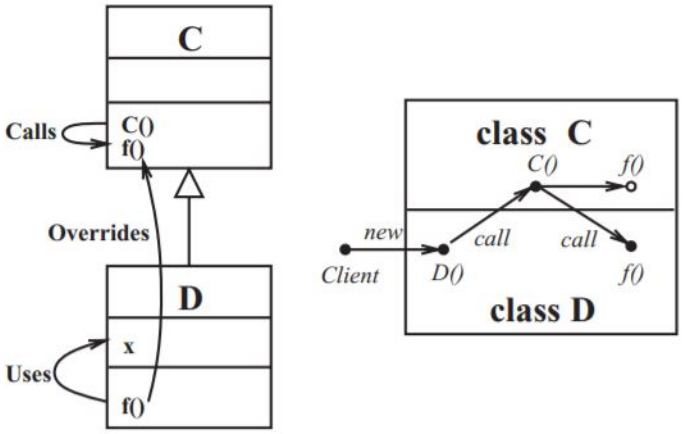
1. Quando múltiplas versões de métodos polimórficos existem, isto é cada membro da hierarquia de herança tem sua própria implementação, a mais próxima da classe invocando o método é chamada.

2. Para construir a instância da subclasse o construtor da superclasse é chamado previamente ao construtor da subclasse.

Seguindo o exemplo da Figura 2.4, a classe *C* tem no seu construtor a invocação ao método estendível *f*. A classe *D*, que estende *C*, sobrecarrega o método *f* redefinindo a sua funcionalidade de modo que utilize a variável de estado exclusiva de *D*, *x*. Na instanciação de um objeto de *C* o construtor de *C* será chamado e a versão de *C* de *f* será invocada. Já na instanciação de um objeto de *D* o construtor de *C* será chamado previamente ao construtor de *D* e nele será invocado a versão de *D* do método *f* que irá usar a variável *x* que pode não estar definida já que o construtor de *D* ainda não foi executado.

No comportamento Anômalo de Construtor 2 (ACB2) o construtor de uma superclasse invoca um método polimórfico que utiliza uma variável de estado herdada. A anomalia ocorre se a variável herdada não é propriamente definida previamente a invo-

Figura 2.4: ACB1



Fonte: AMMANN & OFFUTT (2008)

cação do método no construtor da superclasse. Considere uma classe *C* que contém em sua definição uma variável de estado *x* e que em seu construtor há a invocação do método estendível *f*. Uma classe *D* que estenda *C*, ao sobrecarregar o método *f* com uma implementação que utilize *x*, pode introduzir um erro na construção do objeto ao fazer o construtor de *C* utilizar o valor de *x* previamente a sua inicialização.

A anomalia de Construção Incompleta (IC) ocorre quando o estado inicial de um objeto é indefinido. Os objetos, após a execução do construtor de sua classe, estão em um estado inicial consistente onde cada variável de estado tem seu valor definido explícita ou implicitamente. Nesse cenário há duas formas de uma falha ocorrer:

1. Um valor incorreto foi utilizado para definir o valor inicial de uma variável de estado.

2. A inicialização de uma variável foi ignorada intencionalmente ou não

No item dois existe uma anomalia no fluxo de dados entre o construtor e os métodos que utilizam essa variável previamente a uma definição. Um exemplo dessa anomalia se encontra na Figura 2.5 onde a classe *AbstractFile* tem uma variável *fd* que não é inicializada pelo construtor. A ideia era que uma subclasse fornecesse um valor inicial a essa variável *fd* previamente ao seu uso através da implementação do método *open*. Uma anomalia ocorrerá no caso de um método que use o valor de *fd*, como *read* e *write*, ser chamado previamente a invocação de *open* que define o seu valor inicial.

Na Anomalia de Visibilidade de Estado (SVA) as variáveis de estado em uma classe ancestral *A* são declaradas privadas e existe um método polimórfico *m* que define

Figura 2.5: IC

```
Class abstract AbstractFile
{
  FileHandle fd;

  abstract public open();

  public read() {fd.read ( ... ); }

  public write() {fd.write ( ... ); }

  abstract public close();
}
```

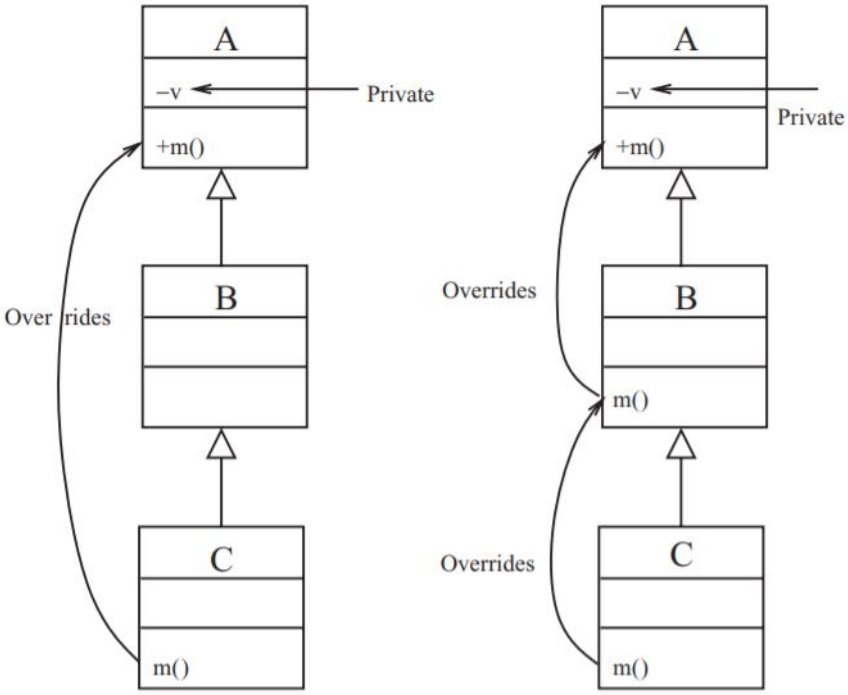
```
Class SocketFile extends AbstractFile
{
  public open()
  {
    fd = new Socket ( ... );
  }

  public close()
  {
    fd.flush();
    fd.close();
  }
}
```

Fonte: AMMANN & OFFUTT (2008)

uma dessas variáveis *v*. Deve existir também uma cadeia de herança como uma classe *B* que estende *A* e uma classe *C* que estende *B* como demonstrado na Figura 2.6. Nesse contexto a classe *C* redefine o método *m* enquanto que *B* não o faz. Como *v* é uma variável privada não é possível que *C*::*m* defina seu estado diretamente sendo o único mecanismo disponível para esse fim a invocação do método *A*::*m*. No caso de *B* também sobrecarregar *m*, *C*::*m* terá de chamar *B*::*m* que por sua vez deverá invocar *A*::*m* para definir o valor de *B*. Nesse cenário *C*::*m* não tem controle direto sobre o fluxo de dados invocado sobre *v* sendo existente a chance de uma anomalia no fluxo de dados dependendo da implementação de *B*::*m*. É observável que a adição da sobrecarga de *m* por *B* pode ter introduzido erros na execução de *C* sem ter alterado nenhuma linha do código de *A* e *C*.

Figura 2.6: SVA



Fonte: AMMANN & OFFUTT (2008)

3 TRABALHOS RELACIONADOS

Os trabalhos que considerei relacionados são os que abordam as anomalias de orientação a objetos ou realizam alguma análise estática no código fonte.

O trabalho de Jayne Guerra (CECONELLO, 2005), "Ferramenta de auxílio à análise de anomalias em códigos Orientados a Objeto", é o mais próximo deste. Ele consiste na implementação de uma ferramenta que realiza uma análise sobre parte das anomalias de Orientação a Objetos em código C++ além de gerar relatórios sobre outros aspectos do código, como grafo YoYo. Ao contrário da abordagem proposta por este trabalho, que utiliza a Árvore Sintática Abstrata (AST) para extrair as informações necessárias para a análise, essa ferramenta utiliza expressões regulares para interpretar programas na linguagem intermediária LLVM-IR que o compilador Clang produz. Um desempenho superior é esperada do nosso analisador na análise das anomalias devido ao uso dos artifícios nativos do compilador como estruturas de dados e métodos. Uma desvantagem da nossa implementação, porém, é que ele se limita apenas a detecção das anomalias, oferecendo pouca visualização para outros aspectos possivelmente problemáticos no código.

A IDE Visual Studio (MICROSOFT, 2019), uma das mais populares para desenvolvimento de C#, contém diversos analisadores embutidos que realizam uma varredura sobre o código fonte utilizando também a API do compilador Roslyn. Não existe nenhuma extensão que busque pelas anomalias diretamente porém tais anomalias podem ser combatidas indiretamente. Algumas práticas ruins, como variáveis escondidas e métodos virtuais no construtor, são desincentivadas por meio de avisos enquanto que refatorações são constantemente sugeridas. Ainda no ambiente dessa IDE pode ser encontrada a extensão comercial ReSharper que realiza uma profunda e complexa análise estática no código que resulta em um detalhado relatório de possíveis problemas no código. Apesar de também não combater diretamente as anomalias, os possíveis problemas originados de sua presença são tratados pelas recomendações de refatoração e suas possíveis origens indicadas na forma de mensagens. Por exemplo, métodos virtuais em construtores são relatados como avisos pela ferramenta na compilação, hierarquias que podem ser convertidas em composição são indicadas e existem avisos sobre a remoção de métodos desnecessários.

Outras ferramentas de análise de código de menor popularidade (SOFTWARE TESTING HELP, 2019):

- PVS-Studio - uma ferramenta para detectar bugs e falhas de segurança em múltiplas linguagem como C, C++, C# e Java (PROGRAM VERIFICATION SYSTEMS,

2019).

- Kritika - ferramenta de análise de código com foco no estilo e nos *bad smells* presentes no código (UPTOSMTH OÜ, 2019).

4 PROPOSTA

Neste capítulo será apresentado o estudo da possibilidade de ocorrência das anomalias de Orientação a Objetos no contexto de C#.

Iniciamos por uma análise teórica onde consideramos os mecanismos disponíveis da linguagem e seu efeito nas anomalias. Listamos os algoritmos utilizados no analisador para realizar as detecções, sendo tais algoritmos propostos pelo autor no contexto deste trabalho.

Descrevemos na seção 4.1 a viabilidade das anomalias na linguagem C#. Em 4.2 aprofundamos a implementação do analisador, abordando sua arquitetura e funcionamento de seus componentes. Em 4.3 expomos os algoritmos utilizados para realizar a detecções de cada uma das anomalias.

4.1 Anomalias em C#

As anomalias de orientação a objetos originalmente definidas por Ammann e Offutt (Tabela 2.2) são descritas como independentes de linguagem e derivadas exclusivamente do mau uso dos recursos do paradigma de orientação a objetos. Contudo o projeto da linguagem, o nível de restrição sob a sintaxe válida e as ações do compilador influenciam na factibilidade das anomalias. Neste capítulo buscamos analisar as anomalias focando no ambiente .Net de C#.

ITU e SDI são anomalias semânticas de difícil detecção via análise estática. SDI depende de como os valores que definem o estado estão sendo alterados durante a execução do método (se a semântica da mudança é compatível). ITU ocorre quando existe herança apenas para reúso de código (sem haver relação semântica de tipo), a detecção é dependente de quão rigoroso é o critério para definir o tipo de herança. Na implementação do analisador consideramos que deve haver ao menos uma sobrecarga para ser esta considerada uma herança de subtipo e não subclasse (as sobrecargas dos métodos herdados de *Object* são filtradas).

IISD, SDA, ACB1, ACB2 e SVA podem ser detectados naturalmente, a linguagem não proporciona qualquer recurso para evitar essas anomalias.

SDIH é uma anomalia de improvável ocorrência uma vez que o compilador detecta o uso de variáveis escondidas (que são o principal requisito) na fase da análise semântica do processo de compilação. Essa detecção é informada ao programador como um *War-*

ning possibilitando ainda que a anomalia ocorra no caso do programador ignorar o aviso do compilador. A linguagem também possibilita o uso da palavra chave *new* para realizar o ocultamento da variável sem gerar o aviso do compilador.

IC não é possível de ocorrer como foi definida por Offut. Todas as variáveis detém um valor por padrão não sendo possível acessar algo indefinido.

4.2 Analisador

O analisador foi projetado como uma aplicação *Console* que recebe como parâmetros de entrada um conjunto de arquivos código fonte C#. Há 2 componentes principais, *CoreParser* e *CoreAnalyser*. O *CoreParser* utiliza os analisadores disponibilizados pelo compilador Rosylin (na forma de biblioteca de C#) para fazer a coleta das informações relevantes, agrupando-as de modo a serem facilmente consumidas pelo *CoreAnalyser*. O *CoreAnalyser* agrega as informações extraídas utilizando a linguagem nativa de pseudo-SQL presente em C#, LINQ, e aplica os algoritmos de detecção das anomalias. Para cada tipo de anomalia detectada um arquivo é gerado como saída. O conteúdo desses arquivos consiste no contexto de detecção de cada uma das anomalias, variando radicalmente pelo tipo, e será detalhado mais adiante.

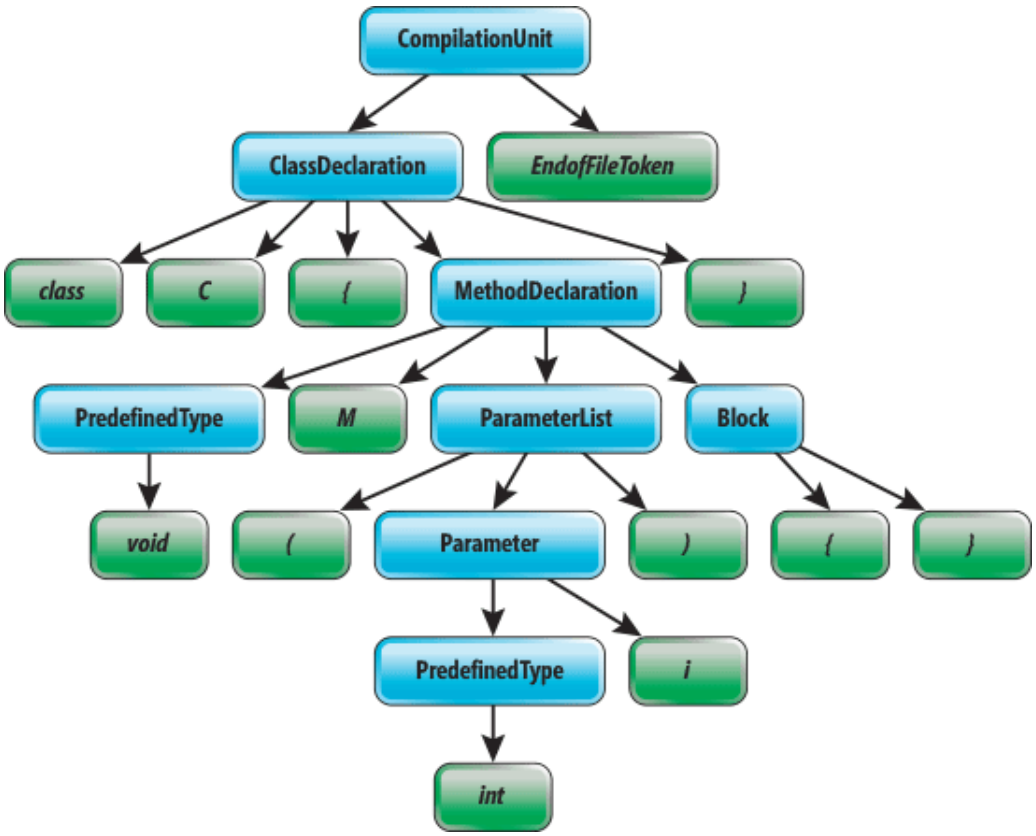
4.2.1 Compilador Roslyn

Junto ao lançamento da versão 4.6 do .Net Framework a Microsoft introduziu o .Net Compiler Platform conhecido como Project Roslyn. Essa plataforma disponibiliza os compiladores das linguagens que compõem o ambiente .Net, como C# e Visual Basic, em forma de APIs.

Além das funções tradicionais de indicação de erros sintáticos e a geração de código, Roslyn permite o uso programático direto das capacidades inerentes do compilador como a análise de código, navegação e manipulação das estruturas sintáticas, como código fonte e árvore de sintaxe abstrata, refatorações e indicações de erros.

Os principais recursos utilizados da plataforma foram as classes que compõem a Árvore de Sintaxe Abstrata (AST) e permitem uma fácil navegação e *pattern matching* para identificar os tipos de nodos e seus atributos. Na Figura 4.1 pode ser visto como Roslyn representa um código fonte C# como AST.

Figura 4.1: Exemplo de AST no Rosylin



Fonte: MSDN Magazine (fevereiro 2015)

4.2.2 Analisador Sintático proposta

O analisador foi desenvolvido para permitir a análise em maior volume de arquivos que uma análise manual permitiria. A arquitetura desse sistema portanto foca em uma interface simplista via terminal que permite a automação da tarefa sem muito esforço.

Seguiu-se o modelo Model View Controller (MVC) na implementação do sistema o que permitiu a este ser extensível ao manter a independência entre a lógica de negócio, o fluxo da aplicação e apresentação dos resultados. O foco foi exclusivamente a linguagem C# porém o modelo seguido permite sua expansão. Os requisitos para detectar as anomalias em uma nova linguagem seriam:

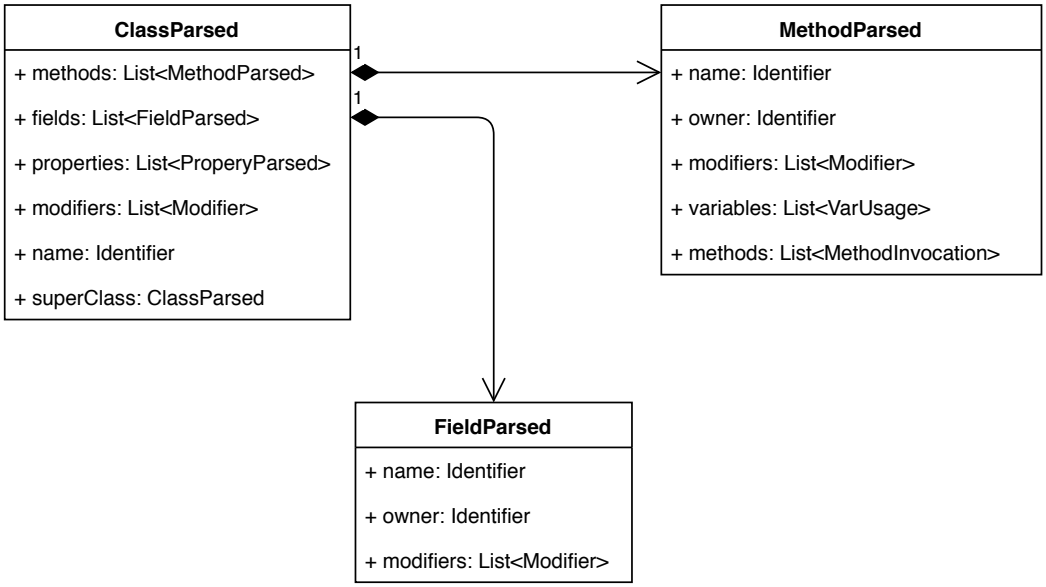
- Modelo de restrição de acesso compatível com C#, palavras reservadas com a mesma semântica.

- Implementação customizada de um componente parser (definido mais abaixo)

4.2.3 Componentes

O sistema é composto por quatro componentes de funções independentes que foram nomeados como *Presenter*, *Core*, *CoreParser* e *CoreAnalyser*. Eles foram planejados da forma a seguir:

Figura 4.2: Parsed Model classes



Fonte: O autor

- *Core* é o coração da aplicação. Ele utiliza o *CoreParser* e o *CoreAnalyser* para proporcionar as funcionalidades do sistema. É o coordenador do fluxo de execução e invocação.

- O *Presenter* tem a função de intermediar a comunicação do Core com agentes externos sendo visto como o componente que concentra os meios de entrada e saída do sistema. Ao abstrair a comunicação podemos alterá-la sem afetar as funcionalidades. A implementação concreta escolhida no analisador foi de uma aplicação console visando o uso da ferramenta de forma automática em um conjunto de repositórios.

- *CoreParser* é o responsável pela extração e filtro das informações relevantes presentes no conteúdo do código fonte sendo analisado. Para cada elemento de uma classe presente no código-fonte existe uma classe específica de mesmo nome com o sufixo

parsed que carrega suas informações de uso, modificadores e hierarquia como visível na figura 4.2. Por exemplo, uma classe pública A com 2 métodos e um campo produziria uma instância de *ClassParsed* que conteria uma propriedade *name* com o valor A, uma propriedade modificadores contendo apenas *public*, uma propriedade que contém os 2 métodos que por sua vez são instância da classe *MethodParsed*, que além das mesmas propriedades citadas previamente ainda conteria os campos utilizados, especificando leitura e/ou escrita, e outros métodos invocados em seu interior, e por fim uma propriedade que conteria o campo que seria uma instância de *FieldParsed*.

Sua implementação concreta utiliza a API do compilador Roslyn para fazer a coleta da árvore de sintaxe abstrata e sua navegação de forma programática. A árvore coletada é percorrida iterativamente por um laço duas vezes. Na primeira iteração, as classes do compilador que definem a AST são utilizadas para realizar um *pattern matching* que permite a fácil identificação dos nodos de acordo com suas funcionalidades e por fim o agrupamento das classes e seus atributos diretos. A segunda iteração é para realizar o preenchimento das informações indiretas que não puderam ser preenchidas na primeira iteração, como classes que ainda não foram tratadas pelo *CoreParser*.

As leituras e definições de variáveis de estado coletadas para cada método consiste em usos diretos, dentro do corpo do método diretamente, e indiretos onde há a invocação de outros métodos da própria classe ou de métodos externos. Um algoritmo de busca por largura foi implementado para percorrer o grafo de invocações de todos as invocações coletadas dentro de cada *MethodParsed*. Isso ocorre no final do processo de *parsing* para manter todos os usos (diretos e indiretos) dentro do objeto.

- O *CoreAnalyser* recebe como entrada os dados do código fonte já estruturados que foram produzidos pelo *CoreParser* e aplica algoritmos específicos para cada uma das anomalias. Sua saída consiste em um conjunto de anomalias que foram detectadas, onde estas são estruturas definidas como uma classe *Anomaly* que contém certas propriedades da classe onde a detecção foi feita com sucesso, como nome da classe e métodos/campos envolvidos. Os algoritmos utilizados serão demonstrados de modo mais abstrato em outra sessão a seguir.

O fluxo de dados é como descrito na Figura 4.3, sendo apresentada abaixo uma descrição textual do mesmo. Note que as instâncias *Anomaly* se refere as as classes *Ano-*

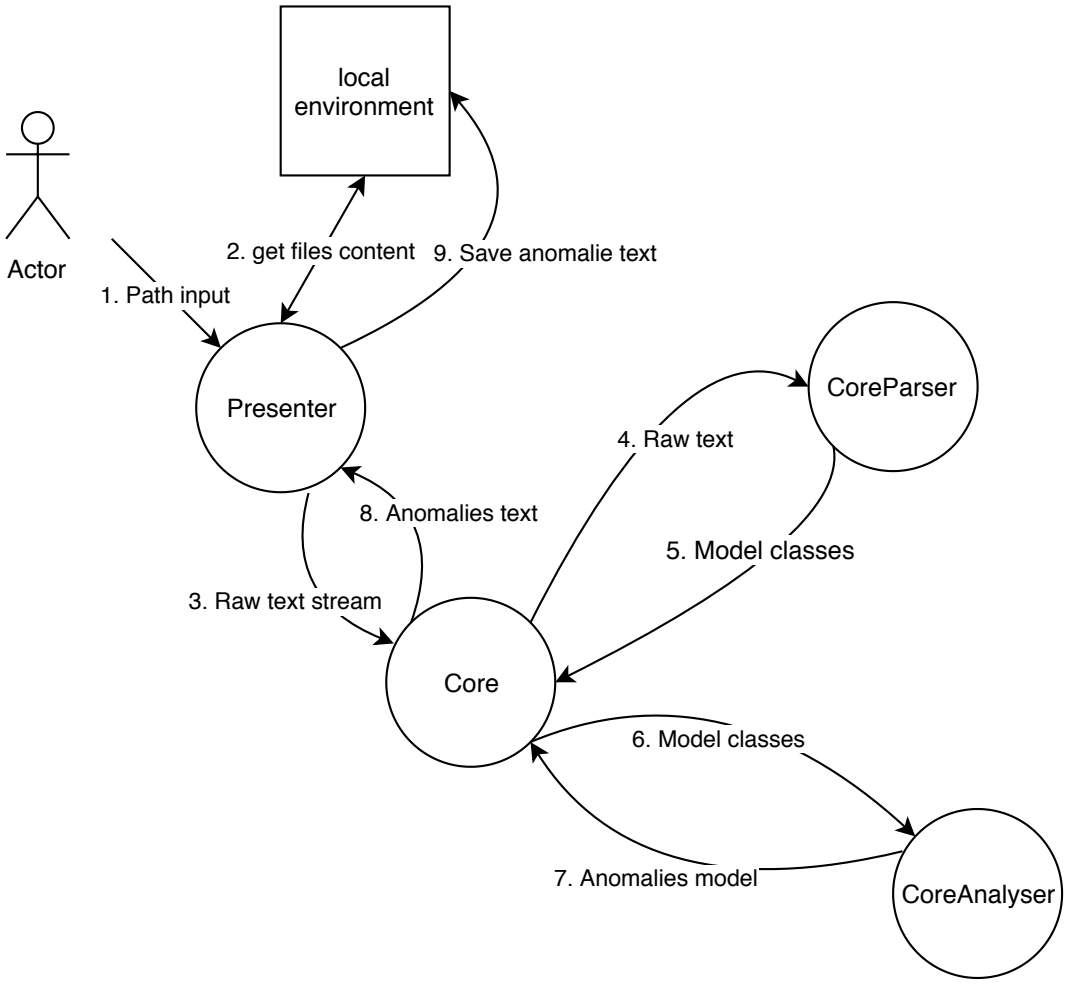
maly nomeadas acima.

1. Um diretório contendo arquivos C# é informado.
2. Ocorre a leitura de todos os arquivos de extensão *.cs*.
3. *Presenter* envia o conteúdo dos arquivos a serem analisados para *Core*
4. *CoreParser* recebe o conteúdo com pré validações básicas de *Core*
5. Core recebe de *CoreParser* um conjunto de instâncias *ClassParsed* devidamente preenchidas
6. *CoreAnalyser* recebe o conjunto de *ClassParsed* de *Core*
7. *Core* recebe de *CoreAnalyser* um conjunto de instâncias *Anomaly*
8. *Core* envia a *Presenter* as instâncias de Anomaly que serão a saída em forma de texto

Algumas considerações e limitações do analisador:

- Apenas são analisadas classes cujo código fonte está presente nos arquivos, no momento que uma classe estende uma classe externa a do projeto esta não será avaliada.
- O analisador não considera o efeito de métodos invocados por código de bibliotecas externas a do projeto. Ex: Um objeto *A* definido no projeto é passado como argumento para um método de uma classe externa.
- Interfaces não foram consideradas já que não foram abordadas por Ammann e Offutt em seu livro. No caso de haver métodos com entradas do tipo interface a versão do método invocado por ela é a da primeira classe encontrada na AST que implementa a interface (limitação imposta pela análise estática).
- Na coleta das definições e uso, no caso de haver a invocação de um método polimórfico a versão utilizada é a da variável fazendo a invocação (outra limitação imposta pela análise estática).

Figura 4.3: Iteração dos componentes



Fonte: O autor

4.3 Detecção das anomalias

Em seu livro Ammann e Offutt apenas definem as anomalias conceitualmente, especificando os requisitos e consequências sem indicar um meio de detecção. Para concretizar a implementação do *CoreParser* um algoritmo para detectar cada anomalia foi desenvolvido. Abaixo estão descritos em uma linguagem um pouco mais abstrata que C# os algoritmos propostos. Para simplificar a apresentação dos algoritmos supõe-se que os nomes de métodos e variáveis são únicos em todas as classes a não ser que seja feita uma sobrecarga. Na implementação concreta dos algoritmos no analisador tal comportamento foi obtido ao usar como prefixo dos nomes a classe que está fazendo a declaração. Todos os algoritmos apresentados abaixo esperam como entrada um conjunto de elementos do tipo *ClassParsed*, tipo abstrato de dado que contém todos os dados relacionados as clas-

ses e seus membros, do código a ser analisado. Todos os algoritmos produzem o contexto relacionado as anomalias (vária de tipo para tipo mas são um conjunto de informações advindas da classe) e que forma a base para a classe de output *Anomaly*.

4.3.1 ITU

Como previamente mencionado, temos que a anomalia ITU é de complexa detecção devido ao caráter semântico. A fim de contornar essa dificuldade definimos que para este analisador uma herança por reúso ocorre apenas quando não há nenhuma sobrescrita de métodos por parte da subclasse (métodos herdados da classe pai de C# object não são contabilizados).

Listing 4.1: C# example

```
bag := Set()

for class c in source_files{

    if c is baseclass{
        continue
    }

    modifiers := Set()

    for method m in c.methods{
        modifiers.union(m.modifiers)
    }

    if modifiers.contains(override){
        bag.add(c)
    }
}
return bag
```

4.3.2 SDI

Esta anomalia consiste em uma subclasse definir o valor de uma variável herdada. Também é de complexa detecção uma vez que a semântica da mudança de estado do ob-

jeto deve ser considerada. Não foi encontrado um método completamente automático e confiável de fazer essa comparação semântica entre a definição original e a versão sobrecarregada sendo necessário que a decisão final seja humana.

Listing 4.2: C# example

```
1
2  anomaly_ok := Set()
3
4  for class c in source_files{
5
6      if c is baseclass{
7          continue
8      }
9
10     base_methods = Set()
11
12     u := c
13     while u is not baseclass{
14
15         u := u.superclass
16
17         for method in u.methods
18             .filter( m => m.modifiers.contains(virtual)){
19                 base_methods.add(method)
20         }
21     }
22
23
24     for method in c.methods
25         .filter(m => m.modifiers.contains(override)){
26
27         base_defs := base_methods
28             .get( method.name )
29             .defs
30
31         for def in method.defs
32             .filter(d => d in base_defs){
33                 anomaly_ok.add(
34                     (c, method.name,
35                     def,
36                     base_defs.get(def))
```

```
                        )
        }
    }

}
return bag
```

4.3.3 IISD

Nesta anomalia buscamos uma subclasse que tenha declarado uma variável oculta e que não tenha sobrecarregado todos os métodos que a utilizem.

Listing 4.3: C# example

```
bag := Set()
methods_by_field := Dictionary()

for class c in source_files{

    // Only add baseclass items on the start
    // so that we don't have current class methods
    // in case of a subclass analysis
    if c is baseclass{
        for method in c.methods{
            for use in method.uses{
                methods_by_field[use.name].add(method)
            }
        }
        continue
    }

    set_super_fields := Set()

    u := c
    while u is not baseclass{

        u := u.superclass

        for field in u.fields{

```

```
27              set_super_fields.add(field)
28          }
29      }
30
31      for field in c.fields.intersect(set_super_fields){
32
33          for m in methods_by_field[field]{
34
35              if not m.modifiers
36                  .contains_any([virtual,override]){
37
38                  bag.add((c,field))
39                  break
40              }
41              else if not c.methods.contains(m)
42              or not c.methods.get(m)
43                  .modifiers.contains(override){
44
45                  bag.add((c,field))
46                  break
47              }
48          }
49      }
50
51      for method in c.methods{
52          for use in method.uses{
53              methods_by_field[use.name].add(method)
54          }
55      }
56
57  }
58
59  return bag
```

4.3.4 SDA

Para detectar SDA basta comparar a definição original dos métodos com a versão sobrecarregada, no caso de ocorrência da anomalia a versão sobrecarregada não tem todas as definições originais.

Listing 4.4: C# example

```
bag := Set()
original_methods := Set()

for class c in source_files{

    if c is baseclass{
        for method in c.methods
            .filter(m => m.modifiers.contains(virtual)){
            original_methods.add(method)
        }
        continue
    }

    for method in c.methods
        .filter(m => m.modifiers.contains(override)){

        original_method := original_methods.get(method)

        if  method.defs.intersect(original_method.defs)
            !=
            original_method.defs {
                bag.add((c, method))
            }

    }
}

return bag
```

4.3.5 ACB1

A detecção desta anomalia consiste basicamente em encontrar métodos que tenham o modificador *virtual* e que sejam usados no construtor. No caso de ocorrência da anomalia a versão sobrecarregada do método contém algum uso de variáveis de estado declarados na subclasse.

Listing 4.5: C# example

```
bag := Set()
```

```
2
3  for class c in source_files{
4
5      if c is baseclass{
6          continue
7      }
8
9      virtual_methods := Set()
10
11     u := c
12     while u is not baseclass{
13
14         u := u.superclass
15
16         for constructor c in u.constructors{
17             for method m in c.calls
18                 .filter(m => m.modifiers
19                     .contains(virtual)){
20
21                     virtual_methods.add(m)
22             }
23         }
24     }
25
26     for method m in c.methods
27         .filter(m => m.modifiers.contains(override)){
28
29         // override method and method reads variable
30         //declared on current class
31         if virtual_methods.contains(m)
32         and m.reads.filter(r => r.owner == current).any(){
33             bag.add( (c,m) )
34         }
35
36     }
37 }
38
39 return bag
```

4.3.6 ACB2

Sendo similar a ACB1, ACB2 também consiste em chamadas a métodos polimórficos dentro do construtor da superclasse. O que a difere é que a anomalia depende do uso de variáveis herdadas que não são antes da invocação do método. A ordem das expressões dentro do construtor é relevante nesta anomalia.

Listing 4.6: C# example

```
1
2  bag := Set()
3  missing_dec_by_method_and_class := Dict()
4
5  for class c in source_files{
6
7      if c is not baseclass {
8
9          for m in c.methods
10             .filter(m => m.modifiers
11                 .contains(override)){
12
13             for s in m.uses
14                 .filter(u => u.owner == inherited) {
15
16                 u := c
17                 while u is not baseclass{
18
19                     u := u.superclass
20
21                     if missing_dec_by_method_and_class
22                         .contains( (u,m) ){
23
24                         if missing_dec_by_method_and_class[(u,m)]
25                             .intersect(s).any(){
26                                 bag.add((c,u,s))
27                         }
28                     }
29                 }
30             }
31         }
32     }
```

```
33
34      f := c.fields
35
36      for constructor in c.constructors {
37
38          for e in constructor.exp{
39
40              if e is def{
41                  f.remove(e)
42              }
43              else if e is call and e.modifiers.contains(virtual){
44                  missing_dec_by_method_and_class[(c,e)] := f
45              }
46
47          }
48      }
49
50  }
51
52  return bag
```

4.3.7 IC

A detecção de IC consiste em encontrar métodos que utilizem variáveis não inicializadas no construtor.

Listing 4.7: C# example

```
1
2   bag := Set()
3   missing_dec_by_ctr := Dict()
4
5   for class c in source_files{
6
7       for ct in c.constructors {
8
9           f := c.fields
10
11          for e in ct.exps{
12
```

```
13              if e is def{
14                  f.remove(e)
15              }
16          }
17
18          missing_dec_by_ctr[ct] := f
19      }
20
21      for m in c.methods{
22
23          for ct in missing_dec_by_ctr.keys{
24
25              if m.uses
26                  .intersect(missing_dec_by_ctr[ct]).any(){
27                      bag.add( (c, ct, m) )
28              }
29          }
30      }
31
32  }
33
34  return bag
```

4.3.8 SVA

A detecção de SVA consiste em encontrar um encontrar um método de posição intermediária na hierarquia de herança que possa afetar a definição de uma variável privada nas subclasses.

Listing 4.8: C# example

```
1  bag := Set()
2  variables_by_method := Dict()
3
4  for class c in source_files{
5
6      for method m in c.methods
7          .where(m => m.modifiers.contains('virtual')){
8
9          variables_by_method.add(m, m.defs
```

```
10              .where(f => f.modifiers.contains('private')))
11
12      }
13
14      if(c is baseclass or c.superclass is baseclass){
15          continue
16      }
17
18      for method m in c.methods
19          .where(m => m.modifiers.contains('override')){
20
21          if not variables_by_method[m].any(){
22              continue
23          }
24
25          u := c.superclass
26          while u is not baseclass{
27
28              if u.methods.contains(m) and
29                  u.methods.get(m).modifiers
30                      .contains('override'){
31                  bag.add((u,c, m))
32              }
33
34              u := u.superclass
35          }
36
37      }
38  }
39
40  return bag
```

5 EXPERIMENTOS

Neste capitulo será descrito como analisador desenvolvido foi utilizado e os resultados por ele obtidos. O código fonte está disponível <https://github.com/weisskaiser/anomaly-analyzer>. Na seção 5.1 as entradas utilizadas são descritas. Em 5.2 estará o resultado das execuções.

5.1 Entradas

Para visualizar o impacto das anomalias o analisador desenvolvido foi utilizado em 5 repositórios públicos aleatórios disponíveis no portal GitHub. Algumas considerações sobre o resultado da seleção dos repositórios:

- Não houve repetição de tipo na escolha. Ex: Não foram selecionadas 2 bibliotecas.
- Repositórios sem herança não foram utilizados.
- O nível de profissionalismo dos programadores difere entre os projetos escolhidos.

Detalhes do conteúdo dos repositórios se encontram na Tabela 5.1. Mais informações sobre cada repositório podem ser encontradas ao acessar github e realizar a busca pelo projeto.

Segue uma breve descrição de cada um desses 5 repositórios:

1. **flappy_bird** e **sharekhan** são repositórios de nível mais amador, com poucos contribuidores e código mais simples;
2. **flappy_bird** consiste em 1 jogo de Unity, onde métodos específicos são invocados pela *engine* do jogo. Portanto, o papel dos construtores de classe é reduzido;
3. **testfx** é um dos frameworks de teste disponibilizados pela própria Microsoft;
4. **testfx** e **MiniProfiler/dotnet** são repositórios de nível profissional, no qual revisões de código e altos padrões de desenvolvimento são um requerimento para contribuições;
5. **petstore** consiste em uma amostra de sistema web voltado para demonstração de boas práticas e *Domain Driven Development*. Este projeto visa ser um exemplo de profissionalismo.

Tabela 5.1: Repositórios analisados

Repositórios	Classes	Métodos	Propriedades	Subclasses
flappy_bird	8	20	0	8
sharekhan	66	293	45	20
petstore_ddd_csharp	68	166	117	11
MiniProfiler/dotnet	223	918	336	90
testfx	443	2290	643	132

Fonte: O autor

5.2 Resultados

Apesar do protótipo informar detalhes da ocorrência de cada anomalia, por exemplo fatores como métodos e campos envolvidos, para essa coleta de resultados apenas foi considerado se a anomalia ocorre em uma classe ou não. O resultado das execuções se encontra na Tabela 5.2, estando visível no gráfico mostrado na Figura 5.1.

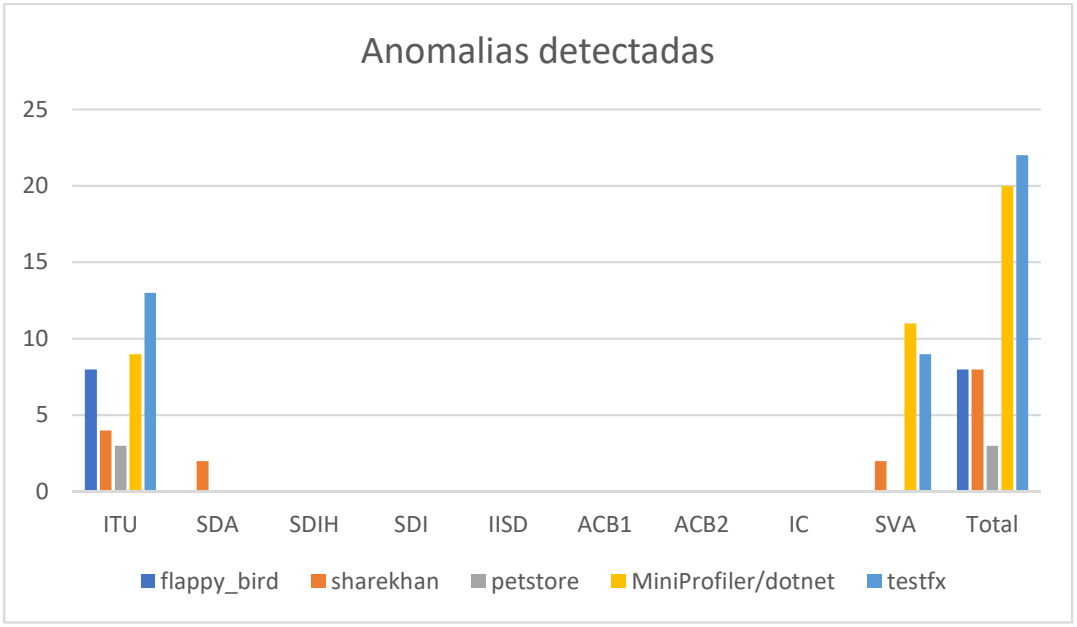
Tabela 5.2: Resultados obtidos

Repositórios	ITU	SDA	SDIH	SDI	IISD	ACB1	ACB2	SVA
flappy_bird	8	0	0	0	0	0	0	0
sharekhan	4	2	0	0	0	0	0	2
petstore	3	0	0	0	0	0	0	0
MiniProfiler/dotnet	9	0	0	0	0	0	0	11
testfx	13	0	0	0	0	0	0	9

Fonte: O autor

É visivelmente baixo o número de anomalias detectadas. As poucas ocorrências de ITU e SVA são reflexo do desenvolvedor tentando utilizar os mecanismos da Orientação a Objetos, como herança, polimorfismo e encapsulamento, de modo que lhe é conveniente. As ocorrências de ITU se resumem a reúso de métodos e campos, principalmente de *Data Transfer Objects* (DTO) onde se busca normalizar campos em comum e classes de teste que utilizam de inicializações semelhantes. As de SVA são consequência do encapsulamento de campos preventivo, só não houve mais detecções desse tipo devido a herança de 3 níveis ou mais não ser comum. Por fim as ocorrências de SDA não tiveram consequência no uso, foram variáveis cuja definição não ocorreu no método sobrecarregado e que acabaram não sendo utilizadas .

Figura 5.1: Resultado das execuções nos repositórios



Fonte: O autor

5.3 Considerações finais

Neste capítulo foi apresentado o experimento realizado e os resultados obtidos. Uma conclusão prática após ter observado os resultados e estudado as anomalias é que estas não apresentam perigo suficiente para exigir planejamento e técnicas de prevenção específicas uma vez que as suas aparições são raras. Os requisitos para a aparição das anomalias são difíceis de serem atingidos e ao seguir boas práticas básicas de programação a chance de sua ocorrência é baixíssima.

Podemos destacar as seguintes boas práticas de programação como exemplo de prevenção contra as anomalias:

- Composição sobre herança - todas as anomalias dependem de herança para acontecerem, menos herança acarreta em menos anomalias. Esse princípio enfatiza que a herança é um mecanismo de polimorfismo e não de reúso.

- Princípio de substituição de Liskov - elimina ITU, SDA e SDI ao tornar herança um mecanismo que zele por polimorfismo e consistência entre as subclasses e superclasse. Se implementado em nível de linguagem permite evitar diversas das situações apontadas como anomalias.

- Evitar variáveis escondidas - variáveis escondidas ofuscam o fluxo de dados da classe, sempre há formas alternativas a sua inclusão, o próprio compilador dispõe de avisos contra seu uso.

- Evitar incertezas em construtores - qualquer chamada que traga incerteza deve ser evitada na construção de objetos. Métodos polimórficos e dependentes de recursos externos podem adicionar comportamento inesperado pelo invocador, a construção de objetos deve ser *barata* e livre de falhas externas (exceção de rede por exemplo).

6 CONCLUSÃO

Este trabalho buscou analisar as anomalias de orientação a objetos no contexto da linguagem de programação .Net C# a fim de identificar se estas causam erros o suficiente para receber atenção especial dos desenvolvedores e testadores. Na literatura pouca atenção é direciona ao tópico, havendo basicamente apenas a definição original de Ammann Offutt como referência.

Iniciou-se o trabalho pela análise da viabilidade de cada uma das anomalias no ambiente .Net. O estudo foi realizado de modo superficial para esclarecer parte do que seria esperado dos resultados e direcionar o esforço de implementação. Um analisador estático de código fonte C# foi desenvolvido em C# utilizando a API do compilador Roslyn com o objetivo de analisar um maior volume de classes. Algoritmos que realizam a detecção de cada uma das anomalias para o analisador foram propostos devido a inexistência dos mesmos.

A aplicação foi exercitada utilizando repositórios de código aberto reais de diferentes complexidades, volumes de código e experiência dos desenvolvedores. Os resultados obtidos fomentam a ideia que as anomalias são desconhecidas devido a sua falta de impacto no desenvolvimento. Houve pouquíssimas ocorrências e mesmo estas não representaram erro. As boas práticas de programação já populares combatem e previnem a existência das anomalias não sendo necessário um escopo de teste dedicado a sua captura.

6.1 Trabalhos futuros

A análise exclusivamente estática de código limita o poder de captura principalmente nas anomalias de teor mais semântico. A adição de inteligência artificial para uma melhor captura desse subconjunto das anomalias poderia melhor a precisão da análise.

Este trabalho tratou exclusivamente de C# porém o ambiente .Net é composto de mais linguagens. F# é uma delas porém diferentemente de C# ela se segue o paradigma funcional de programação em conjunto com a orientação a objetos. Poderia ser comparado o impacto do paradigma na ocorrência das anomalias.

REFERÊNCIAS

AMMANN, P.; OFFUTT, J. **INTRODUCTION TO SOFTWARE TESTING**. [S.l.]: Cambridge University Press, New York, 2008.

BOOCH, G. **Object oriented design: with applications**. [S.l.]: Redwood City, Calif.; Wokingham: Benjamin/Cummings, 1991.

CECONELLO, J. G. **Ferramenta de auxílio à análise de anomalias em códigos orientados a objeto**. Monografia (TCC) — UFRGS, 2005.

DRAYTON, P.; ALBAHARI, B.; NEWARD, T. **C in a nutshell:**. [S.l.]: OReilly, 2003.

FRISCHKNECHT, C. **The 2018 Software Fail Watch Awards**. 2018. Available from Internet: <https://www.tricentis.com/blog/software-fail-awards-2018/>.

HOWDEN, W. Functional program testing. **The IEEE Computer Societys Second International Computer Software and Applications Conference, 1978. COMPSAC 78.**, 1978.

IDG. **Digital Business Survey**. 2018. <https://www.idg.com/tools-for-marketers/2018-digital-business-research/>, acesso em: 01/09/2019.

ISO/IEC 23270:2003. 2006. <https://www.iso.org/standard/36768.html>, acesso em: 25/11/2019.

IVANOV et al. **Adoption of Robots, Artificial Intelligence and Service Automation by Travel, Tourism and Hospitality Companies – A Cost-Benefit Analysis**. 2017. Available from Internet: <https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3007577>.

JORGENSEN, P. **Software testing: a craftsmans approach**. [S.l.]: CRC Press, Taylor & Francis Group, 2014.

MEYER, B. **TOUCH OF CLASS: learning to program well with objects and contracts**. [S.l.]: SPRINGER-VERLAG BERLIN AN, 2016.

MICROSOFT. **Visual Studio: Overview of source code analyzers**. 2019. <https://docs.microsoft.com/pt-br/visualstudio/code-quality/roslyn-analyzers-overview?view=vs-2019>, acesso em: 17/12/2019.

MYERS; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2012.

NATO. **The 1968/69 NATO Software Engineering Reports**. 1968. <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html>, acesso em: 10/08/2019.

PROGRAM VERIFICATION SYSTEMS. **PVS-Studio**. 2019. <https://www.viva64.com/en/pvs-studio/>, acesso em: 18/12/2019.

RUMBAUGH, J. B. M. **Object - oriented: modeling and design**. [S.l.]: PRENTICE HALL, 1998.

SOFTWARE TESTING HELP. **TOP 40 Static Code Analysis Tools**. 2019. <https://www.softwaretestinghelp.com/tools/top-40-static-code-analysis-tools/>, acesso em: 17/12/2019.

TECHNOLOGIESCIGNITI, C.; SERVICES, I. Q. E. . S. T. **Rise of the Software Development Engineer in Test: Software Testing**. 2018. Available from Internet: <https://www.cigniti.com/blog/rise-of-the-software-development-engineer-in-test-sdet/>.

UPTOSMTH OÜ. **Kritika**. 2019. <https://kritika.io/>, acesso em: 18/12/2019.