Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Compreendendo Programas por meio de *Design By Contract*: um estudo com desenvolvedores

Normando Gomes de Carvalho Júnior

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Tiago Massoni
(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

C331c Carvalho Júnior, Normando Gomes de.
Compreendendo programas por meio de *Design by Contract:* um estudo com desenvolvedores / Normando Gomes de Carvalho Júnior. ¯ Campina Grande, 2017.
94f. : il. color.

Dissertação (Mestrado em Ciência da Computação) ¯ Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2017.
"Orientação: Tiago Lima Massoni, Dr.".

1. Compreensão de Programas. 2. Design by Contract. 3. Contratos. I. Massoni, Tiago Lima. II. Universidade Federal de Campina Grande, Campina Grande (PB). III. Título.

CDU 004.41(043)

Resumo

Compreender programas é difícil porque cada trecho de código atende requisitos específicos. Em alguns casos, fatores como compreender programas que foram escritos por outras pessoas, o escopo limitado das ferramentas existentes, e a falta de documentação adicionam complexidade. Sendo assim, os desenvolvedores necessitam de uma abordagem de compreensão efetiva que diminua os custos na manutenção e que reduzam os riscos de erros, caso o entendimento do programa seja incompleto. Existem abordagens sistemáticas, apoiadas por ferramentas, para compreensão que utilizam verificações estáticas (análise do código fonte) ou dinâmicas (dados sobre a execução). As abordagens dinâmicas são utilizadas por sua efetividade, pois basta executar um teste para ver o resultado, no entanto, falham por não utilizar informações de alto nível sobre o comportamento que possam ser verificadas. Todavia, estas informações podem ser capturadas ao definir contratos, como por exemplo, na metodologia *Design by Contract*. Contudo, não existe uma abordagem sistemática de compreensão que utilize contratos. Este trabalho, propõe um estudo para compreender programas apoiada por uma abordagem sistematizada a fim de priorizar a escrita de contratos em programas *C#* utilizando os princípios de *Design by Contract* por meio da biblioteca *Code Contracts*. Podendo, mesmo assim, ser utilizada em qualquer linguagem de programação que dê suporte a *Design by Contract*. A avaliação foi feita em ambientes de desenvolvimento de software com 12 desenvolvedores de um centro de pesquisa e desenvolvimento em ciência da computação, considerando a aplicação da abordagem a três métodos (rotinas) de um projeto *open-source*. Os resultados do estudo apontam indícios de melhor compreensão dos métodos usando contratos, e por outro lado, de favorecimento da própria escrita dos contratos em métodos, a princípio desconhecidos pelos desenvolvedores.

Abstract

Program comprehension is generally a difficult task, since each part of code meets specific requirements. In some cases, factors - such as comprehend programs that were written by others, limited scope of existing tools, and lack of documentation - add complexity. Therefore, developers need an effective program comprehension approach that reduces maintenance costs and the risk of errors whether the program comprehension is incomplete. In order to mitigate that problem, systematic approaches are used, supported by tools, to verify comprehension using static (source code analysis) or dynamic (data on execution). Dynamic approaches are used because of their effectiveness, since they simply run a test to see the result, however, they fail to use high-level behavioral information that can be verified. Still, this information can be captured while defining contracts, for example in the Design by Contract methodology. Nevertheless, for the best of our knowledge, there is no systematic approach to program comprehension that uses contracts. In this work, we propose a study to program comprehension supported by a systematized approach, in order to prioritize the writing of contracts in C# programs using the principles of Design by Contract through the Code Contracts library. It can still be used in any programming language that supports Design by Contract. We evaluated it in software development environments with 12 developers of a research and development center in computer science, considering the approach application to three methods (routines) of an open-source project. The results of our study indicate better comprehension of methods using contracts, and, on the other hand, favoring the writing of contracts in methods, initially unknown to developers.

Agradecimentos

A Deus. Sua graça me ilumina e guia nessa passagem tão rápida que é a vida.

A Priscila, por seu apoio, dedicação e paciência. Sem você tudo seria mais difícil. Te amo todos os dias, cada vez mais.

A minha mãe por ter proporcionado o melhor nas suas condições. A minha irmã pelo apoio e conversas. A todos os meus entes, em especial meu tio que partiu durante esta caminhada. Aos meus sogros. Vocês são importantes para mim.

A Tiago, por ter me orientado durante esses anos e por ser um ótimo mentor, sempre disposto a me ouvir. Tive o privilégio de aprender a cada conversa que tivemos.

A Rohit pelos ensinamentos. Sua dedicação comigo foi extremamente valiosa para que pudesse aprender tanto em tão pouco tempo. Muito obrigado.

Ao professor Hyggo Almeida pelo apoio durante esta jornada. Agradeço também ao professor Jacques Sauvé pelos ensinamentos, e aos funcionários da COPIN.

Aos amigos que fiz na COPIN, no Embedded e no SPLab. As conversas foram proveitosas.

Aos membros da banca da proposta: Adalberto Cajueiro e João Arthur.

Aos participantes do estudo que disponibilizaram tempo para concretização desta pesquisa.

A todos meus amigos, pelos momentos divertidos e pelas cervejas tomadas para descontrair.

Por fim, ao CNPq pelo apoio financeiro.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Grandes sistemas de *software* tendem a ser desenvolvidos por uma ou mais equipes de desenvolvedores. Estes, por sua vez, possuem diferentes níveis de experiência, o que afeta a forma com que as soluções são implementadas. Além disso, a heterogeneidade de conhecimento e a rotatividade entre os membros de um time de desenvolvimento podem dificultar posteriormente na compreensão e manutenção do código. Diversos fatores podem estar relacionados a essas dificuldades, dentre eles, a inutilização de técnicas que auxiliem na compreensão de códigos, o escopo limitado das ferramentas existentes, a falta de documentação, a falta de testes e a ausência de uma abordagem utilizando formalismo em tempo de execução [27; 3].

As técnicas para auxiliar nos processos de compreensão de código são relevantes para o desenvolvimento e manutenção de *software*. Embora muitas estejam relacionadas a fatores distintos, o processo cognitivo dos desenvolvedores com os programas que estão sendo desenvolvidos e mantidos merece destaque [47]. Carma McClure [33] aponta que a fase de manutenção é uma das mais custosas no ciclo de vida de desenvolvimento de *software*. É comum desenvolvedores dedicarem parte do tempo de seus trabalhos lendo códigos-fonte, isto porque para evoluir um sistema existente ou para integrar novas funcionalidades no programa, os desenvolvedores precisam entendê-lo corretamente [48]. Nesse sentido, dependendo do nível de especificidade de uma atividade, não basta apenas a leitura desses códigos, é preciso contar também com outros artefatos, a exemplo da documentação.

Sendo assim, documentar o *software* é primordial para manter a qualidade e funcionamento do sistema. Isto porque *softwares* tendem a evoluir rapidamente em projetos de médio

e grande porte. Além disso, atualmente, os *softwares* estão se tornando cada vez mais complexos devido a sua distribuição, algoritmos, plataformas, metodologias e tecnologias [34]. No entanto, muitos desenvolvedores dedicam poucos esforços a documentação devido ser tedioso e custoso. Timothy Lethbridge et al. [27], em seu estudo, por exemplo, relatam que a atualização da documentação muitas vezes é esquecida. Em alguns casos, a falta de esforço para documentar o sistema ocorre devido à influência do tempo ou o surgimento de uma demanda prioritária de novas atividades. Sendo assim, a documentação passa a perder a sua efetividade quanto a manutenção do sistema, tornando-se obsoleta, de má qualidade e desatualizada, chegando ao ponto de ser descartada [16].

Embora a documentação de programas leve em consideração a arquitetura, os requisitos funcionais e não funcionais, as tecnologias, as linguagens e as múltiplas plataformas, os desenvolvedores na sua grande maioria ignoram esses documentos. Existem casos em que a documentação segue aparentemente atualizada, porém a implementação não condiz com a esta. Tobias Roehm et al. [48] ressaltam que desenvolvedores gastam a maior parte do tempo realizando a leitura de códigos-fonte. Assim, de modo a apoiar o entendimento, alguns tendem a escrever comentários nos trechos de código [54; 55]. Nesses casos, o objetivo é documentar o que foi implementado a fim de auxiliar no processo de compreensão tanto de desenvolvedores antigos como para novos membros da equipe. Contudo, apesar de ser uma atividade relativamente simples, muitos esquecem de fazê-la.

Outro ponto que deve ser levado em consideração e de grande importância para que um *software* funcione como especificado são os testes de *software*. Testes de *software* são utilizados para verificar e validar funcionalidades de programas. Existem diversas formas de se testar um programa, a exemplo, testes unitários, testes exploratórios e testes de regressão [19]. Dentre estes, os testes de unidade apresentam grande relevância para os desenvolvedores no momento da construção do *software*. Isso, porque quando aliados a documentação, reduzem a probabilidade do sistema conter implementações diferentes das especificadas. Sem contar que rodar testes são uma boa forma de adquirir conhecimento sobre o código.

Nas últimas décadas, um grande número de pesquisadores têm dedicado seus esforços em testes de *software*. Frederick Brooks [18] em sua pesquisa, relata que desenvolvedores gastam uma grande parte do tempo testando. Porém, o propósito com o qual esses testes estão sendo realizados é questionável, porque é comum um grande número de erros se faze-

rem presentes mesmo com um sistema documentado, levando a fortes indícios de que não existem testes. Em uma pesquisa recente, Moritz Beller et al. [3] apontam que apenas 9% do tempo dos desenvolvedores é dedicado a fase de testes em seus ambientes integrados de desenvolvimento, fortalecendo a ideia de que desenvolvedores continuam a maior parte do tempo lendo códigos e implementando/mantendo funcionalidades.

Diante dessas situações, o principal problema que pode estar atrelado a essas ocorrências é o fato das abordagens existentes possuírem um escopo limitado. Como efeito disso, e com a escassez da documentação, muitos problemas surgem. Assim, uma forma para contorná-los é fazer o uso de ferramentas que auxiliem na compreensão de programas.

1.1 Problema

Ferramentas tendem a facilitar e automatizar as estratégias de compreensão. Ao logo do tempo, diversas foram propostas [44; 62; 4; 40; 56]. Entretanto, por apresentarem uma gama de técnicas, muitas delas são limitadas à experiência, aos tipos de atividades, à familiaridade com o programa, às tecnologias utilizadas e ao interesse dos desenvolvedores [57]. Atualmente, as técnicas se limitam a auxiliar os desenvolvedores apenas no processo de compreensão de uma funcionalidade do sistema [26]. Um exemplo desses fatos são as ferramentas que auxiliam na busca de itens de interesse no código-fonte e na navegação das relações que são encontradas no código [20; 62]. A tendência é que esse cenário se modifique, visto que pesquisas vêm contribuindo nessa linha. Por exemplo, Tobias Roehm et al. [48] realizaram um estudo observacional em diferentes empresas de *software*, investigando como os desenvolvedores compreendem programas. Em particular, o estudo observou as estratégias adotadas, as informações necessárias e as ferramentas utilizadas. Dentre as estratégias, as que merecem destaque são: a depuração de código fonte, a clonagem do código como forma de implementar e manter funcionalidades e as situações em que os desenvolvedores se colocam no papel de usuários finais.

Yida Tao et al. [60] realizaram um estudo empírico investigando o papel das mudanças no código na prática industrial. Eles perceberam que a lógica da mudança é a informação que apresenta maior relevância para efetuar uma alteração. Já outros trabalhos [7; 31; 65] apresentam técnicas automatizadas. Essas técnicas incluem a análise das mudanças em tipos

distintos de artefatos de *software*, a identificação de diferenças estruturais juntamente com as diferenças textuais dentro de ambientes de desenvolvimento integrado e a geração automática de mensagens com base em *commits* como forma de documentar o código.

Diante disso, poucas ferramentas para compreender programas são utilizadas na prática. Nesses casos, o que se percebe é a adoção de metodologias variadas ao processo de desenvolvimento de *software*. Essa adoção muitas vezes pode confundir os desenvolvedores. Isso porque existem dois tipos de desenvolvedores: os experientes e os novatos. Os experientes são aqueles que conhecem mais do sistema e que estão presentes na fase do desenvolvimento desde o início, enquanto os novatos, pouco sabem das funcionalidades e implementações do sistema. A principal diferença entre os dois é que os novatos, na maior parte dos casos, não sabem por onde começar e acabam gastando mais tempo procurando entender o código [49].

Existem abordagens, a exemplo do *Design by Contract*, que estabelecem contratos para as rotinas do programa. Os contratos, quando presentes no código, representam uma forma de documentação mais formal e detalhada. Por outro lado, as rotinas podem auxiliar na compreensão de trechos do programa, no momento que definem o que o método deve fazer. Nesse sentido, o entendimento desses trechos se torna mais fácil. Sergio Areias et al. [1] introduziram noções para fatiar programas utilizando os princípios de *Design by Contract*. A motivação era garantir o correto comportamento do programa, facilitando a reutilização de componentes já verificados, evitando a reconstrução de novos conceitos.

Assim, três pontos importantes podem ser observados. O primeiro deles é que não existe uma abordagem que faça a relação do uso entre *Design by Contract* e testes para compreender programas. O segundo é que as ferramentas de compreensão de código tendem a auxiliar os desenvolvedores apenas no processo de compreensão de uma funcionalidade do sistema. Por último, porém não menos importante, seria integrar as ferramentas de compreensão ao processo de desenvolvimento.

1.2 Exemplo Motivante

Nesta seção, um exemplo de código em *C#* é apresentado de modo a enfatizar a dificuldade de analisar programas. Por exemplo, o trecho do Código fonte 1 é parte do código fonte de um projeto real, desenvolvido em *C#*. Na Linha 10 pode-se observar o método *GetAp-*

pListByCategoryRange(...), responsável por retornar uma lista de aplicativos cadastrados.

Caso seja necessário implementar um novo método que retorne aplicativos contendo outras características, não basta apenas o desenvolvedor checar se algum método relacionado existe, é preciso ter conhecimento do que é necessário para realizar a atividade. Interpretar a princípio quais são as condições definidas em um método é difícil. Entretanto, códigos menores são mais fáceis de serem compreendidos.

Analisando o Código fonte 1 é possível visualizar na Linha 25 uma chamada para o método *Get(...)* da classe *Application*. Esse método requer que cinco argumentos sejam passados. Porém, não se sabe quais restrições esses possuem e nem como esses são utilizados. Nesses casos, é necessário o desenvolvedor checar detalhadamente o método correspondente de modo a evitar uma compreensão equivocada. O Código fonte 2 descreve as instruções presentes no método *Get(...)*.

A princípio basta fazer uma leitura superficial para perceber as chamadas e checagens no escopo do método *Get(...)*. Note que o código contêm regras de domínio específicas, como por exemplo, as presentes nas Linhas 3, 4, 7, 9, 11 e 14. Na linha 4 o programa estabelece a regra de retornar os aplicativos de acordo com as categorias e dispositivos cadastrados no banco de dados. Considere agora que o desenvolvedor necessite ter conhecimento sobre todas essas regras para então realizar uma mudança. Isto torna o processo de compreensão custoso mesmo para desenvolvedores experientes. Sendo assim, o exemplo do Código fonte 2 é uma prova que para compreender programas existe a necessidade de um aparato técnico mais amplo. Por exemplo, se o método *GetAppListByCategoryRange(...)* tivesse pré-condições e pós-condições definidas, o entendimento do código seria mais fácil, visto que melhoraria a legibilidade no momento que reduziria a quantidade de declarações *if*.

Código fonte 1 Método responsável por listar aplicativos cadastrados.

```
1  /// <summary>
2  /// Retrieves applications from a category
3  /// </summary>
4  /// <param name="category">Category name used such a filter</param>
5  /// <param name="from">Start of Paging</param>
6  /// <param name="quantity">Lenght of result list</param>
7  /// <param name="capabilities">List of object capabilities.</param>
8  /// <param name="Density">Density of images of the applications</param>
9  /// <returns>List of ApplicationWS</returns>
10 public List<DBC.ApplicationWS> GetAppListByCategoryRange(string
       category, int from, int quantity, List<DBC.CapabilityWS>
       capabilities, string Density)
11 {
12     DBM.Device dbDev = null;
13     if (from >= 0 && quantity > 0)
14     {
15         if (Util.HttpHeaders.UserAgentModel != null)
16         {
17             DBA.Device device = new DBA.Device();
18             dbDev = device.Fix(new DBM.Device() { Model = Util.
       HttpHeaders.UserAgentModel }, capabilities);
19         }
20         DBM.Category dbcat = new DBM.Category { Name = category };
21         if (dbcat != null && dbDev != null)
22         {
23             DBA.Application application = new DBA.Application();
24             DBM.Category dbcat2 = new DBM.Category { Name = "Tablet" };
25             List<DBM.Application> Apps = application.Get(dbcat2, dbcat,
        dbDev, from, quantity);
26             List<DBC.ApplicationWS> laws = DBC.ApplicationWS.Convert(
       Apps, Density);
27             ...
28         }
29         ...
30     }
31     ...
32     return new List<DBC.ApplicationWS>();
33 }
```

Código fonte 2 Implementação do método *Get(...)*.

```
1  public List<DBM.Application> Get(DBM.Category categoryA, DBM.Category
       categoryB, DBM.Device device, int from, int quantity, DBM.
       MorpheusDBEntities databaseContext = null)
2  {
3      DBM.MorpheusDBEntities db = Database.Context.Get(databaseContext);
4      List<DBM.Application> apps = Get(categoryA, categoryB, device, db);
5      List<DBM.Application> appsByCategory = new List<DBM.Application>();
6
7      if (apps != null)
8      {
9          if (from >= 0 && quantity > 0)
10         {
11             appsByCategory = apps.Skip(from).Take(quantity).ToList<DBM.
       Application>();
12         }
13     }
14     return appsByCategory;
15 }
```

1.3 Relevância

Compreender programas é uma atividade realizada constantemente por programadores que têm como objetivo evoluir sistemas. Nas últimas décadas, pesquisadores vêm dedicando esforços em trabalhos envolvendo compreensão de programas [32; 58; 48; 44; 62]. A maioria dos trabalhos na área propõem técnicas com base em observações, ontologias e pesquisas no código ou em meios externos (internet). No entanto desconhecemos a existência de uma abordagem sistemática utilizando *DbC* para este propósito. Nesse sentido, *DbC* possibilitaria benefícios que somados a compreensão de programas minimizariam problemas relacionados a ausência da documentação do código, a falta de testes e a inexistência da especificação dos requisitos. Portanto, pensando em unificar os benefícios que o *DbC* agrega ao desenvolvimento, uma abordagem sistemática utilizando contratos para auxiliar programadores a compreenderem programas é relevante.

Além de tornar possível a compreensão de programas, em especial os que foram desenvolvidos por outras pessoas, a abordagem motiva a escrita de contratos, pois é de conhecimento que desenvolvedores não costumam escrevê-los nos programas [46]. Dentre os motivos para não escrita destacam-se o tempo limitado para o desenvolvimento do programa e a dificuldade na escrita e manutenção de contratos.

1.4 Solução

Com o intuito de tratar dos problemas citados anteriormente, este trabalho propõe um estudo por meio de uma abordagem sistemática baseada na escrita de contratos para compreender programas. O objetivo do estudo é compreender como os desenvolvedores escrevem contratos. Para isso, é descrita uma abordagem contendo passos que auxiliam o desenvolvedor a analisar o método e a definir os contratos. A análise é feita para a assinatura, para a chamada, e para as variáveis definidas no escopo e em uma classe *C#*. Para a escrita de contratos, a abordagem propõe o uso de pré-condição e pós-condição utilizando a biblioteca *Code Contracts*. A principal razão em fazer o uso da biblioteca é que os contratos podem ser escritos utilizando a mesma linguagem de programação que os desenvolvedores utilizam para desenvolver ou manter programas. Outras razões adicionais referem-se a integração com as linguagens que fazem o uso do *.NET Framework*, e a adoção da biblioteca, que foi baixada mais de 207 mil vezes[1].

Os contratos escritos são interpretados pelo compilador e validados em tempo de execução pelos testes de unidade existentes no programa. Dessa forma, se o teste não violar o contrato indica que a suposição do desenvolvedor está correta, implicando no aumento de conhecimento. Do contrário, a suposição do desenvolvedor está equivocada, implicando em uma menor compreensão devido está não ocorrer na sua totalidade.

Pensando na aplicabilidade da abordagem, esta pode ser empregada para a qualquer código-fonte desenvolvido em linguagens de programação que deem suporte ao uso de contratos. Entretanto, a abordagem proposta é avaliada em métodos dentro de classes declaradas em programas escritos na linguagem *C#*.

[1] https://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970

1.5 Avaliação

Dois estudos foram realizados a fim de avaliar a abordagem. O primeiro deles consistiu na busca do refinamento dos passos descritos pela abordagem, enquanto o segundo observou o comportamento de desenvolvedores utilizando a abordagem, em termos da qualidade da atividade de compreensão e da efetividade dos contratos escritos.

Ambos os estudos utilizaram o protocolo *Think Aloud* [41], para capturar as estratégias e dificuldades dos programadores. Desse modo, três métodos de um programa *open-source* escrito na linguagem *C#* foram utilizados. Além do propósito, o que difere os dois estudos é a quantidade de desenvolvedores que participaram de cada um. Nesse sentido, para o primeiro foi solicitado a três desenvolvedores que utilizassem a abordagem de modo a refinar os passos descritos por esta. Os resultados auxiliaram a aprimorar os passos que descrevem a abordagem. O segundo estudo teve a participação de 12 desenvolvedores que foram orientados a compreenderem os três métodos de uma classe, advinda de um projeto real de código aberto, escrevendo contratos. Os resultados deste estudo classificam a abordagem como sendo de fácil usabilidade por 83% dos desenvolvedores, além de apontar indícios de compreensão do código.

1.6 Resultados e Contribuições

As contribuições desse trabalho são:

- uma abordagem sistemática para auxiliar a compreensão de programas, e motivar a escrita de contratos apresentada no Capítulo 3;
- uma avaliação com 12 desenvolvedores aplicada para compreensão de programas, apresentada no Capítulo 4.

1.7 Estrutura

Essa dissertação segue estruturada da seguinte forma. No próximo capítulo estão os fundamentos necessários para o entendimento do trabalho. No Capítulo 3, é apresentada a abordagem para compreensão de programas, e no Capítulo 4, é descrita a metodologia e

os resultados da avaliação do trabalho. Por fim, são apresentadas as considerações finais, seguidos de trabalhos relacionados e futuros (Capítulo 5).

Capítulo 2

Fundamentação Teórica

Neste capítulo, é apresentada a fundamentação teórica necessária para o entendimento desta dissertação. A princípio, na Seção 2.1, é apresentado o conceito de compreensão de código. Em seguida são detalhadas as abordagens existentes, seguido do apoio ferramental. Posteriormente, na Seção 5.1.2, os conceitos de *Design by Contract* são apresentados, bem como um exemplo expondo o uso da biblioteca *Code Contracts* na (Seção 2.2.1). Por fim, as ferramentas existentes, seguido do estado da arte são apresentados na Seção 2.2.2.

2.1 Compreensão de Programas

O processo de compreensão de programas é importante para a evolução de sistemas. De acordo com o padrão IEEE 1219 [17], o termo evolução está relacionado a modificação de um *software* existente, com o propósito de corrigir falhas ou adaptar o programa ao ambiente modificado, visando aumentar seu desempenho ou outras qualidades. Levando em consideração essa descrição, para que desenvolvedores possam desenvolver ou manter programas é necessário fazer o uso de linguagens de programação. No entanto, as linguagens possuem especificações distintas, obrigando os desenvolvedores a possuírem certo nível de conhecimento sobre as regras empregadas por cada uma delas. Isto pode dificultar a compreensão de programas mesmo para desenvolvedores experientes, visto que não basta apenas ter conhecimento dos paradigmas da linguagem. Diferentemente da linguagem natural, os trechos de códigos são interpretados por compiladores. Os compiladores analisam um conjunto de especificações pré-definidas por cada linguagem. Nesse sentido, são eles quem determinam

o resultado da execução com base na leitura dos códigos.

Ao longo do tempo, pesquisas têm sido realizadas [6; 66; 51; 53; 64; 9] relacionando outros elementos à compreensão de programas. Estes elementos possuem teorias e modelos distintos. Dentre eles, vale ressaltar que o conhecimento prévio sobre o domínio do programa é importante para que os desenvolvedores compreendam-no mais rapidamente. Dentre os domínios, existem três tipos diferentes de abordagens de compreensão, são elas: a abordagem *bottom-up*, a abordagem *top-down* e a uma abordagem híbrida. De modo a fortalecer esse entendimento, as três abordagens são descritas a seguir.

2.1.1 Abordagem Bottom-up

A abordagem *bottom-up* define que desenvolvedores começam lendo certos trechos de códigos. Os trechos lidos possuem um significado específico que é armazenado mentalmente. Na medida em que o número de trechos armazenados cresce, ocorre uma combinação entre eles. Desta forma, o desenvolvedor consegue adquirir níveis mais altos de abstração do programa.

Partindo da análise do Código fonte 1 (Seção 1.2), na Linha 12 é possível verificar que o atributo *dbDev* é definido com um valor nulo. Seguindo para a Linha 18, um novo valor é definido para o atributo *dbDev*. Dando continuidade à análise, na Linha 20 o atributo *dbcat* é instanciado recebendo na sua propriedade *Name* um valor do tipo *string*. Sequencialmente, na Linha 21, percebe-se uma condição de dois atributos que não podem ser nulos. Até o momento a leitura do código é insuficiente para predizer o que o método faz. Todavia, as instruções subsequentes podem auxiliar no entendimento da condição. Ao verificar as Linhas 23 e 24 é possível visualizar que dois objetos são instanciados. O primeiro deles chamado *application* faz referência ao tipo *Application*. O segundo chamado *dbcat2*, faz referência ao tipo *Category* e necessita que um nome seja definido. Até o momento um número pequeno de instruções foi lido. No entanto, ao juntar as instruções das Linhas 12, 18, 20, 21, 23 e 24 é subentendido que os atributos são necessários para alguma instrução futura. Sendo assim, dando continuidade a análise, na Linha 25 mais um objeto é definido, porém dessa vez ao invés de instanciá-lo, uma chamada é feita para o método *Get(...)* passando alguns argumentos. Dentre estes, *dbcat2*, *dbcat*, *dbDev*, *from* e *quantity* seguem respectivamente nas Linhas 24, 20, 18 e 10. Dessa forma o desenvolvedor adquire conhecimento sobre o método ao analisar sequencialmente as instruções. Contudo é necessário continuar a leitura

para adquirir um conhecimento mais aprofundado.

Pennington [43] em sua pesquisa apresenta uma teoria de compreensão de programa com base na compreensão do código-fonte. Ela observou que os desenvolvedores iniciam criando uma abstração do fluxo de controle do programa capturando uma sequência de operações do código. Essas sequências de operações formam conjuntos menores que posteriormente são unificados formando conjuntos maiores. Desse modo, os desenvolvedores passam a ter conhecimento do domínio do programa.

Analisando Código fonte 1, pode-se perceber que o método descreve instruções diversificadas. No entanto, de maneira a esclarecer a abordagem do modelo de Pennington, o desenvolvedor começaria avaliando os trechos de códigos contidos no escopo do método *GetAppListByCategoryRange(...)*. Com o passar das linhas, esse conhecimento aumentaria, chegando ao ponto ser possível esclarecer qual a função o método realiza.

O modelo de Shneiderman e Mayer [51] por sua vez, estabelece uma diferença entre o conhecimento sintático e semântico dos programas. O conhecimento sintático apresenta dependência do idioma e concentra-se em declarações e unidades básicas de um programa. Por outro lado, o conhecimento semântico independe de linguagem e sua construção é dada de maneira progressiva até que o modelo mental possa ser formado descrevendo o domínio da aplicação. Desse modo, o programa pode ser compreendido com a junção dos fragmentos sintáticos e componentes semânticos.

Adotando o modelo proposto por Shneiderman e Mayer [51] o desenvolvedor utiliza dois conhecimentos, conhecidos como conhecimento semântico e conhecimento sintático. Buscando adotar o modelo proposto, o desenvolvedor inicia com o modelo sintático. Por exemplo, o método *GetAppListByCategoryRange(...)* possui como tipo de retorno uma lista. É de conhecimento que esse tipo de retorno na linguagem *C#*, devolve um ou mais objetos que fazem referência ao seu tipo. Nessa perspectiva, isto pode auxiliar o desenvolvedor no entendimento de uma possível representação do método. Por outro lado, o modelo também descreve a importância do conhecimento semântico. Nestes casos, a experiência com outros sistemas pode auxiliar no entendimento da funcionalidade pretendida.

2.1.2 Abordagem Top-down

De maneira alternativa, a abordagem *top-down* é aplicada por desenvolvedores que possuem conhecimentos prévios de outros domínios de programas. Estes conhecimentos podem ter sido adquiridos em soluções desenvolvidas pelos próprios desenvolvedores ou em programas que são similares ao que está em análise. Desse modo, o modelo proposto por Brooks [6] sugere que o desenvolvedor cria hipóteses sobre o programa no momento que o mesmo possui conhecimento da solução. Essas hipóteses representam suposições a respeito das funcionalidades do programa. Dessa forma, o avanço do entendimento está relacionado com as hipóteses que vão conduzindo a validação do código.

Sendo assim, desenvolvedores tendem a utilizar a abordagem *top-down*, iniciando com a formulação de hipóteses do programa. Em seguida, estas são refinadas podendo ser confirmadas ou refutadas. O que determina essa condição é a presença ou a ausência do conjunto de estruturas ou operações no código. Assim, diante desta explicação é importante exemplificar como seria na prática a adoção do modelo proposto por Brooks [6].

Considerando o Código fonte 1, é importante salientar que o escopo do exemplo é limitado para a abordagem *top-down*, uma vez que expõe uma funcionalidade do sistema. Entretanto é possível exemplificar a abordagem, devido a presença de instruções que podem predizer informações relevantes sobre o respectivo trecho. Assim, observando os comentários presentes nas Linhas 1 à 9, o desenvolvedor pode formular um conjunto de hipóteses a respeito da funcionalidade do código. Por exemplo, na Linha 4 é descrito o nome do atributo *category* como um filtro. Logo, a hipótese pode ser definida com a seguinte pergunta “a categoria é nomeada por gêneros de filmes?”.

Todavia, de modo complementar à abordagem, Soloway e Ehrlich [53] acreditam que desenvolvedores analisam o programa buscando trechos no código que possam ser relevantes para o entendimento do programa. Em estudo, eles observaram que desenvolvedores experientes buscam padrões e regras de conhecimento nestes trechos. Os padrões caracterizam os cenários da programação, e as regras as implementações dos códigos. Sendo assim, um exemplo de padrão na linguagem *C#* seria o nome dos métodos que devem começar com letra maiúscula. Já um exemplo para regra seria a atribuição de um valor para um tipo específico de variável, por exemplo, um valor *string* só pode ser atribuído a uma variável que possua o tipo *string*.

Considerando o Código fonte 1, primeiro o desenvolvedor realiza a análise do código em busca de alguma regra ou padrão característico do paradigma da linguagem. Por exemplo, na Linha 20 a variável *dbcat* é instanciada passando um nome da categoria. Essa variável *Name* recebe uma variável chamada *category* que é do tipo *string* como pode ser observado nos argumentos do método na Linha 10. Caso o desenvolvedor não passe o nome da categoria, provavelmente o código que desempenha a função principal do método não seria executado. Neste sentido, o desenvolvedor ao adquirir esse conhecimento, passa a perceber que o valor presente na variável *dbcat* é relevante.

2.1.3 Abordagem Híbrida

Como complemento, uma abordagem para compreender programas é proposta combinando as abordagens *bottom-up* de Pennington e *top-down* de Soloway [63]. Von Mayrhauser e Vans, em estudos, perceberam que desenvolvedores alternam entre as abordagens de compreensão. Dessa maneira, propuseram dividir o processo de compreensão de código em quatro componentes. Dentre estes, três - (1) modelo do programa, (2) modelo situação e (3) modelo de domínio - são responsáveis por construir o entendimento do código e um - (4) base de conhecimento - é destinado a descrever a base de conhecimento necessária para executar o processo de compreensão.

O modelo do programa utiliza o modelo proposto por Pennington, descrevendo a necessidade de construir um modelo mental para entender o código desconhecido. Uma vez que a representação do modelo do programa é definida, o modelo de situação é desenvolvido criando um fluxo de dados ou uma abstração funcional. No entanto, se o código é conhecido, não existe a necessidade de formular novas hipóteses para resolver o problema. Nestes casos, o modelo do domínio auxilia o desenvolvedor no processo de resolução do problema, visto que o mesmo possui conhecimento prévio. Por fim, a base de conhecimento atua como um repositório para qualquer novo conhecimento adquirido.

Letovsky [28] também considera que a compreensão de programas ocorre utilizando as abordagens *bottom-up* e *top-down*. Seu modelo, segue de acordo com o modelo desenvolvido por Brooks [6]. Nele, desenvolvedores são definidos como processadores oportunistas que levam em consideração o domínio e outras abordagens para implementar programas. Três componentes formam o seu modelo.

O primeiro deles é a base de conhecimento do desenvolvedor. Essa base é responsável por quantificar a experiência e conhecimento deste. Essa quantificação, consiste na experiência com linguagem de programação, com desenvolvimento de programas, com o uso de bibliotecas, entre outros. O segundo componente apresentado é responsável pelo modelo mental. Inicialmente, o modelo mental consiste em especificar os objetivos do programa. Essas especificações, consequentemente tendem a evoluir, levando a implementações do código. Assim, de acordo com as implementações, o modelo mental inclui um mapeamento das partes relevantes do programa. Por fim, o terceiro componente descreve o surgimento da evolução do modelo mental do desenvolvedor.

Diante desses pontos, a abordagem integrada pode auxiliar na formação de base para identificação de necessidades de informações durante a compreensão de programas, podendo ser útil para definição de ferramentas.

2.1.4 Ferramentas de Compreensão de Código

Compreender programas continua sendo uma atividade difícil devido a falta de documentação e a inconsistência entre os requisitos e as implementações no programa. Por estes motivos, desenvolvedores tendem a recorrer ao código-fonte de modo a compreender como o programa funciona. Além disso, compreender programas muitas vezes exige o entendimento das funcionalidades do sistema. Nesse sentido, pesquisadores têm dedicado esforços para facilitar o entendimento de programas. Assim, diversas pesquisas foram realizadas com desenvolvedores da indústria e da academia. Como consequência, uma variedade de ferramentas apresentando abordagens distintas têm sido propostas.

De acordo com Tilley et al. [61] a ideia principal das ferramentas é prover suporte a várias atividades que são características ao processo de compreensão de programas. Estas, por sua vez, adotam perspectivas funcionais e comportamentais. As perspectivas funcionais objetivam fornecer aos utilizadores uma visão das funcionalidades do programa, enquanto as perspectivas comportamentais referem-se em como essas funcionalidades são desempenhadas no programa.

Storey [58], analisou algumas das principais teorias cognitivas de compreensão de programas que apareceram ao longo das últimas décadas. Em seu estudo ela verificou como diferentes ferramentas auxiliam os desenvolvedores a compreenderem artefatos do programa.

Isso, também fica claro em outras etapas do processo de desenvolvimento do *software* [8]. Portanto, é visível como as ferramentas auxiliam nas fases de desenvolvimento, manutenção e documentação. Adiante, alguns exemplos de ferramentas são apresentados.

Rigi é uma ferramenta interativa que utiliza a abordagem de engenharia reversa. Proposta por Müller e Klashinsky [40], ela possui um editor gráfico, chamado *RegiEdit*, que permite visualizar a estrutura do sistema. Seu principal foco é prover suporte e criação a descoberta de abstrações e hierarquias de subsistemas a desenvolvedores, assim facilitando a compreensão do programa durante a manutenção do mesmo.

SHriMP [56] utiliza uma abordagem integrada com recursos organizados em janelas identificadas. Seu foco principal consiste em fornecer visualização de gráficos aninhados para apresentar a hierarquia do sistema, permitindo aos utilizadores explorar informações em múltiplas perspectivas e níveis de abstração. Isso facilita a análise de dependências dos módulos principais, visto que é possível visualizar as relações entre as classes, chegando até ao código-fonte.

SNiFF+ é um ambiente de desenvolvimento comercial destinado a desenvolvedores profissionais que desejam analisar o código-fonte de grandes sistemas [59]. Apresentando um conjunto de ferramentas de análise, navegação e compreensão, sua proposta é aumentar a produtividade dos desenvolvedores e a qualidade do programa. Diante disto, *SNiFF+* provê suporte a diferentes ferramentas de desenvolvimento, sendo possível integrar-se a uma variedade de linguagens de programação, editores de texto e compiladores.

CODES é um *plugin* para Eclipse que documenta códigos-fonte *Java*. Esse *plugin* auxilia desenvolvedores de duas formas, redocumentando o seu próprio código-fonte ou documentando sistemas desconhecidos que carecem de documentação. Para isto, *CODES* [62] utiliza um minerador de descrição do fórum de discussão *StackOverflow*[1]. Nesse sentido, para que a busca seja realizada, os desenvolvedores necessitam escolher os métodos que desejam compreender. Assim, o *plugin* inicia as buscas com um conjunto de descrições formados por palavras chaves como nome do projeto, nome

[1] http://stackoverflow.com/

da classe e nome do método. Ao serem encontrados no fórum, uma lista de métodos é retornada com as descrições. Essas descrições quando implantadas nos métodos, aparecem no formato *Javadoc* [21].

2.2 Design by Contract

Esta seção mostra uma visão geral da teoria utilizada pela abordagem *Design by Contract* (*DbC*) [36; 38; 39]. Descrito por Bertrand Meyer [36], contratos são fatos a respeito de um programa que devem ser verdadeiros. Estes, quando verdadeiros, minimizam a existência de *bugs*. Contudo, para isto, é preciso adicionar contratos a programas orientados a objetos durante atividades de *design* e implementação.

Contratos podem aparecer como parte de um método (pré-condições, pós-condições), como invariantes de classe e como instruções de checagem. Estes são construídos por uma ou mais cláusulas que representam expressões booleanas. As expressões booleanas envolvem constantes e/ou variáveis que resultam em apenas dois valores; verdadeiro ou falso. Para que uma cláusula seja verdadeira, o contrato deve ser verdadeiro, caso contrário, ocorrerá violação, resultando no lançamento de uma exceção no código.

Neste contexto, contratos são escritos utilizando construções das linguagens de programação. Estes são checados no código estaticamente e dinamicamente, beneficiando os desenvolvedores com a documentação, depuração e teste. Os contratos são definidos em:

- pré-condições: responsáveis por garantir que as condições do método são verdadeiras para que sejam executadas;
- pós-condições: responsáveis por garantir que o comportamento do método é verdadeiro após a execução; e
- invariantes: responsáveis por garantir que os objetos da classe satisfazem determinado comportamento antes e depois da execução dos métodos.

2.2.1 Exemplo

Com o propósito de explicar como definir contratos com a biblioteca *Code Contracts*, o método descrito no Código fonte 1 (Seção 1.2) é utilizado. A princípio apenas a assinatura do

método é levada em consideração. Desta forma, provavelmente a primeira reação dos desenvolvedores é se questionarem a respeito das definições do método. Estes questionamentos são importantes para a definição das hipóteses que posteriormente são transformados em contratos. A seguir alguns questionamentos são definidos para o método *GetAppListByCategoryRange(...)*:

- o que acontece se a categoria for inexistente?;
- o que acontece se a lista for vazia?; e
- como garantir que o atributo *from* seja sempre menor ou igual ao atributo *quantity*?.

Tendo o conhecimento das perguntas definidas acima, os desenvolvedores podem então estruturar as pré-condições, pós-condições e invariantes do programa. Utilizando o *Code Contracts* é possível definir contratos em qualquer linguagem do pacote *.NET Framework*. Assim, com a finalidade de exemplificar o seu uso, a seguir é definido um contrato para cada tipo.

As pré-condições são empregadas para especificar valores de parâmetros. Estas podem ser escritas utilizando o método *Requires()* definido na classe *Contracts*. Por exemplo, o contrato definido para o método *GetAppListByCategoryRange(...)* pode ser escrito conforme mostrado no Código fonte 3.

Código fonte 3 Definição de pré-condição.

```
1 ...
2 public List<DBC.ApplicationWS> GetAppListByCategoryRange(string
      category, int from, int quantity, List<DBC.CapabilityWS>
      capabilities, string Density)
3 {
4   Contract.Requires(category != null);
5   ...
6 }
```

Na Linha 4 a pré-condição verifica a existência da categoria para que a lista de aplicativos seja retornada. Caso a categoria não seja definida, uma exceção será lançada pelo *ContractException*. Desta forma, fica claro que pré-condições definem o que esperar das entradas dos

métodos. Por outro lado, pós-condições são úteis para expressar os resultados das chamadas dos métodos. Apesar das declarações serem exigidas no início destes, estas são verificadas apenas após a execução dos métodos. Assim, para escrevê-las é preciso utilizar o método *Ensures()*. Exemplificado uma pós-condição, o Código fonte 4 define a escrita do contrato.

Código fonte 4 Definição de pós-condição.

```
1 ...
2 public List<DBC.ApplicationWS> GetAppListByCategoryRange(string
      category, int from, int quantity, List<DBC.CapabilityWS>
      capabilities, string Density)
3 {
4   Contract.Ensures(Contract.Result<List<DBC.ApplicationWS>>() != null);
5   ...
6 }
```

A expressão *Contract.Result<List<DBC.ApplicationWS»() != null* é responsável por verificar se após a execução do método a lista não é vazia. Caso o resultado seja vazio uma exceção será lançada indicando que a pós-condição foi violada. Neste sentido, a violação do contrato transmite ao desenvolvedor o conhecimento de que a lista é vazia. Desta forma ele pode verificar se o código é tratado para esta rotina. Todavia, vale destacar que a exceção pode ser tratada utilizando a expressão *Contract.EnsuresOnThrow*.

Por fim, as invariantes de objetos são mais difíceis de especificar, porque devem ser verdadeiras para cada instância da classe antes e depois de cada chamada. Diferentemente das pré-condições e pós-condições, as invariantes de objeto são definidas em um método a parte. Este método só pode conter instruções de chamadas para o método *Invariant()*. Para definir invariantes de objeto é preciso declarar um método com visibilidade protegida e com a seguinte anotação *[ContractInvariantMethod]* acima da assinatura do método. O Código fonte 5 representa um exemplo do contrato.

2.2.2 Linguagens e Ferramentas

Algumas linguagens e ferramentas de programação suportam o uso de contratos para expressar conformidades no código. Estas apoiam os desenvolvedores na escrita e na verificação

Código fonte 5 Definição de invariante.

```
1 ...
2 [ContractInvariantMethod]
3 protected void ObjectInvariant()
4 {
5   Contract.Invariant(from <= quantity);
6   ...
7 }
```

de contratos nos programas, fazendo o uso da abordagem *DbC* fundamentada na verificação e especificação formal da lógica de Hoare [15]. Exemplos de tais linguagens e ferramentas incluem *Eiffel* [37], *Java Modeling Language* (*JML*) [24], *iContract* [22].

Dentre as apresentas acima, a linguagem *Eiffel* é a pioneira. Apresentada em 1986 por Meyer [35], *Eiffel* é uma linguagem que integra os conceitos de engenharia de *software* da época a orientação a objetos. Seu objetivo é construir programas reutilizando seus componentes, de modo a diminuir os custos, aumentar a qualidade dos programas e a produtividade dos desenvolvedores. Para isto, a linguagem permite acesso a classes, métodos e bibliotecas por meio de interface de outras linguagens de programação, a exemplo de *C* e *C++*.

No entanto, o presente estudo utiliza *C#*, uma linguagem orientada a objetos integrada ao ambiente *.NET Framework* [67]. Dentre os sistemas e bibliotecas que fazem o uso da linguagem *C#* e suportam a utilização dos conceitos de *DbC*, destacam-se *Spec#* e *Code Contracts*. *Spec#* [2] é uma linguagem formal que permite a interoperabilidade de bibliotecas existentes da linguagem *C#* e do *.NET Framework*. Essa provê ao desenvolvedor especificar pré-condições e pós-condições em métodos e em algumas abstrações de dados de nível superior no código-fonte. Além disso, é possível distinguir referências de objetos não nulos a partir de referências de objetos possivelmente nulos e gerenciar exceções no código-fonte. Contudo *Spec#* garante o funcionamento apenas pelo que for escrito por ela.

Pensando nisto, Fähndrich et al. [12] desenvolveram o *Code Contracts*, uma ferramenta capaz de integrar contratos em códigos-fonte *C#*. Ao invés de inserir anotações contratuais personalizadas [2] ou comentários [24], a ferramenta permite que marcações dos métodos sejam detectáveis pelo compilador da linguagem tornando-os parte da execução do programa. Isto facilita o uso pelos desenvolvedores no momento em que não é necessário aprender uma

nova linguagem. Assim, *Code Contracts* aproveita ao máximo a linguagem, o compilador e o ambiente de desenvolvimento.

Capítulo 3

Uma Abordagem para Auxiliar a Compreensão de Programas

No capítulo anterior são destacadas abordagens e ferramentas utilizadas por desenvolvedores para auxiliar na compreensão de programas, bem como os conceitos de *DbC*. Neste capítulo, é descrita uma abordagem utilizando contratos para compreender programas. Esta é aplicada a linguagens de programação orientadas a objetos que possuam suporte a contratos, e a programas que contenham testes.

A abordagem provê a compreensão do código, à medida que contratos são escritos nos métodos. Os contratos escritos são então validados por meio da execução dos testes de unidade contidos no programa. Neste sentido, detectada ou não a violação do contrato, o desenvolvedor adquire conhecimento parcial ou total acerca do comportamento do método.

Inicialmente, na Seção 3.1 é mostrada uma visão geral da abordagem. Nas Seções 3.2 a 3.6, são descritas as etapas da abordagem. Cada etapa é exemplificada com trechos de um programa *open-source* utilizado na avaliação deste estudo. Em seguida, na Seção 3.7 é descrita a compreensão do método. Por último, são apresentadas as limitações da abordagem (Seção 3.8).

3.1 Visão Geral

Especificada para ser aplicada em diversas linguagens de programação orientada a objetos, atualmente, a abordagem foi definida sobre programas escritos em *C#* [50; 67] que possui

satisfatório suporte ferramental para *DbC* através da biblioteca *Code Contracts* [12; 30]. Entretanto, esta pode ser aplicada de maneira similar a outras linguagens.

A seguir, é descrita a abordagem que auxilia desenvolvedores a compreender programas. Dividida em cinco etapas, a Figura 3.1 ilustra a abordagem. Sendo assim, para a Etapa 1 ocorre a identificação do método a ser compreendido, pelo desenvolvedor, de acordo com a sua necessidade. Na Etapa 2, é identificada a hierarquia de chamadas do método escolhido, que ocorre manualmente, consistindo na procura de métodos definidos pelo programa. Caso não existam chamadas de métodos definidos pelo programa, o desenvolvedor deve analisar a assinatura do método em que se encontra. Em seguida na Etapa 3, ocorre a escrita de contratos, sendo estes pré-condição e/ou pós-condição, utilizando o *Code Contracts*. Os contratos escritos são então validados pela execução dos testes de unidade do programa (Etapa 4). É importante mencionar que a execução dos testes deve ocorrer a cada pré-condição ou pós-condição escrita, pois dessa forma é mais fácil compreender o comportamento do método (rotina), uma vez que o desenvolvedor valida sua suposição visualizando o resultado do teste. Nesse sentido, a execução dos testes resulta na violação ou não do contrato. Dessa maneira, caso o teste não o viole, significa que o contrato é válido indicando aumento de conhecimento do desenvolvedor. Por outro lado, a falha do teste indica a violação do contrato, entretanto, esta ainda gera conhecimento para o desenvolvedor, porém em menor quantidade por não ocorrer em sua totalidade. Considerada a análise dos resultados dos testes de unidade, na última etapa deve-se avaliar a necessidade em escrever novos contratos. Dessa forma, o desenvolvedor pode definir novos contratos ou então considerar desnecessária a escrita destes para o método que encontra-se. Caso o desenvolvedor julgue desnecessária a escrita, é declarada a compreensão para aquele trecho do programa. No entanto, caso o desenvolvedor tenha escrito contrato para métodos declarados pelo programa, este deve voltar na hierarquia, e reiniciar a aplicação da abordagem para o método que realizou a chamada.

É importante destacar que a abordagem necessita dos testes de unidade do programa para validar as suposições dos desenvolvedores. As suposições são expressas no formato de contratos nos trechos do programa. Dessa forma, os contratos escritos além de validarem as suposições do desenvolvedor, tornam-se complementares para o programa, ao assegurar que os requisitos estão em conformidade com os testes de unidade, que foram definidos pela equipe oficial que desenvolveu o programa. Além disso, a abordagem descreve uma

metodologia sistemática, por meio de passos, que especifica quais tipos de contratos devem ser escritos. Em suma, esta pode ser utilizada em qualquer linguagem de programação que suporte os princípios de *DbC*. Por fim, a abordagem não se encontra automatizada. A seguir, as etapas da abordagem são detalhas.

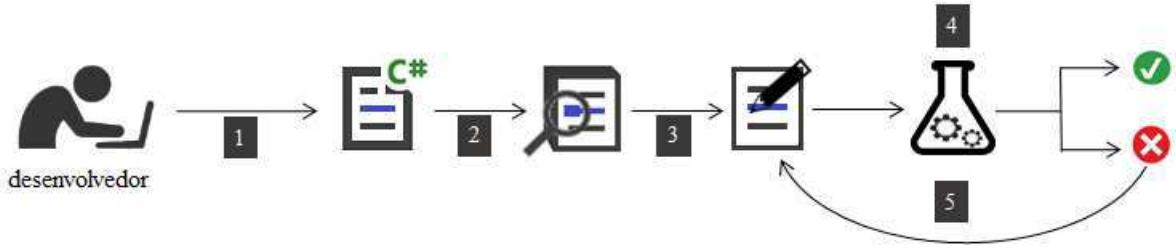


Figura 3.1: Abordagem para auxiliar a compreensão de programas e a escrita de contratos; 1. Identificação do método, 2. Análise da hierarquia buscando chamadas de métodos definidos pelo programa e a assinatura, 3. Escrita de pré-condição e/ou pós-condição, 4. Execução dos testes de unidade e visualização dos resultados indicando violação ou não do contrato, 5. Avaliação da escrita de novos contratos.

3.2 Etapa 1: Identificação do Método

De acordo com a Figura 3.1 a utilização da abordagem inicia, na Etapa 1, com o desenvolvedor identificando o trecho do programa ao qual pretende compreender. Para que isto ocorra, é preciso que este decida por qual método iniciar. Comumente, desenvolvedores iniciam por métodos maiores, com fluxos condicionais, por acharem que estes facilitam a compreensão, tendo em vista que uma rotina pode ser confirmada. Dessa maneira, identificado o método, é importante destacar que a sua visibilidade independe para a abordagem, podendo esta ser pública ou restrita. Sendo assim, a abordagem visa forçar a escrita de contratos como instrumento de comparação, com a vantagem de mantê-los no programa, uma vez que desenvolvedores não possuem prática e os consideram difíceis de escrever.

O método identificado nessa etapa é posteriormente analisado na Etapa 2. Nesse sentido, como forma de exemplificar a aplicabilidade da abordagem o método *Get(...)* é utilizado. Este representa um trecho[1] de programa real, de um projeto *open-source* chamado

[1] A implementação completa do trecho do programa está disponível no Apêndice A.

*Free SCADA*2[2], sendo o mesmo avaliado por este estudo. O Código fonte 6 apresenta a implementação do método.

3.3 Etapa 2: Análise Inicial

Após identificado o método, ocorre a Etapa 2, na qual analisa o escopo e a assinatura deste. Para essa tarefa, a abordagem, de modo sistemático, propõe passos que devem ser seguidos. Inicialmente é necessário especificar que para as próximas etapas, a abordagem considera, pelo menos uma vez, a escrita de contratos e a execução do teste. Assim, o fluxo a seguir descreve as ações que o desenvolvedor deve fazer:

- escrever uma pré-condição;
- executar o teste;
- escrever uma pós-condição;
- reexecutar o teste.

Considerando que a ordem na escrita de uma pré-condição ou pós-condição pode ser alterada, devido à composição da assinatura do método ou da inicialização da variável global da classe, o fluxo que representa a escrita de um contrato e a execução do teste deve ser adequado conforme cada trecho do programa. Nesse sentido, o objetivo é ter pelo menos uma escrita e uma execução dos testes de modo a auxiliar a compreensão. Por exemplo, para os métodos que possibilitam apenas a escrita de pós-condição, o desenvolvedor deve escrevê-la e em seguida validá-la com a execução dos testes, desconsiderando o fluxo que determina a escrita de pré-condição e execução dos testes.

Analisando o Código fonte 6 o primeiro passo é procurar as chamadas de métodos feitas pelo programa no *Get(...)*. Observando o método em questão, são identificadas duas chamadas nas linhas 7 e 30, sendo estas *GetProcessorArch()* e *GetAvailableDB()*. Tendo em vista a inexistência de outras, e encontrados dois métodos, o desenvolvedor deve seguir para o que é chamado primeiro. Assim, para o trecho do programa, o método *GetProcessorArch()*

[2]https://sourceforge.net/projects/free-scada/files/

Código fonte 6 Implementação do método *Get(...)*.

```
1  ...
2  public static DbProviderFactory Get(string invariantName)
3  {
4    if (invariantName == SQLiteName)
5    {
6      string sqlightAssembly = Path.Combine(AppDomain.CurrentDomain.
       BaseDirectory, "SQLite");
7      switch (GetProcessorArch())
8      {
9        case ProcessorType.x86:
10         sqlightAssembly = Path.Combine(sqlightAssembly, "x32");
11         break;
12       case ProcessorType.x64:
13         sqlightAssembly = Path.Combine(sqlightAssembly, "x64");
14         break;
15       case ProcessorType.IPF:
16         sqlightAssembly = Path.Combine(sqlightAssembly, "Itanium");
17         break;
18     }
19     sqlightAssembly = Path.Combine(sqlightAssembly, "system.data.sqlite.
       dll");
20     Assembly lib = Assembly.LoadFrom(sqlightAssembly);
21     foreach (Type t in lib.GetExportedTypes())
22     {
23       if(t.FullName == "System.Data.SQLite.SQLiteFactory")
24         return (DbProviderFactory)Activator.CreateInstance(t);
25     }
26     return null;
27   }
28   else
29   {
30     DataTable dt = GetAvailableDB();
31     DataRow[] rows = dt.Select(string.Format("InvariantName = '{0}'",
       invariantName));
32     if (rows.Length == 0)
33       return null;
34     return DbProviderFactories.GetFactory(rows[0]);
35   }
36 }
37 ...
```

é analisado. No entanto, algumas características devem ser consideradas. O Código fonte 7 descreve a implementação deste método.

Código fonte 7 Implementação do método *GetProcessorArch()*.

```
1  ...
2  private static ProcessorType GetProcessorArch()
3  {
4    using (System.Management.ManagementClass processors = new System.
       Management.ManagementClass("Win32_Processor"))
5    {
6      foreach (System.Management.ManagementObject processor in processors.
       GetInstances())
7      {
8        int AddressWidth = int.Parse(processor["AddressWidth"].ToString());
9        int Architecture = int.Parse(processor["Architecture"].ToString());
10
11       if (AddressWidth == 32)
12         return ProcessorType.x86;
13       if (AddressWidth == 64 && Architecture == 9)
14         return ProcessorType.x64;
15       if (AddressWidth == 64 && Architecture == 6)
16         return ProcessorType.IPF;
17     }
18   }
19   throw new NotSupportedException();
20 }
21 ...
```

As linguagens de programação que seguem o paradigma orientado a objetos (OO), possuem características que as fazem ser bastante utilizadas [38]. Uma dessas características é a sobrecarga de métodos. Nesse sentido, assumindo a possibilidade da existência de sobrecarga para o *GetProcessorArch()*, o desenvolvedor deve analisar se de fato esta existe. A abordagem trata a sobrecarga da seguinte forma: o desenvolvedor, ao identificar a existência de sobrecarga para o método chamado, deve iniciar pelo método sobrecarregado que contém o menor número de parâmetros. Este possivelmente representa a implementação mais sim-

ples do método por conter menos parâmetros que são utilizados no escopo. Por outro lado, existindo dois ou mais métodos sobrecarregados com apenas um parâmetro, o desenvolvedor deve iniciar pelo que foi chamado primeiro. Do contrário, não existindo sobrecarga, o desenvolvedor deve procurar chamadas a outros métodos definidos pelo programa. Por fim, existindo ou não sobrecarga, a abordagem determina que o desenvolvedor siga até o método mais interno do programa antes de iniciar a escrita de contratos. O ato de seguir para os métodos mais internos do programa é denominado método raiz.

Com o intuito de esclarecer a aplicação da abordagem, damos continuidade com a análise do método *GetProcessorArch()* descrito no Código fonte 7. Para este, é necessário refazer a análise inicial do seu escopo buscando por chamadas de métodos. Dito isto, considerando a inexistência de chamadas, deve-se então analisar a assinatura e o escopo deste para em seguida escrever contratos. Para a assinatura do método, a abordagem provê passos que descrevem a existência de retorno e parâmetros. Para o escopo a abordagem considera o uso de variáveis globais e variáveis que foram definidas no método. Em especial, para as variáveis globais só existirão pré-condição e/ou pós-condição caso estas sejam utilizadas no escopo do método em questão. A seguir são descritos os passos considerando a assinatura do método:

- Com retorno e com parâmetros – pós-condição e pré-condição;
- Com retorno e sem parâmetros – pós-condição;
- Com retorno, sem parâmetros e com variáveis globais - pós-condição (para o retorno e para as variáveis globais) e pré-condição (variáveis globais);
- Com retorno, com parâmetros e com variáveis globais - pós-condição (para o retorno e para as variáveis globais) e pré-condição (para os parâmetros e para as variáveis globais);
- Sem retorno e sem parâmetros – nada;
- Sem retorno e com parâmetros – pré-condição;
- Sem retorno, sem parâmetros e com variáveis globais - pré-condição e pós-condição para as variáveis globais;

- Sem retorno, com parâmetros e com variáveis globais - pré-condição (para os parâmetros e para as variáveis globais) e pós-condição (para as variáveis globais).

Por fim, ressaltamos dois passos presentes na abordagem que podem ser utilizados pelos desenvolvedores. O primeiro sugere a possibilidade em transformar variáveis definidas no escopo do método em variáveis globais, enquanto o segundo, define que a escrita de pré-condições e pós-condições para as variáveis globais deve ser feita apenas uma vez no método raiz. Estes passos podem ser observados, na Etapa 3, à medida que contratos são escritos.

3.4 Etapa 3: Contratos para Entendimento

Depois de identificado o método e analisadas as chamadas, ocorre a terceira etapa, na qual pré-condição e/ou pós-condição são escritas. Para a escrita, utiliza-se o *Code Contracts*; uma biblioteca com métodos estáticos que permite a validação do código, em tempo de execução, ao inserir contratos. Essa biblioteca também possibilita a análise estática do código, bem como a documentação do mesmo.

Antes de darmos continuidade com aplicação da abordagem, é importante mencionar que os métodos do *Code Contracts* utilizados por este trabalho são o *Contract.Requires(...)* para pré-condição e o *Contract.Ensures(...)* para pós-condição. Além disso, a biblioteca impõe requisitos que referem-se ao local e a ordem onde os contratos são definidos. Assim, para o local, a biblioteca determina que a escrita ocorra dentro do escopo do método, logo após a abertura da chave ("{"). Em relação a ordem, esta refere-se especificamente a pós-condição quando existir pelo menos uma pré-condição escrita, dessa maneira, é necessário definir a pós-condição depois da última pré-condição do método. Dito isto, seguimos com o processo de compreensão do *GetProcessorArch()* especificado no Código fonte 7. A seguir são escritos contratos considerando os passos, explicados na Etapa 2, da abordagem.

Conforme analisado nas etapas anteriores, sabemos da inexistência de chamadas a métodos do programa. No entanto, é observado que variáveis globais são utilizadas como retorno no seu escopo. Revendo os passos da abordagem, isto indica a escrita de pré-condição e pós-condição para estas. Além disso, analisando a assinatura do método, visualizamos que este contém apenas retorno indicando a escrita de pós-condição. Por fim, para as variáveis globais, é importante mencionar que a abordagem determina:

- Observar se há inicialização. Considerar a existência de pré-condições para variáveis globais que não são inicializadas;
- Definir pré-condições de acordo com o seu tipo;
- Observar onde a variável global é chamada e se existe alguma limitação;

A inicialização de variáveis pode conter algum valor que deve ser verdadeiro para ser utilizado pelo método. Por exemplo, constantes são campos com valores explicitamente definidos, em boa parte dos casos, que não são alterados e que fornecem nomes significativos para valores especiais. O desenvolvedor ao observar o uso de constante pode escrever pré-condições específicas envolvendo o valor e o tipo da variável. Todavia, mesmo para variáveis não inicializadas, contratos podem ser escritos, pois em alguns casos estas são relacionados às regras de negócio no método. É importante mencionar que a abordagem determina a escrita de uma pré-condição e/ou pós-condição por vez, nesse sentido, a compreensão ocorre de imediato ao executar e visualizar os resultados dos testes.

Assim, para as variáveis globais, utilizadas nas linhas 12, 14 e 16 do Código fonte 7, escrevemos a pré-condição de acordo com seu tipo. Segundo a implementação da classe, o tipo definido corresponde a uma enumeração, representada pela palavra-chave *enum* que é usada para declarar um conjunto de constantes nomeadas. No código, as enumerações nomeadas são *x86*, *x64* e *IPF* cujo tipo é *ProcessorType*. Adiante o Código fonte 8 descreve as enumerações definidas pela classe, nas linhas 2-7, e a pré-condição escrita na linha 11. O objetivo da pré-condição é validar a inexistência de uma quarta constante nomeada, no momento em que nega a existência de uma constante com o valor 3. Vale relembrar que para validar o contrato é necessário executar os testes (Etapa 4). Contudo, antes de seguir para a próxima etapa é preciso explicar outras características que a abordagem provê.

Considerando a existência de uma pré-condição, a abordagem determina caminhar no *trace* do método assumindo-a. O *trace* faz alusão a trajetória até um determinado ponto final permitindo a compreensão da rotina do método. Nesse sentido, um conjunto de pós-condições pode ser inferido considerando o seu resultado. A abordagem indica que uma pré-condição está para uma pós-condição, no entanto, nem sempre isto ocorre. Assim, para a pré-condição escrita, na linha 11 do Código fonte 8, podemos escrever uma pós-condição com o objetivo de compreender qual o retorno do método. Para isto, dentre as 3 possibilidades

Código fonte 8 Pré-condição verificando a existência de variáveis globais enumeradas no método *GetProcessorArch()*.

```
1  ...
2  enum ProcessorType
3  {
4    x86,
5    x64,
6    IPF
7  }
8  ...
9  private static ProcessorType GetProcessorArch()
10 {
11   Contract.Requires(!Enum.IsDefined(typeof(ProcessorType), 3));
12   ...
13 }
14 ...
```

contidas no *enum*, o Código fonte 9 traduz a intenção de determinar que o retorno represente o valor *x64*.

Código fonte 9 Pós-condição escrita assumindo o trace da pré-condição para as variáveis globais.

```
1  ...
2  private static ProcessorType GetProcessorArch()
3  {
4    Contract.Ensures(Contract.Result<ProcessorType>() > ProcessorType.x86
      && Contract.Result<ProcessorType>() < ProcessorType.IPF);
5    ...
6  }
7  ...
```

Escritos os contratos para as variáveis globais, devemos escrever a pós-condição para o retorno do método. Contudo, antes de escrevê-la, destacamos uma sugestão da abordagem que é transformar variáveis do método em variáveis globais. A sugestão é comumente adotada para métodos que implementam blocos *if* utilizando como condição variáveis definidas

no seu escopo. Dessa forma, como o retorno do método pode depender de condições envolvendo essas variáveis, é necessário defini-las na classe. Além disso, enquanto não forem transformadas em variáveis globais, essas não poderão ser utilizadas no contrato devido a escrita deste ocorrer antes da sua definição dentro do método. Desse modo, considerando a sugestão da abordagem, iremos transformar em variáveis globais *AddressWidth* e *Architecture* declaradas nas linhas 8 e 9 do método *GetProcessorArch()* do Código fonte 7.

Antes de mostrar como fazer a transformação, é necessário dizer que uma forma de inferir contratos para as variáveis globais que são utilizadas para determinar um retorno é utilizando implicações. Implicações indicam que uma condição deve ser satisfeita para que a outra seja verdade. Nesse sentido, como implicações representam expressões booleanas, estas podem servir como entrada para pós-condições. Todavia, a linguagem *C#* não disponibiliza um operador que represente implicações para expressões booleanas, assim, não possuindo suporte diretamente. Diante disto, existindo a necessidade de utilizar implicações lógicas, o desenvolvedor deve retirar do método o tipo da variável para em seguida declará-lo no escopo da classe precedido do mesmo nome atribuído a ela. Tal ação caracteriza a transformação de uma variável em global. Feito isto, o desenvolvedor deve verificar se é importante escrever contratos - pré-condição e/ou pós-condição - para as variáveis. A verificação deve considerar os mesmos critérios citados anteriormente. O Código fonte 10 descreve como fica a implementação do *GetProcessorArch()* após a transformação das duas variáveis.

Código fonte 10 Transformação das variáveis de método *AddressWidth* e *Architecture* em variáveis globais.

```
1  ...
2  private static int AddressWidth, Architecture;
3  ...
4  private static ProcessorType GetProcessorArch()
5  {
6    ...
7        AddressWidth = int.Parse(processor["AddressWidth"].ToString());
8        Architecture = int.Parse(processor["Architecture"].ToString());
9    ...
10 }
11 ...
```

Realizadas as transformações, desconsideramos a escrita de pré-condição para *AddresWidth* e *Architecture*, devido estas serem estáticas e não possuírem inicialização. Em relação a ser estática, o valor da variável só poderia ser alterado por meio de uma função. Assim, como a abordagem não caracteriza a criação de métodos, esta passa a ser desconsiderada. Além do mais, o fato de não haver inicialização induz o desenvolvedor a não escrever contratos para essa situação, uma vez que o conhecimento está implícito e não agrega conhecimento além do que já se sabe. Desta forma, para casos similares, se estes possuírem valores padrão determinados pelos seus tipos, não faz sentido verificá-los novamente. Todavia, cada caso precisa ser avaliado com atenção.

Por outro lado, a transformação favorece a escrita de pós-condição envolvendo implicações. Desse modo, considerado também a existência de pós-condição para o retorno do método, o contrato do Código fonte 11 descreve a suposição para um possível retorno. Por fim, com o intuito de validar os contratos escritos, os testes de unidade estabelecidos pelo programa são executados na próxima etapa.

Código fonte 11 Pós-condição utilizando implicação lógica para a variável global *AddresWidth* e o retorno do método *GetProcessorArch()*.

```
1 ...
2 private static ProcessorType GetProcessorArch()
3 {
4   Contract.Ensures(AddresWidth == 32 && Contract.Result<ProcessorType>()
     == ProcessorType.x86);
5   ...
6 }
7 ...
```

3.5 Etapa 4: Execução e Resultados dos Testes de Unidade

Nesta etapa a abordagem determinada que a escrita de contratos deve ser sucedida pela execução de todos os testes de unidade. Isto é, escrito um contrato os testes devem ser executados logo em seguida. A execução de todos ocorre porque os desenvolvedores dificilmente sabem de imediato quais estão relacionados às rotinas do programa. Em outras palavras,

por desconhecerem o programa eles não sabem qual executar. Nesse sentido, o objetivo da execução é validar as suposições que são escritas em forma de contratos. É importante mencionar que os testes devem ser executados antes da escrita de um contrato, bem como após a cada escrita de um novo. Para execução o *framework NUnit* é utilizado. A seguir os resultados da execução dos testes para os contratos escritos nos Códigos fonte 8 e 11 são apresentados. Vale salientar que a abordagem considera correta a implementação dos métodos e dos testes, uma vez que foram desenvolvidos pela equipe oficial do programa. A implementação dos testes encontra-se no Código fonte 12.

Código fonte 12 Testes de unidade do programa.

```
...
[Test]
public void ProvidersList()
{
  DataTable dt = DatabaseFactory.GetAvailableDB();
  Assert.IsNotNull(dt);
  Assert.IsNotEmpty(dt.Rows);
  DataRow[] rows = dt.Select("InvariantName = 'System.Data.SQLite'");
  Assert.IsNotEmpty(rows);
}

[Test]
public void CreateProvider()
{
  DataTable dt = DatabaseFactory.GetAvailableDB();
  foreach (DataRow row in dt.Rows)
  {
    DbProviderFactory factory = DatabaseFactory.Get(row["InvariantName"].
    ToString());
    Assert.IsNotNull(dt);
  }
}
...
```

Antes de serem executados os testes, precisamos entender alguns conceitos determina-

dos pela abordagem. Um deles refere-se à violação do contrato. Antes de tudo, violar um contrato indica que o programa possui alguma restrição com algum requisito. No entanto, para a abordagem a violação do contrato tem o significado inverso, pois esta considera a perspectiva do desenvolvedor e não da implementação do programa. Nesse contexto, o desenvolvedor ao inferir uma suposição no programa espera que seja verdadeira. No entanto, para isto acontecer o teste de unidade não pode falhar ao ser executado, pois caso isto ocorra indica que a suposição do desenvolvedor acerca do programa está errada. Contudo, mesmo com a suposição errada o desenvolvedor pode compreender algo, porém esta é menor, pois não ocorre na sua totalidade. Em contrapartida, quando o teste de unidade não falha indica que a suposição do desenvolvedor está correta, gerando um ganho maior de compreensão.

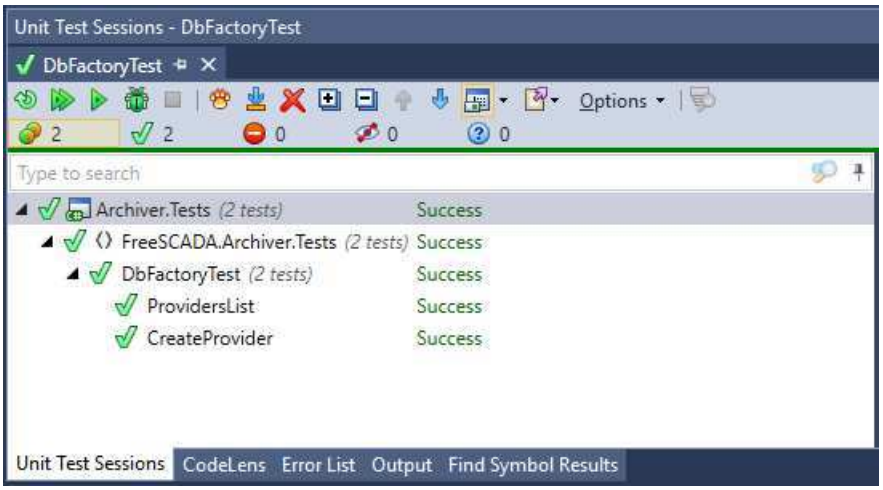
Considerados os conceitos da abordagem, os resultados dos testes são analisados. Assim, para o Código fonte 8 o contrato não foi violado, indicando que não existe uma quarta constante do tipo *ProcessorType*. Todavia, o contrato do Código fonte 11 resulta em uma suposição errada, indicando que a variável *AddressWidth* não é igual a 32 e consequentemente o retorno não é um *x86*. A Figura 3.2 ilustra os resultados dos testes. Por último, referente ao contrato violado, é possível escrever novos contratos para os outros possíveis retornos, desse modo, a avaliação de novos contratos é considerada na Etapa 5.

3.6 Etapa 5: Avaliação da necessidade de novos contratos

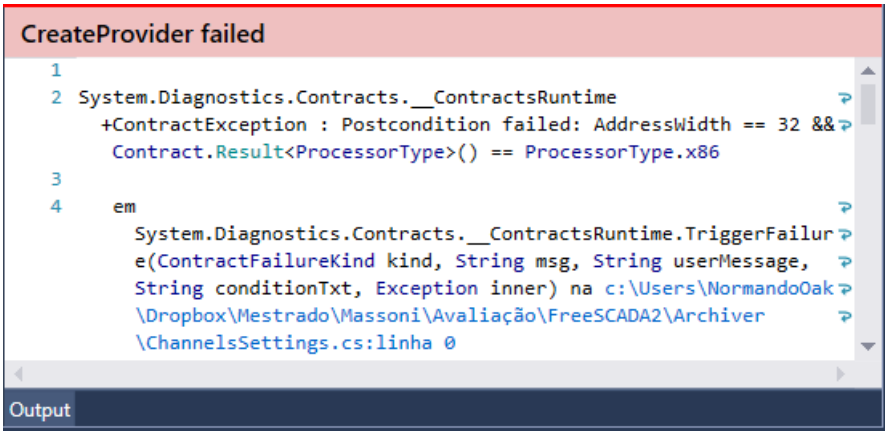
Após executar o teste e verificar o seu resultado, é avaliada a necessidade da escrita de novos contratos. Comumente, contratos violados induzem o desenvolvedor a pensar em novas suposições. As suposições geradas são então transformadas em novos contratos. Dessa maneira, visto que a pós-condição do Código-fonte 11 é violada, devemos escrever uma outra. Refeita a análise e considerando dois outros possíveis valores, desta vez para *Architecture* e para o retorno, sugerimos a escrita da pós-condição do Código fonte 13.

3.7 Compreensão do Método *GetProcessorArch()*

Nesta seção, falamos o que foi compreendido do método *GetProcessorArch()*. Com base nos contratos escritos anteriormente, é pressuposto que duas variáveis determinam o resultado



(a) Resultado dos testes para a pré-condição considerando a inexistência de uma quarta constante do tipo *ProcessorType*.



(b) Pós-condição violada referente a implicação considerando a *AddressWidth* e o retorno do método *GetProcessorArch()*.

Figura 3.2: Execução e resultados dos testes de unidade para uma pré-condição e pós-condição.

do método como base em condições. Nesse sentido, utilizando a variável *AddressWidth* é possível inferir dois possíveis valores (32 e 64) para esta. No entanto, como a pós-condição utilizando o valor 32 foi violada, até aquele momento não havia sido possível compreender o método, nem ao menos a que essa variável referenciava-se. Todavia, ao escrever a pós-condição, considerando a variável *Architecture* com os valores (9 e 6), foi possível confirmar que o método retorna *x64* ou *IPF*. Entretanto, não é possível determinar com certeza qual *if* é verdadeiro.

Pensando nisto, é aconselhável a escrita de contratos separadamente. Com o objetivo de

Código fonte 13 Pós-condição gerada após a violação.

```
1 ...
2 private static ProcessorType GetProcessorArch()
3 {
4   Contract.Ensures(Architecture == 9 && Contract.Result<ProcessorType>()
    == ProcessorType.x64 || Architecture == 6 && Contract.Result<
    ProcessorType>() == ProcessorType.IPF);
5 }
6 ...
```

exemplificar a situação os dois contratos do Código fonte 14 são escritos e verificados. Ao executar os testes, a pós-condição da linha 4 não é violada. Assim, se consolida a compreensão que o método retorna *x64*. Além disso, o fato das variáveis possuírem nomes sugestivos, facilita a compreensão do que elas representam, sendo a largura de endereço do sistema operacional para *AddressWidth* e a arquitetura do processador para *Architecture*. Finalmente, concluímos que o retorno do método não será sempre *x64* devido ao programa em questão considerar o processador do computador no qual é executado. Além do mais, o método retornou x64 pelo fato do processador do computador que está executando o programa ser 64 *bits*. Por fim, o programa pode ser executado em computadores que utilizam processadores *x86*, *x64* e *Itanium*.

Código fonte 14 Pós-condições separadas verificando os possíveis retornos do método.

```
1 ...
2 private static ProcessorType GetProcessorArch()
3 {
4   Contract.Ensures(Architecture == 9 && Contract.Result<ProcessorType>()
    == ProcessorType.x64);
5    Contract.Ensures(Architecture == 6 && Contract.Result<ProcessorType
    >() == ProcessorType.IPF);
6 }
7 ...
```

3.8 Limitações da Abordagem

Essa seção tem por finalidade mostrar quais as limitações da abordagem. São elas:

- A inexperiência com a escrita de contratos pode afetar a compreensão;
- A abordagem só faz uso de pré-condição e pós-condição, sendo desconsideradas as invariantes de classe;
- Contratos podem não ser suficientes para compreender o programa;
- Alguns passos são enfadonhos, desse modo, poderíamos automatizar parte deles;
- Os testes do programa podem não ser confiáveis.

Capítulo 4

Avaliação

Neste capítulo são descritos os métodos avaliativos do estudo empírico realizado neste trabalho, bem como seus resultados. Para isso, utilizamos métodos e observações buscando avaliar a usabilidade da abordagem em relação ao auxílio na compreensão de programas e a facilidade na escrita de contratos. O estudo considera a utilização da abordagem por desenvolvedores de programas que por algum motivo necessitam compreender funcionalidades de métodos ou de uma função provavelmente já escrita por outro desenvolvedor.

4.1 Planejamento e Design do Estudo

Com o intuito de avaliar a usabilidade da abordagem, um estudo foi conduzido com desenvolvedores de software de um centro de pesquisa e desenvolvimento em ciência da computação ligado à Universidade Federal de Campina Grande (UFCG), no qual considera os contratos escritos por eles.

4.1.1 Definição do Estudo

O objetivo do estudo é investigar de forma quantitativa e qualitativa como o impacto da escrita de contratos influencia no entendimento dos métodos do programa. Diante disto, analisamos o comportamento de desenvolvedores em ambientes de trabalho perante a escrita de pré-condição e pós-condição em métodos de classes *C#*, enquanto tentam compreender um método específico. Assim, a pretensão é que desenvolvedores utilizem a abordagem

para compreender programas quando este precisa ser compreendido, como se fosse ser modificado no futuro. Para isto, o presente trabalho busca responder às seguintes questões de pesquisa:

QP1 *A escrita de contratos, de acordo com a abordagem proposta auxilia os desenvolvedores a compreender o método?* O objetivo é verificar se os desenvolvedores compreendem o que o método faz enquanto escrevem contratos.

QP2 *A necessidade de compreender um programa favorece a escrita de contratos?* Aqui o objetivo é avaliar se os passos descritos pela abordagem favorecem a escrita de contratos nos métodos.

4.1.2 Contexto do Estudo

O estudo consiste na escrita de contratos para métodos de uma classe *C#* extraída de um projeto *open-source*. Os contratos são escritos por participantes que atuam como desenvolvedores e pesquisadores de um centro de pesquisa e desenvolvimento da Universidade Federal de Campina Grande (UFCG). Os métodos utilizados no estudo são da solução *Free SCADA*2[1], que disponibiliza aos usuários ferramentas para visualização e controle interativo de processos industriais. A escolha da solução decorre de ser desenvolvida com tecnologia *Microsoft* e possuir testes, pois a escrita dos contratos ocorre por meio do uso da biblioteca *Code Contracts*, sendo esta utilizada pelos participantes.

A solução supracitada contém 20 projetos. Entretanto, para o estudo apenas dois são selecionados, (i) Archiever e (ii) Archiever.Tests, que representam uma classe extraída de cada projeto[2]. O primeiro contém arquivos que definem comportamentos, recursos e *layout* de tela. O segundo contém testes. No projeto (i) treze arquivos são descritos, sendo um para definição de constantes, um para *design* de tela e onze são classes *C#*. Dentre as classes, foi selecionada a *DatabaseFactory*, caracterizada pela ausência de comentários de código que pudessem auxiliar na compreensão. Essa classe descreve métodos que consultam instâncias de banco de dados, e a arquitetura do processador do computador que está executando o

[1] https://sourceforge.net/projects/free-scada/

[2] As classes encontram-se em https://www.dropbox.com/sh/6yn8zj8hkoshdrw/AABWrML8MUEc6mwy8Qf-Slua?dl=0

programa. No projeto (ii) todos os testes de unidade são selecionados. A maior motivação para escolha destes projetos ocorre devido a não possuírem erros que impeçam a sua *build*. A Tabela 4.1 detalha as características da classe utilizada no estudo[3].

Como forma de verificar a conformidade do código com as especificações declaradas, a classe *DbFactoryTest* representa os testes de unidade implementados pela equipe oficial do programa. A utilização dessa é justificada por conter a implementação dos testes de unidade para os métodos presentes em *DatabaseFactory*. É importante destacar que a abordagem depende dos testes de unidade definidos no programa para que os contratos escritos pelos desenvolvedores sejam validados. Por fim, os contratos não devem ser escritos em métodos que não são implementados pelo programa.

Tabela 4.1: Classe *C#* utilizada como unidade experimental do estudo

Solução	Projeto	Classe	Métodos	Linhas de Código
Free SCADA	Archiver	DatabaseFactory	3	90

O estudo empírico é viável para ser concluído em tempo relativamente apto pelos participantes, uma vez que a unidade experimental possui poucos métodos definidos. No total são 90 linhas de código, considerando quebras de linha e aberturas e fechamentos de chaves. O estudo não estabelece limite de tempo, devido a alguns participantes não terem familiaridade com os princípios de *DbC*, e por considerar que esse fator pode causar desconforto quando estipulado um limite. O centro de *P&D* de onde vem a amostra possui parcerias com grandes empresas do segmento industrial tecnológico, o que facilita o contato com desenvolvedores e pesquisadores atuantes na área. No total 12 pessoas participaram do experimento, cujo perfil está descrito na Tabela 4.2. Destes, dez são desenvolvedores em projetos de pesquisa e dois são pesquisadores. Do total de participantes, sete possuem experiência com a linguagem de programação C#. No entanto, apenas dois possuem conhecimento em *DbC*.

[3] A implementação da classe contendo as variáveis globais e métodos está disponível no Apêndice A.

Tabela 4.2: Experiência dos participantes do estudo

Tempo[1] de Exp. com Desenvolvimento	Participantes (#[2])	#[2] Experiência com C#?	#[2] Experiência com Contratos?
$1 \leq 2$	2	Sim (2)	Não (2)
$2 \leq 3$	2	Não (2)	Sim (1) – Não (1)
$3 \leq 4$	3	Sim (1) – Não (2)	Não (3)
> 4	5	Sim (4) – Não (1)	Sim (1) – Não (4)
$\sum$	12	Sim (7) – Não (5)	Sim (2) – Não (10)

[1] Tempo em anos
[2] # Valor absoluto

4.1.3 Procedimento Experimental

O experimento executado ocorre no ambiente de trabalho dos desenvolvedores, de forma individual, sendo cedido pelo condutor do estudo o *notebook* para coleta e monitoramento das atividades. A coleta e análise dos dados é feita em um *notebook* contendo processador *Intel® Core™ i7-2675QM* 2.2 *GHz* com 10 *GB* de memória *RAM* e *SSD* de 240 *GB* executando o sistema operacional *Windows 10 Pro*. O projeto utilizado no experimento é o *Free SCADA2*. Este contêm a classe e os testes necessários para o estudo, estando disponível para download[4]. Cada participante recebe as (i) mesmas classes *C#*, (ii) instruções e materiais para realização do experimento e; (iii) um questionário pós-experimento.

Antes de iniciar o experimento, uma apresentação é ministrada para cada participante. A apresentação tem duração máxima de 15 minutos e é feita por meio de slides focando nos conceitos de pré-condição e pós-condição e no uso dos métodos do *Code Contracts*. Após ministrada a apresentação, é solicitada a cada participante a leitura da abordagem[5]. descrita no arquivo .pdf, para em seguida serem retiradas as dúvidas. Compreendida a abordagem, os participantes podem iniciar as atividades utilizando a *IDE Visual Studio*, juntamente com o *plugin* do *Code Contracts*. Durante o experimento o arquivo .pdf pode ser utilizado, bem

[4] https://sourceforge.net/projects/free-scada/files/

[5] A abordagem encontram-se em https://www.dropbox.com/sh/6yn8zj8hkoshdrw/AABWrML8MUE-c6mwy8Qf-Slua?dl=0

como a internet, para consultar a *API* do *Code Contracts* e outras funções que desejem. O objetivo definido é compreender os três métodos presentes na classe *DatabaseFactory*. Dito isto, o participante é responsável por escolher o método que deseja iniciar.

Nas instruções passadas, são ressaltadas a importância em seguir os passos descritos pela abordagem. Nesse contexto, aberto o projeto, os participantes têm que analisar as chamadas do método, analisar a assinatura, escrever contratos e executar os testes de unidade. É importante destacar a necessidade da execução do teste, pois é nessa etapa que o contrato é validado. Nesse sentido, o participante adquire conhecimento à medida que o contrato não é violado. Isto pode ser identificado ao visualizar o símbolo circular na cor verde contendo na frente a mensagem *Success*. Consequentemente se o símbolo exibido representar o sinal negativo (-) na cor vermelha contendo na frente a mensagem que descreve o erro, o conhecimento adquirido é menor pois não ocorre na sua totalidade. A Figura 4.1 ilustra esse processo.

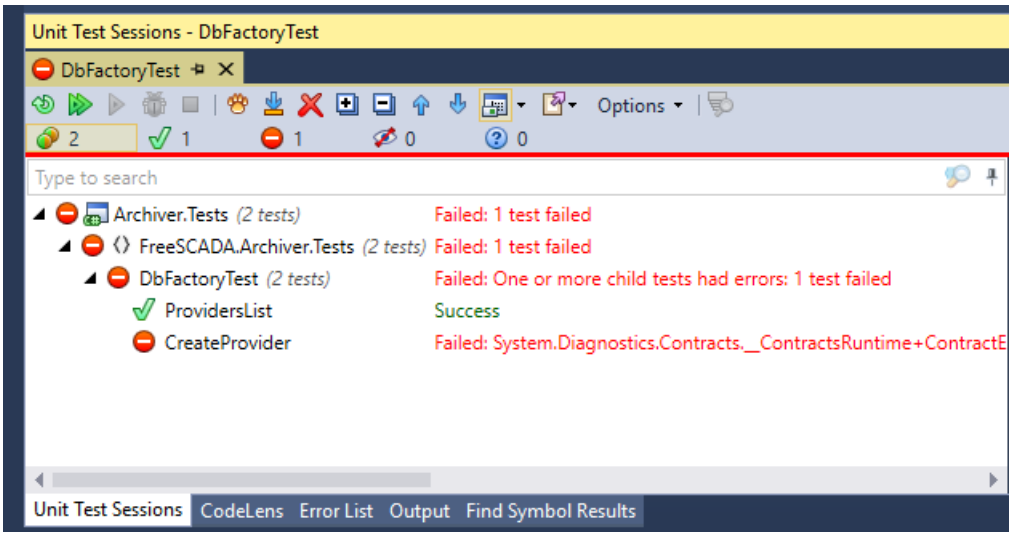


Figura 4.1: Execução dos testes de unidade para um contrato não violado e outro violado.

Em relação a abordagem foi realizado um estudo piloto envolvendo 3 participantes com o objetivo de refinar as instruções dos passos que descrevem as ações dos desenvolvedores. Referente a essas ações, a abordagem passou a destacar explicitamente que uma pós-condição pode ser escrita independente de uma pré-condição. A partir disso, no arquivo *pdf* que descreve a abordagem essa informação foi detalhada logo no seu início. Além do mais, uma outra informação, desta vez fazendo referência à transformação de variáveis locais de méto-

dos em variáveis globais foi introduzida na abordagem, pois no piloto esta era inexistente. Por fim, em relação aos passos que descrevem a assinatura dos métodos, a escrita de pré-condição, e a implicação de uma pós-condição decorrente de uma pré-condição não foram alterados.

Para auxiliar a coleta de dados, é utilizado o protocolo *Think Aloud* [41]. Esse ajuda na compreensão das ações dos participantes no momento que captura os seus pensamentos. Desse modo, os participantes devem verbalizar os pensamentos e ações em relação aos objetivos e expectativas, enquanto escrevem contratos, e executam os testes dos métodos da classe.

De modo complementar aos dados, a tela do computador dos participantes é gravada por um software chamado *CamStudio Recorder*, versão 2.7. Os discursos, incluindo o *Think Aloud*, também são gravados por meio do uso de um microfone. Anotações acerca do comportamento dos participantes são feitas pelo condutor do estudo, além de perguntas com o objetivo de compreender porque os contratos estão sendo escritos. Após a finalização desse, cada participante recebe um questionário[6] web. Esse contêm perguntas referentes ao código da classe *DatabaseFactory* e informações referentes ao participante e à abordagem proposta por este trabalho. A respeito do código da classe, quatro perguntas são elaboradas para os três métodos. A primeira refere-se a informações do computador que detalha o processador e o sistema operacional deste. A segunda objetiva compreender o comportamento do método que envolve o gerenciamento do banco de dados. Para a terceira, é questionado qual a representação de uma variável no método que tem como finalidade definir coordenadas. Em relação à quarta, essa destaca o fluxo do método considerando as condições determinadas. Os demais questionamentos referem-se à experiência do participante quanto: ao tempo de desenvolvimento e o uso da linguagem *C#*, bem como o nível de esforço para a escrita de contratos. Por último, as demais questionam os benefícios, a ausência de passos, possíveis melhorias e a aplicabilidade da abordagem. Tendo em vista capturar as respostas, o questionário para o código da classe define questões objetivas, apresentando assim opções de múltipla escolha. Para as informações do participante e da abordagem esse contém opções objetivas e subjetivas, sendo utilizados na análise dos dados.

Ainda são detalhados os softwares e versões necessários para a execução do estudo. A

[6]https://goo.gl/forms/OdGxI3YgKZQvtY9c2

IDE utilizada para manipulação do projeto é o *Visual Studio Ultimate 2013*. A ferramenta que checa os contratos é o *Code Contracts* versão 1.9.10714.2. O projeto utiliza a versão 3.5 do *.NET Framework*. Nesse sentido, é necessário adicionar a biblioteca *Microsoft.Contracts* para ter acesso aos métodos de escrita de contratos. A execução dos testes de unidade ficou por parte do *NUnit* versão 2.0.2327.4786.

4.1.4 Método da Pesquisa

Ao término do estudo, cada participante produz quatro artefatos:

- o conjunto de contratos escritos na classe *DatabaseFactory*, com eventuais contratos comentados ou comentários no código por parte dos participantes, por exemplo, acrescentando comentários que identificam as chamadas para métodos internos e externos do programa;
- o áudio e o vídeo contendo os passos realizados durante o estudo;
- os questionários referentes ao código e experiência do participante;
- e por fim, anotações contendo o pensamento do condutor do estudo.

De modo a responder a QP1 e QP2, são analisados as anotações do protocolo *Think Aloud*, as gravação do estudo (áudio e vídeo) e os questionários pós-experimento. Referente ao questionário, as quatro primeiras perguntas atendem a QP1: para os três métodos presentes na classe, são inspecionadas as ações realizadas por cada participante, observando se a metodologia utilizada está de acordo com a proposta. Assim, são levantadas novas análises para determinar até que ponto a abordagem auxilia a compreensão do código.

Para QP2, são consideradas as perguntas que visam identificar a experiência do usuário em relação ao tempo de desenvolvimento e ao conhecimento dos princípios do *DbC*, bem como ao nível de uso da abordagem.

Diante disto, o estudo utiliza da seguinte metodologia. Primeiro, as anotações dos participantes são lidas individualmente. Em seguida, o áudio e o vídeo são reavaliados. A reavaliação dos dados obtidos por estes conteúdos, ocorre devido à necessidade em capturar pontos relevantes que, por alguma razão, possam ter passado despercebidos durante o estudo. Dessa maneira, a medida que o áudio e o vídeo são reavaliados, novas anotações são

feitas e revisadas por um segundo pesquisador. Ao finalizar as novas anotações, é possível identificar pontos em comum entre os participantes. Nesse sentido, é necessário estruturar as informações em três categorias. São elas: (i) áudio e vídeo; (ii) contratos definidos por métodos; e (iii) captura por meio do protocolo *Think-Aloud*.

Para a categoria (i), são destacados os pontos relevantes e o fluxo utilizado pelo participante para compreender os três métodos da classe. Como forma de exemplificar alguns desses pontos relevantes, são descritas transformações e identificações de determinados comportamentos no método. Por exemplo, a transformação de variáveis internas do método *GetProcessorArch()* em globais, a identificação do uso de duas chamadas definidas pelo programa e o uso de variáveis globais no método *Get(...)*. Em seguida, a categoria (ii), expõe os contratos definidos em cada um dos três métodos da classe. Estes, por sua vez, auxiliam em parte da compreensão, considerando o tipo de contrato escrito. Por último, a utilização do protocolo *Think Aloud*, destacado pela categoria (iii), avalia as ações dos participantes e as percepções do condutor do estudo. As anotações referentes a cada participante encontram-se disponibilizadas em um repositório online[7].

Por fim, considerada concluída a estruturação das informações, são identificados passos em comum entre os participantes, categorizando as principais conclusões de acordo com cada questionamento. Assim para QP1, os passos em comum observados são: pesquisa na internet feitas pelos participantes durante a execução; consulta e comentários da abordagem; identificação de variáveis e métodos; execução e análise dos testes; e avaliação do fluxo do código pelo participante. Logo, para QP2 os passos são: pesquisa na internet feitas pelos participantes durante a execução; consulta e comentários da abordagem e da apresentação pré-estudo; identificação de variáveis e métodos; comentários dos contratos escritos e da sintaxe do *Code Contracts*; e avaliação do fluxo do código pelo participante.

4.2 Resultados e Discussão

A seguir, os resultados do estudo são apresentados, com o objetivo de responder às questões de pesquisa formuladas na Seção 4.1.1. Inicialmente, para QP1-QP2, são discutidos os passos em comum entre os participantes, seguido da análise dos questionários. Os passos em

[7]https://www.dropbox.com/sh/6yn8zj8hkoshdrw/AABWrML8MUE-c6mwy8Qf-Slua?dl=0

comum e o resultado da análise dos questionários estão divididos conforme são considerados pertinentes a cada questão. Por fim, as observações do condutor do estudo são destacadas.

QP1: Compreensão do Código

Para resposta da QP1, baseamo-nos nos resumos de observação do protocolo *Think Aloud*, do áudio, do vídeo e das respostas dos questionários. A seguir, são relatados os resumos das observações e as respostas do questionário. A Tabela 4.3 apresenta resultado de respostas para perguntas objetivas.

Pesquisa na internet feitas pelos participantes durante a execução

Para pesquisas na internet, são esperadas buscas referentes às *API*s do *Code Contracts* e do *C#*. A finalidade destas é compreender particularidades da ferramenta e da linguagem. Diante disto, é perceptível a busca de informações pelos participantes no momento que sentem dificuldades. Nesse sentido, pesquisas visando compreender o significado de trechos do código são destacadas. Três participantes pesquisam informações a respeito da classe *DbProviderFactory*, devido o método *Get(...)* retornar este tipo. Ao efetuarem as pesquisas, os participantes encontram em um *site*[8], informações de como obter um *DbProviderFactory*. No site é possível encontrar informações de como recuperar um provedor de dados. Nas informações passadas, existe uma tabela descrevendo as colunas retornadas pela estrutura de dados *DataTable*. Tal estrutura é denominada *DbProviderFactories*.

O método *Get(...)* contém na sua assinatura a variável *invariantName*. O Código fonte 6 descreve a implementação do método. É importante destacar que esta possui o mesmo nome descrito na tabela do *site*. Como forma de explicar o seu uso, é destacada a definição da variável *dt* no escopo do método *Get(...)*. Tal variável, é utilizada para retornar uma consulta que possibilita o usuário selecionar um *DataRow* em tempo de execução. O *DataRow* selecionado é repassado para o método *GetFactory(...)*. O *GetFactory()*, por sua vez, é um método estático presente na classe *DbProviderFactories* que retorna um *DbProviderFactory*. Por fim, o participante ao adquirir este conhecimento, define contratos verificando se o comportamento de determinado fluxo do código é válido.

Em outro caso, um participante alega não ter conhecimento de informações mais gerais,

[8]https://msdn.microsoft.com/pt-br/library/dd0w4a2z(v=vs.110).aspx

Tabela 4.3: Questões referentes ao código da classe DatabaseFactory

Total de Participantes	Questão	Alternativa	Resultado (%)
12	Qual a função do *GetProcessorArch()*?	a	58.33
		b	16.66
		c	25
		d	0
		e	0
	Qual a função do *GetAvailableDB()*?	a	8.33
		b	0
		c	58.33
		d	16.66
		e	16.66
	O que representa a variável *invariantName*?	a	8.33
		b	8.33
		c	50
		d	25
		e	8.33
	Qual o fluxo do *Get()*?	a	50
		b	50
		c	0
		d	0
		e	0

a exemplo da existência de processadores com arquiteturas diferentes. Nesse caso, duas pesquisas são realizadas. A primeira busca o significado da palavra *processor*, de modo a confirmar sua tradução. Enquanto a segunda, busca a existência de processadores com arquiteturas diferentes. Ambas as pesquisas, possibilitam o participante a pensar em um possível contrato. Este, validaria o lançamento da exceção do método *GetProcessorArch()*. No entanto, tal contrato não foi escrito, devido ao participante desconhecer quando a exceção do método deveria ser lançada. Por último, é importante ressaltar que a palavra *processor*

faz referencia ao conhecimento acerca da regra de negócio.

Consulta e comentário da abordagem

Os resultados indicam que o uso da abordagem auxilia a compreensão. Com base nas observações do estudo, todos os participantes recorrem ao arquivo que descreve a abordagem para identificar quais passos devem realizar. Como forma de validação da abordagem, diversos comentários são capturados por áudio. Dentre eles, é importante destacar a relevância da abordagem, em realizar inferência e avaliações lógicas sobre o funcionamento do código, além da possibilidade em validar hipóteses e predições do funcionamento e da relação entre as entidades.

Em outros comentários registrados, cinco participantes destacam a facilidade ao compreender o código, uma vez que, escreve primeiro contratos para os métodos mais internos do programa, para em seguida, escrever contratos para os métodos mais externos. Segundo eles, a facilidade na compreensão surge, devido a abordagem tratar partes específicas do código isoladamente. Assim, os contratos escritos nos métodos, asseguram que seu comportamento não sofra qualquer tentativa de modificação. Do contrário, o participante seria alertado, mudando assim a sua compreensão. Desse modo, escrever contratos para compreender o código, garante que o comportamento seja exatamente o pressuposto.

Por fim, o processo de compreensão também ocorre devido à abordagem estimular a escrita de contratos. Assim, ao serem analisados os trechos do código, os fragmentos considerados importantes são armazenados mentalmente. Esses fragmentos são as chamadas definidas no escopo do método, as variáveis globais, os parâmetros do método, os valores de retorno, dentre outros. Dessa forma, o participante inicia o processo de compreensão ao pensar em como escrever os contratos. Nesse sentido, basta surgir hipóteses de novas verificações para que novos contratos sejam escritos.

Identificação de variáveis e métodos

A identificação de variáveis e métodos faz parte dos pontos importantes do estudo. A partir da análise dos resultados, é possível identificar a utilização da abordagem com frequência. Para esse cenário, o participante busca compreender o que deve fazer ao identificar métodos e variáveis do programa. Nesse sentido, os participantes são livres para realizarem as verificações que julguem necessárias. Em particular, um dos participantes tentou alterar os

parâmetros *AddressWidth* e *Architecture* para lançar uma exceção. Porém, percebeu que não faria sentido alterar tais parâmetros apenas para verificar se a exceção realmente seria lançada. A observação é identificada por meio do protocolo *Think Aloud*.

Em outros casos, alguns participantes utilizam a abordagem para confirmar apenas uma das rotinas do código. Apesar de correto, isto pode se tornar um problema. Em métodos com retornos diversificados é visível o participante tentar compreender apenas o trecho que considera viável. A viabilidade ocorre de acordo com o grau de dificuldade em escrever o contrato. Em um dos casos, o participante confirmou a rotina do método *GetProcessorArch()* depois de ter escrito o contrato presente no Código fonte 15 e ter checado nas configurações do sistema do computador, se o retorno realmente correspondia ao informado. Para este caso ele teve êxito. Porém, para métodos não compreendidos completamente, falsas impressões de compreensão podem surgir.

Código fonte 15 Contrato verificando o retorno do método *GetProcessorArch()*.

```
1 ...
2 private static ProcessorType GetProcessorArch()
3 {
4   Contract.Ensures(Contract.Result<ProcessorType>() == ProcessorType.
      x64);
5   ...
6 }
```

No geral, diversos fatores influenciam a compreensão do código. No entanto, outros dois casos merecem destaque. O primeiro destaca quatro participantes associando o nome do método *GetAvailableDb()* a possíveis funcionalidades. Por ter um nome sugestivo fazendo alusão a banco de dados, comentários como sendo uma instância ou várias tabelas foram mencionados. No entanto, apesar de 58,33% dos participantes acertarem a questão, 91,67% desconhecem o que representa a variável *SQLiteName* utilizada no método, se considerarmos a soma dos resultados dos participantes que erraram a questão três. Isto porque a questão pergunta o que representa a variável *invariantName* utilizada no *Get(...)*, que por sua vez é comparada com *SQLiteName*. Para o segundo fator, é destacado o comportamento adotado por cinco participantes. Em especial, estes avaliaram o método *Path.Combine(...)* da *API* do *C#*. Um deles, disse saber o que o método faz por tê-lo utilizado em um projeto no

passado. Apesar de métodos externos não serem considerados pelo estudo, foi percebida a importância que estes podem apresentar para compreensão do código.

Execução e análise dos testes

A execução e análise dos resultados dos testes são pontos em comum entre os participantes. Estes auxiliam na confirmação de suposições. No entanto, é observado que alguns participantes efetuam a leitura dos testes de unidade. A leitura destes auxilia na compreensão do trecho do código, uma vez que expressa o comportamento esperado pelos desenvolvedores do programa. Nesse sentido, há indícios de que testes de unidade são ótimas fontes de informação, porém nem sempre são escritos.

Código fonte 16 Teste de unidade *ProvidersList()*.

```
1  ...
2  [Test]
3  public void ProvidersList()
4  {
5    DataTable dt = DatabaseFactory.GetAvailableDB();
6    Assert.IsNotNull(dt);
7    Assert.IsNotEmpty(dt.Rows);
8
9    DataRow[] rows = dt.Select("InvariantName = 'System.Data.SQLite'");
10   Assert.IsNotEmpty(rows);
11 }
12 ...
```

Ao efetuar a leitura do teste *ProvidersList()* descrito no Código fonte 16, um participante comenta que o retorno do método *GetAvailableDB()* não pode ser nulo e vazio. O comentário é feito com base nas linhas 6-7. Sendo assim, é possível afirmar que um *data table* sempre é inicializado. Como forma de validar a informação, o contrato especificado pelo participante no Código fonte 17 reforça a afirmação.

Outro caso a respeito da leitura dos testes chama a atenção. Desta vez, a presença da *string InvariantName* nos testes de unidade é identificada pelo participante. Com dúvidas do que representa a palavra, ele decide escrever o contrato do Código fonte 18. O contrato escrito supõe um possível valor presente no *data table*. Executado o teste e confirmado

Código fonte 17 Contrato verificando o retorno do método *GetAvailableDB()*.

```
1 ...
2 public static DataTable GetAvailableDB()
3 {
4   Contract.Ensures(Contract.Result<DataTable>().Rows.Count > 0);
5   ...
6 }
```

a presença, o participante tem a certeza que o valor "*System.Data.SQLite*"é adicionado ao provedor de dados. Assim, o nome da variável ajuda a escrever o contrato, que por sua vez ajuda a compreender a rotina do método.

Código fonte 18 Contrato verificando o valor de *InvariantName*.

```
1 ...
2 public static DataTable GetAvailableDB()
3 {
4     ...
5   Contract.Ensures(Contract.Result<DataTable>().Rows[Contract.Result<
    DataTable>().Rows.Count - 1]["InvariantName"].ToString() ==
    SQLiteName);
6   ...
7 }
```

Por outro lado, em alguns casos, a escrita de contrato não foi seguida pela execução dos testes. A inexperiência com o uso da abordagem pode ser a causa. Nesses casos, a compreensão pode ser afetada devido à escrita de mais de um contrato ocorrer sem a execução dos testes. Assim, o participante ao definir dois contratos seguidos, e não executar os testes, pode supor um comportamento errôneo acerca do código. Isto acontece com quatro dos 12 participantes. Ilustrando o caso, os participantes perguntaram ao condutor do estudo se ambos os contratos foram violados. Sugerido a leitura do *log* de erro, os participantes percebem que os contratos definidos após o contrato violado não são verificados. Sendo assim, para participantes inexperientes com as ferramentas *Code Contracts* e *NUnit*, isto pode dificultar o entendimento, pois parece que ambos os contratos foram verificados.

Fluxo do participante no código

O fluxo dos métodos revela como a abordagem é utilizada. No geral, os 12 participantes iniciam o estudo lendo o código. A leitura é superficial na maioria dos casos. No entanto, aponta o início da compreensão. A seguir, a Tabela 4.4 enumera os participantes de acordo com as escolhas dos métodos. Nela é possível deduzir que 75% dos participantes iniciam a compreensão do código pelo método *Get(...)*, enquanto 25% iniciam pelo método *GetAvailableDB()*. Os participantes dão início pelo Get(...) ao perceberem que o método realiza a chamada para os outros dois definidos pelo programa. Em contrapartida os que iniciam pelo *GetAvailableDB()* classifica-o como sendo o mais simples por possuir a menor quantidade de instruções.

Tabela 4.4: Quantidade de participantes que iniciam pelos métodos

Total de participantes	Método Get(...)	Método GetAvailabeDB()	Método GetProcessorArch()
12	9	3	0

As escolhas dos participantes variam de acordo com alguns fatores. Entre eles estão a familiaridade com trechos dos métodos, o tamanho, a quantidade de chamadas internas, a assinatura e a ordem implementada na classe. Isto pode ser observado com base nos comentários do *Think Aloud*. Dos 100% que avaliam o método *Get(...)*, 58,33% identificam as chamadas internas. Desses, 41,66% representam o percetual dos que iniciam pelo *Get(...)*. Os outros 16,66% representam os que iniciam pelo *GetAvailableDB()*. Todavia, a identificação ocorre para os demais após a releitura da abordagem. Em um dos casos, o participante identifica as chamadas após comentar o código, classificando-as em internas e externas. Para os outros, o condutor do estudo perguntou se existem chamadas internas do programa definidas no método.

Os resultados apontam indícios da eficácia da abordagem. Ao analisar o método *Get(...)*, um participante comenta a possibilidade do retorno ser nulo em duas ocasiões. Entretanto, uma delas não ocorre. Analisando o escopo do *else{...}*, descrito no Código fonte 19, é possível observar, na linha 10, a condição verificando se a quantidade de linhas é igual a zero. De acordo com o participante, a pós-condição escrita no Código fonte 20 garante que

o retorno do método *GetAvailableDB()* não é nulo. Desse modo, a compreensão de código escrevendo contratos para os métodos do programa fortifica o uso da abordagem.

Código fonte 19 Implementação do *else{...}*.

```
1  ...
2  public static DbProviderFactory Get(string invariantName)
3  {
4      ...
5      else
6    {
7      DataTable dt = GetAvailableDB();
8      DataRow[] rows = dt.Select(string.Format("InvariantName = '{0}'",
       invariantName));
9
10     if (rows.Length == 0)
11       return null;
12
13     return DbProviderFactories.GetFactory(rows[0]);
14   }
15 }
16 ...
```

Por fim, é compreendido que tanto a escrita de alguns tipos de contratos, a exemplo de pós-condição, quanto o fluxo dos métodos, podem acelerar o processo de compreensão. No entanto, na abordagem considerada por este estudo, a compreensão só ocorre em sua totalidade. Assim, é necessário compreender primeiro as declarações dos métodos que são utilizados no escopo e foram definidos pelo programa. Dessa maneira, seria um equívoco associar a compreensão ao fluxo dos métodos seguidos pelos participantes, uma vez que esta independe de onde se inicia. Por último, a velocidade que os participantes compreendem o código não faz parte da avaliação deste estudo.

Discussão sobre as respostas do questionário

Considerando as respostas para QP1 do questionário, é possível identificar que as questões com maior quantidade de acertos foram a 1 e a 2. Isto ocorre devido as questões envolverem conceitos específicos a qual o programa se aplica. Nesse sentido, a Questão 1 indaga que

Código fonte 20 Pós-condição verificando o retorno do método *GetAvailableDB()*.

```
1 ...
2 public static DataTable GetAvailableDB()
3 {
4     Contract.Ensures(Contract.Result<DataTable>() != null && Contract.
    Result<DataTable>().Rows.Contains(SQLiteName));
5   ...
6 }
```

conceitos são relacionados às variáveis *AddressWidth* e *Architecture* utilizadas como condicionais no método. Desse modo, as variáveis por possuírem nomes sugestivos fazem com que os participantes deduzam o que representam. No mesmo entendimento, a Questão 2 favorece a compreensão por possuir nome sugestivo para a declaração do método, estando esse associado a estruturas de bancos de dados. Contudo, as confirmações ocorrem ao escreverem contratos.

Observando a Questão 3, os resultados indicam o maior número de respostas erradas. Tal fato tem relação com a falta de conhecimento de conceitos técnicos de bancos de dados, bem como a implementação desses em programas. Considerando a condição de igualdade entre a variável *invariantName* e *SQLiteName* no método *Get(...)*, os participantes não identificaram a relação que as duas possuem. Nessa lógica, no método *GetAvailableDB()* a variável *SQLiteName* é utilizada pela instrução *dt.Rows.Add()*. Assim, ao ler a palavra *Add* alguns acharam que esta fazia referência à inserção de uma linha no banco de dados. Apesar de acharem isso, tal suposição poderia ser confirmada com a escrita de contratos, entretanto, para adquirir esse conhecimento é necessário aplicar instruções de banco de dados nas condições dos contratos.

Por último, a Questão 4 apresenta uma porcentagem de acerto médio, devido a possuir várias instruções contidas no método. Desse maneira, algumas não foram verificadas pelos participantes, porque envolviam o uso de *APIs* externas, sendo estas desconsideradas pela abordagem. Em compensação, a outra metade que acertou a questão procurou compreender os métodos que não foram definidos pelo programa. Ainda para estes, alguns observaram a existência de variáveis de método recebendo o valor retornado por *APIs* externas. Pensando nisso as transformaram em variáveis globais e escreveram contratos para essas.

QP2: Escrita de Contratos

Pesquisa na internet

Nesse ponto, são relatadas e discutidas as pesquisas na internet realizadas pelos participantes. De acordo com os resultados, todos os 12 participantes realizam buscas. As buscas são variadas. Entre elas, a maioria faz referência à escrita de contratos. No entanto, algumas ocorrem para a *API* do *C#*. Além disso, algumas possuem o mesmo objetivo de outras, por este motivo, a seguir são destacados apenas alguns exemplos realizadas no estudo:

- "*Code contracts add row*";
- "*Code contracts string requires*";
- "*Code Contracts syntax*";
- "*Typeof C#*";
- "*C# rows count*";
- "Como checar a quantidade de linhas";
- "Como checar se a *string* é um inteiro".

Como mencionado em tópicos anteriores, as pesquisas sobre contratos foram motivadas pela tarefa de compreender. Como base nessa afirmação, é observado que os exemplos encontrados na internet, auxiliam os participantes a escreverem as suas validações. Em especial, são encontrados nos sites da *Microsoft* e no fórum *Stackoverflow*, exemplos de pré-condição e pós-condição que são adaptados pelos participantes.

Por fim, os exemplos destacados acima, objetivam: verificar se uma linha é realmente adicionada; definir ocasiões de pré-condições para o parâmetro passado em *invariantName*; analisar a sintaxe da biblioteca possibilitando o entendimento para escrita de contratos; como verificar o tipo de retorno do método; procurar exemplos de contratos escritos com "*Rows.Count*"; procurar possibilidades de verificação para a quantidade de linhas; e identificar como uma *string* pode ser um inteiro.

Consultas e comentários da abordagem e da apresentação pré-estudo

Para este ponto, são descritos o uso da abordagem e da apresentação que antecedeu o estudo, bem como os comentários dos participantes acerca dos artefatos. Neste sentido, é destacado que a apresentação serve como suporte para os conceitos de pré-condição e pós-condição, enquanto que a abordagem, serve para facilitar a escrita de contratos à medida que é consultada. Sendo assim, os participantes precisam ficar atentos aos passos descritos pela abordagem que considera a escrita de pré-condição e pós-condição para o retorno, o parâmetros e as variáveis globais.

Apesar dos participantes possuírem acesso a internet durante todo o estudo, os exemplos descritos na apresentação motivam a escrita de alguns tipos de contratos. Um deles comenta a necessidade de a apresentação conter mais exemplos de pré-condição e pós-condição. Considerado desnecessário, pois o intuito é mostrar como contratos são escritos, as observações mostram a importância da apresentação para a escrita de contratos. Assim, sem ela o estudo se tornaria inviável, uma vez que demandaria mais esforço para a escrita.

Neste caso, a afirmação pode ser confirmada com base em uma das perguntas do questionário submetido aos participantes. De acordo com os resultados, o nível de esforço foi classificado igualmente entre leve, moderado e intenso. O histograma 4.2 ilustra as respostas.

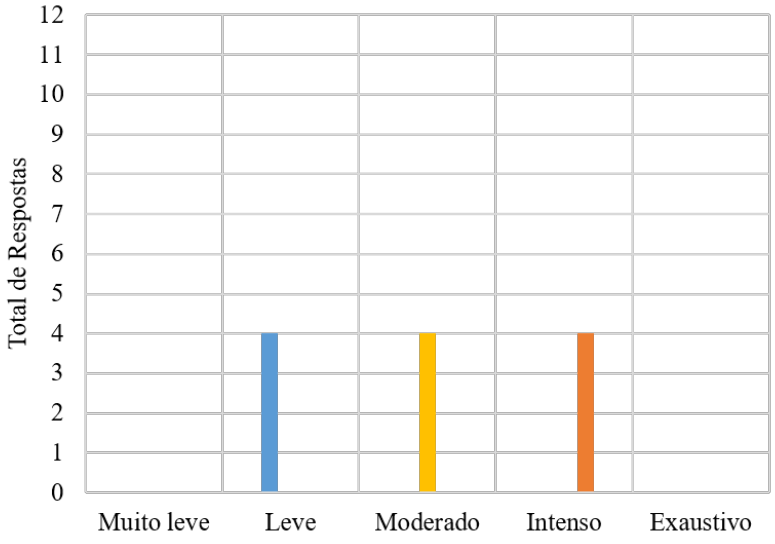


Figura 4.2: Nível de esforço dedicado para escrita de contratos.

Analisando as justificativas das respostas, dos quatro participantes que classificam como

leve, dois consideram simples (i) a escrita de condições para verificações do sistema; e (ii) a especificação de condições acerca do funcionamento do método de modo a compreender o retorno e as entradas. Entre os outros dois, um declara que (iii) são necessários poucos contratos em relação ao código estudado; enquanto o outro (iv) menciona a existência de apenas três métodos na classe, não tendo que avaliar chamadas de bibliotecas externas. Considerando os que têm esforço moderado, os quatro relatam dificuldades com a sintaxe do *Code Contracts*. Além disso, incluem (i) a inexperiência com a linguagem *C#*; e (ii) a dificuldade com a manipulação lógica. Por fim, vale ressaltar dois comentários: (iii) a sintaxe utilizada para a criação de contratos não é intuitiva o suficiente a ponto de permitir a escrita de contratos sem a necessidade de consulta à documentação; e (iv) levando em consideração que nunca havia escrito contrato, o participante declara que apenas recebendo a explicação breve do pré-estudo, ele entende a motivação do seu uso, conseguindo escrever para alguns casos, contratos. Porém, se tivesse um conhecimento maior em relação à sintaxe e à ferramenta, a atividade poderia ser executada com mais facilidade. Para os que classificam como intenso, os quatro associam a dificuldade em formular contratos com a inexperiência em *DbC*, *C#* e *Code Contracts*. Por último, um destaca a preocupação em compreender as rotinas dos métodos.

Como esperado, ao consultar a abordagem, os participantes conseguem escrever pré-condição e pós-condição nos métodos. A confirmação ocorre devido aos contratos definidos. No entanto, para alguns métodos são desconsiderados a definição de pré-condições após consultas a abordagem. Contudo, mais uma rotina de método é observada, dessa vez, os participantes garantem a validação do *GetProcessorArch()* com a pós-condição descrita no Código fonte 21.

Outro ponto a ser destacado é o fato da abordagem minimizar a escrita de contratos. A diminuição ocorre devido ao participante adquirir conhecimento à medida que analisa as chamadas internas do método. Assim, contratos que validam rotinas repetidas podem ser descartados. Até mesmo o fato da abordagem permitir a criação de premissas sobre a execução do código, possibilita o participante a validar comportamentos que não estão explicitamente visíveis nos métodos.

No geral, os participantes consideram certos trechos do código como sendo os principais a serem validados. Embora a abordagem sugira primeiro a escrita de contratos para os méto-

Código fonte 21 Pós-condição validando os possíveis retornos do método *GetProcessorArch()*.

```
1 ...
2 private static ProcessorType GetProcessorArch()
3 {
4     Contract.Ensures(Contract.Result<ProcessorType>().Equals(
    ProcessorType.x86) || Contract.Result<ProcessorType>().Equals(
    ProcessorType.x64) || Contract.Result<ProcessorType>().Equals(
    ProcessorType.IPF));
5   ...
6 }
```

dos internos do programa, um participante tende a validar sua suspeita antes disso. O Código fonte 22 especifica o contrato escrito tentando validar o trecho do *if{...}*. Segundo ele, a pré-condição sendo válida, não há necessidade de compreender o método *GetAvailableDB()*, pois o retorno do *Get(...)* ocorreria antes. Apesar de compartilhar do mesmo pensamento do participante, a abordagem objetiva compreender todas as chamadas internas do método ao considerar que contratos devem ser escritos em métodos definidos pelo programa. Sendo assim, para casos como este, onde os conceitos da abordagem não são absorvidos na sua totalidade, a automatização reduziria prováveis compreensões incompletas.

Por último, mais um caso precisa ser descrito para enfatizar a importância da abordagem. O participante ao entrar no método *Select(...)* da classe *DataTable*, percebe que seu retorno é um *array* de *DataRow*. Por se tratar de um *array*, o participante deduz que a contagem de linhas inicia no índice 0. Sendo assim, ele escreve o contrato do Código fonte 18. Os resultados mostram que 91% consideram a usabilidade da abordagem de "muito fácil" a "fácil", enquanto 8% a considera "difícil". O histograma 4.3 quantifica os valores. No fim, é concluído que a abordagem facilita a escrita de contratos não só por descrever quais tipos de contratos devem ser escritos nos métodos, mas por especificar uma metodologia sistemática.

Identificação de variáveis e métodos

O ponto identificação de variáveis e métodos faz parte do processo de escrita de contratos. De acordo com os resultados: nem todas as variáveis e métodos da classe são identificados; e

Código fonte 22 Pós-condição validando um possível fluxo do método *Get(...)*.

```
...
public static DbProviderFactory Get(string invariantName)
{
    Contract.Ensures(invariantName == SQLiteName && GetProcessorArch()
   == ProcessorType.x86);
    if (invariantName == SQLiteName)
    {
        string sqlightAssembly = Path.Combine(AppDomain.CurrentDomain.
   BaseDirectory, "SQLite");
        switch (GetProcessorArch())
        {
        case ProcessorType.x86:
          sqlightAssembly = Path.Combine(sqlightAssembly, "x32");
          break;
        case ProcessorType.x64:
          sqlightAssembly = Path.Combine(sqlightAssembly, "x64");
          break;
        case ProcessorType.IPF:
          sqlightAssembly = Path.Combine(sqlightAssembly, "Itanium");
          break;
      }
      sqlightAssembly = Path.Combine(sqlightAssembly, "system.data.
   sqlite.dll");
      Assembly lib = Assembly.LoadFrom(sqlightAssembly);
      foreach (Type t in lib.GetExportedTypes())
      {
        if(t.FullName == "System.Data.SQLite.SQLiteFactory")
          return (DbProviderFactory)Activator.CreateInstance(t);
      }
      return null;
    }
    ...
}
```

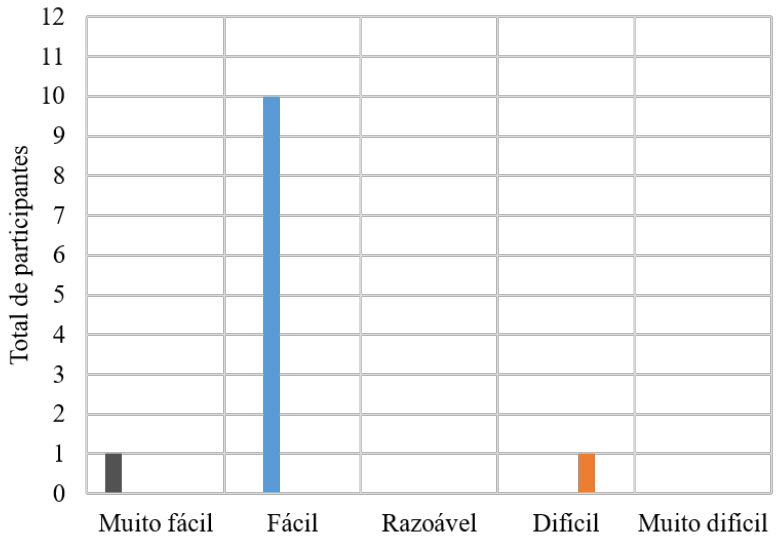


Figura 4.3: Opinião sobre a usabilidade da abordagem.

a transformação de variáveis do escopo do método em globais, podem dificultar a escrita e a compreensão. A seguir são destacados os principais pontos considerados pelas observações.

Mencionado anteriormente os participantes tendem a identificar e definir pré-condição e pós-condição de acordo com a presença de variáveis. Como forma de exemplificar os fatos, é perguntado aos participantes o que os motivam a escrever suposições para determinados trechos de código. Como resposta, o uso da variável *SQLiteName* no *Get(...)* descrito pelo Código fonte 22 é mencionado. Assumindo que esta seja igual à variável *invariantName*, o código apresenta um determinado resultado. No entanto, se escrita a pré-condição verificando a igualdade entre *invarianName* e *SQLiteName*, a validação dela auxiliaria na escrita de outras verificações para aquele trecho. Contudo, não pode ser descartada verificações no método *GetAvailableDB()*. Por último, os valores presentes no *enum*, e a variável global *SQLiteName* não são identificadas por alguns participantes em *GetProcessorArch()* e *GetAvailableDB()*.

Além disso, a tentativa de escrita de pós-condição para a variável *dt* definida no escopo do método *GetAvailableDB()* é feita por um participante. Para este caso, o participante ao perceber o retorno, pensa em escrever uma pós-condição com base no nome dos atributos do objeto. Sem sucesso, informa não saber o que definir pois só tem conhecimento que os atributos são do tipo *string*. Continuando, o mesmo participante identifica no método *Get(...)* a definição da variável *sqlightAssembly*. Ao analisar o trecho do código, resolve transformar

a variável em global, pois considera necessário escrever um contrato verificando qual seria o fluxo do método, o que ressalta a importância de trechos do código para escrita de contratos.

No decorrer do estudo, outros participantes transformam variáveis de método em globais. Quatro participantes ao analisarem o método *GetProcessorArch()*, consideram importante a variável *AddressWidth*, pois ela representa condições para dois tipos de retorno. Diante disto, é preciso avaliar que tipo de contrato utilizar. Para a escrita de pré-condições, é importante considerar a inicialização de acordo com a validação que deseja fazer. O Código fonte 23 descreve duas pré-condições para *GetProcessorArch()* capturadas no estudo. Os contratos das linhas 7 e 8 são escritos para variável não inicializada. Nesse sentido, os contratos são violados devido ao valor ser zero. Assim, para casos similares, a compreensão pode ser afetada. No entanto, as transformações podem representar risco maiores. O risco é atribuído ao impacto que uma transformação pode causar no código, como por exemplo, a mudança de comportamento da variável no método. Todavia, durante o estudo impactos não são detectados. Vale destacar também que quanto maior é o número de transformações e o tempo que leva para serem feitas, maior tende a ser a dificuldade em escrever contratos. Por último, existem mudanças na estrutura do código que antecedem a escrita de contratos.

Código fonte 23 Pré-condições definidas no método *GetProcessorArch()*.

```
1  ...
2  private static int AddressWidth;
3  private static int Architecture;
4  ...
5  private static ProcessorType GetProcessorArch()
6  {
7      Contract.Requires(AddressWidth >= 32 || AddressWidth >= 64);
8      Contract.Requires(AddressWidth == 32 || AddressWidth == 64);
9    ...
10 }
```

Com base nos resultados, as transformações das variáveis em globais, não representam um indicador de acerto para o estudo. Cinco dos 12 participantes realizam este tipo de transformação. Por fim, apenas dois participantes acertam três questões. Entre estes, um transforma variáveis em globais, enquanto o outro não.

Comentários dos contratos escritos e da sintaxe do Code Contracts

A escrita de contrato requer atenção dos participantes. Os resultados apontam que apenas dois dos 12 participantes possuem experiência com *DbC*. Entretanto, apesar do número ser considerado pequeno em relação ao todo, no total quatro participantes têm conhecimento em *DbC*. Perguntado onde os adquiriram, os quatro dizem academicamente. No entanto, desses quatro, dois declaram complementar o conhecimento em sites na internet. Destes, um informa adquirir também em livros, artigos e na prática desenvolvendo e avaliando projetos.

No geral, para os dez participantes que nunca escreveram contratos, o aprendizado inicia com a escrita de contratos simples. Como exemplo, pós-condições verificando se o tipo de retorno é igual ao declarado pelo método, são comuns entre os participantes. Considerada redundante por alguns, a verificação não pode ser descartada. O Código fonte 24 mostra algumas pós-condições escritas com este propósito para os três métodos da classe.

Durante o estudo, quatro participantes escrevem a pós-condição da linha 5 e percebem que o tipo retornado não é igual ao declarado pelo método. De acordo com as observações de alguns participantes, o tipo *DbProviderFactories* retornado no bloco *else{...}* pode ter relação com a violação do contrato. Em especial, um participante menciona que o comportamento pode estar relacionado a herança de tipos da classe *DbProviderFactory*, porém a suposição não é validada. Em contrapartida a afirmação tem sentido. Um outro participante ao escrever a pós-condição da linha 6 valida uma das suposições. De acordo com o resultado, é possível afirmar que o *cast* não é feito para algumas condições. Contudo, o contrato negando se o tipo de retorno é igual a *DbProviderFactory* não foi escrito.

As observações indicam a escrita de contratos específicos à medida que o código tende a ser compreendido. Nesse sentido, um participante ao observar a instrução de adição de um novo objeto a uma linha, define a pós-condição da linha 4 do Código fonte 25. Assumindo que o contrato escrito valida se a quantidade de linhas é igual ao seu antigo valor mais um, o participante comenta não compreender o motivo da violação do contrato. Segundo ele, a violação representa um resultado errado. Entretanto, um outro participante ao escrever a pós-condição da linha 5, valida a suposição. Assim, diante dos fatos, é possível concluir que a inexperiência com a escrita de contratos pode afetar a compreensão, uma vez que, a lógica do participante está correta e o contrato está escrito errado.

Por um momento, suspeitamos que o tempo de experiência com desenvolvimento pode

Código fonte 24 Pós-condições verificando o retorno dos métodos.

```
...
public static DbProviderFactory Get(string invariantName)
{
    ...
    Contract.Ensures(Contract.Result<DbProviderFactory>().GetType() ==
    typeof(DbProviderFactory));
    Contract.Ensures(!Contract.Result<DbProviderFactory>().GetType().
    Equals(typeof(Activator)));
    ...
    else
    {
      DataTable dt = GetAvailableDB();
      DataRow[] rows = dt.Select(string.Format("InvariantName = '{0}'",
     invariantName));
      if (rows.Length == 0)
        return null;
      return DbProviderFactories.GetFactory(rows[0]);
    }
}

public static DataTable GetAvailableDB()
{
    Contract.Ensures(Contract.Result<DataTable>().GetType() == typeof(
    DataTable));
    ...
}

private static ProcessorType GetProcessorArch()
{
    Contract.Ensures(Contract.Result<ProcessorType>().GetType() ==
    typeof(ProcessorType));
    Contract.Ensures(Contract.Result<ProcessorType>().GetType().IsEnum)
    ;
    ...
}
...
```

Código fonte 25 Pós-condição verificando a quantidade de linhas do *GetAvailableDB()*.

```
1 ...
2 public static DataTable GetAvailableDB()
3 {
4     Contract.Ensures(dt.Rows.Count == Contract.OldValue(dt.Rows.Count)
    + 1);
5     Contract.Ensures(Contract.Result<DataTable>().Rows.Count ==
    Contract.OldValue(DbProviderFactories.GetFactoryClasses().Rows.Count
     + 1));
6   ...
7 }
```

influenciar na escrita de contratos. Com base nas doze amostras, observando o diagrama de caixa da Figura 4.4 é possível identificar que o máximo de questões corretas foram de 3 para os participantes que possuem de 2 a 3 anos e de 3 a 4 anos. Em relação ao número mínimo de questões corretas os participantes que possuem de 2 a 3 anos acertaram pelo menos 2, enquanto que os com experiência de 3 a 4 anos acertaram 1. Contudo, os resultados apontam que devido a pequena quantidade de participantes não é possível inferir correlação entre a experiência dos desenvolvedores em anos e o acerto de questões.

Embora existam outros fatores que possam estar relacionados à dificuldade na escrita de contratos, uma pergunta pós estudo qualifica o nível de compreensão do código sem a escrita de contratos. O histograma 4.5 mostra as respostas dos participantes em relação a pergunta. As respostas variam de "fácil" a "muito difícil". O motivo da qualificação está relacionado aos detalhes específicos que compõem a implementação de banco de dados.

Durante o estudo alguns participantes mencionam não compreender o código apenas lendo, entretanto, outros declaram compreender parte deste lendo, porém só tem a certeza do que o método faz ao escrever contratos. É comum entre participantes que desconhecem os princípios do *DbC* escreverem contratos que realizam verificações mais simples, a exemplo de não nula. Nesse sentido, um dos 12 participantes classifica a escrita dos seus contratos como simples. Outros casos ocorrem envolvendo a escrita de contratos com o operador lógico "*&&*". Dessa vez, surgem incompreensões para contratos violados que são verificados pela primeira vez. Desse modo, ao utilizarem o operador "*&&*" em um único contrato, os

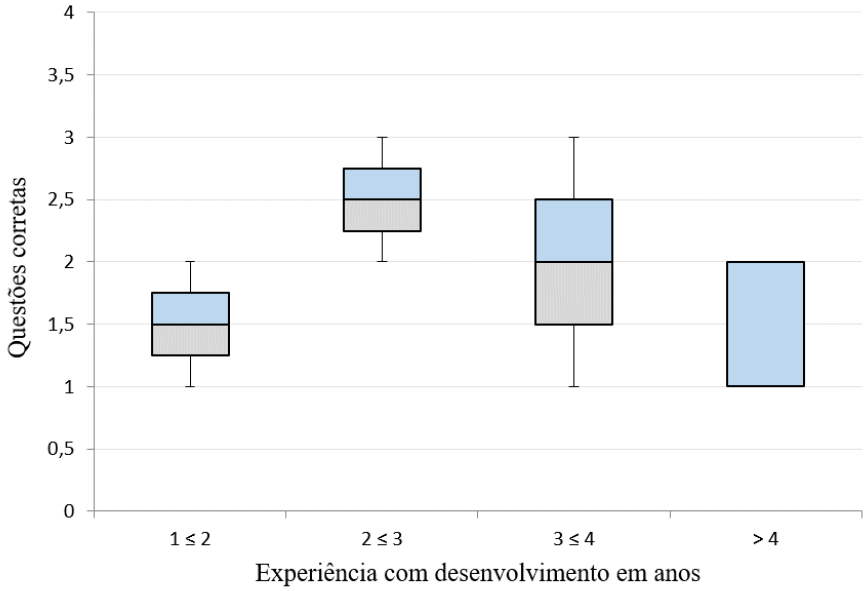


Figura 4.4: Diagrama de caixa considerando a experiência com desenvolvimento com o acerto das questões.

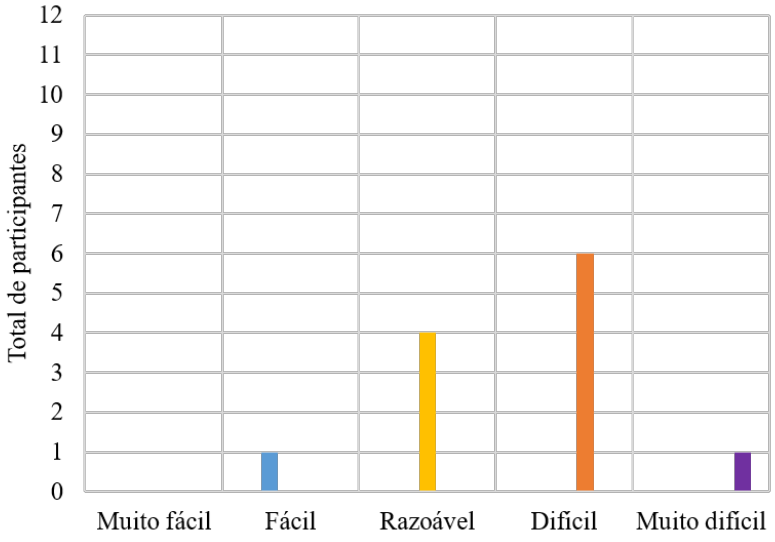


Figura 4.5: Nível de dificuldade para compreender o código sem contratos.

participantes, a princípio, são incapazes de identificar qual lado está sendo violado. Diante disto, para uma melhor compreensão é importante escrever contratos isoladamente. Por último, a violação de contratos sugere aos participantes a escrita de novas suposições. No estudo são observados estes tipos de casos. Destacando um deles, um participante ao constatar no *GetAvailableDB()* a violação da pós-condição que verifica se a quantidade de linhas

é igual ao antigo valor mais um, define uma nova verificando se a quantidade de linhas é diferente do antigo valor.

Fluxo do participante no código

Para responder a QP2, é verificado como os participantes avaliam a escrita de contratos para os métodos de acordo com as chamadas destes na classe. Nesse sentido, um participante avalia a visibilidade do método. Segundo ele, métodos privados interessam apenas a classe, enquanto métodos públicos são relevantes para o cliente. A afirmação do participante tem sentido devido a possibilidade de métodos públicos externarem comportamentos para métodos de outros pacotes. Entretanto, o estudo considera a escrita de contratos para qualquer método que seja definido pelo programa independente da visibilidade. Assim, o comportamento é mencionado como forma de mostrar características diversificadas adotadas pelos participantes.

Com base em outras observações, as escolhas podem ocorrer devido a dificuldades ao escrever contratos. Em dois casos, os participantes decidem começar pelo método *GetProcessorArch()*. Ao ler o método, eles não identificam possíveis contratos que podem ser escritos. Neste caso, a dificuldade está associada à inexperiência com *DbC*. Diante disto, os participantes mudam de método e começam pelo *Get(...)*, devido à presença de parâmetro. Segundo eles, o fato da abordagem relacionar pré-condição com parâmetro facilita a escrita dos contratos. Dessa forma, o primeiro contrato definido ocorre no método *Get(...)*. Em seguida, compreendida a proposta do estudo e criados os primeiros contratos, eles voltam para o *GetProcessorArch()*.

A análise de chamadas a métodos incentiva a escrita de contratos. Durante o estudo, os participantes ao considerarem blocos condicionais relevantes, definem pré-condição e pós-condição com base em suposições. As linhas 4-5 do Código fonte 26 apresentam duas pré-condições definidas com este propósito. Apesar de ambas serem violadas, de modo geral, os participantes classificam a validação como parte da compreensão do fluxo e não do código como um todo.

Código fonte 26 Pré-condições para o *if(...)* do *Get(...)*.

```
1 ...
2 public static DbProviderFactory Get(string invariantName)
3 {
4     Contract.Requires(invariantName != SQLiteName);
5     Contract.Requires(invariantName == SQLiteName);
6     ...
7 }
```

4.3 Ameaças à validade

Verificar se a compreensão e escrita de contratos ocorre para variáveis **ameaça à validade de construto**. Devido ao fato do estudo avaliar a compreensão envolvendo variáveis em métodos que possuem blocos condicionais, os participantes consideram estes trechos como potenciais indicadores comportamentais. Assim, definem pré-condição e pós-condição para estes trechos buscando validá-las. Outro ponto destaca a tentativa de compreensão de fragmentos do código com base nos nomes de variáveis e métodos. No entanto, para minimizar esta ameaça, o questionário pós-estudo apresenta questões avaliando o comportamento e o fluxo de métodos.

O fato das ações e pensamentos serem gravados pode afetar o comportamento de alguns participantes. Isto representa uma **ameaça à validade interna**. Outra particularidade do estudo é o fato de terem que escrever contratos para compreender o código. Nesse sentido, a experiência prévia e o ambiente podem afetar os resultados. Por fim, existe a possibilidade de os participantes inexperientes sentirem-se pressionados a escreverem contratos. Para evitar estas ameaças, nas considerações pré-estudo são informados que a avaliação não leva em consideração como os contratos são definidos, e que podem utilizar a internet para qualquer tipo de consulta a qualquer momento. Além disso, o fato da compreensão depender da escrita de contratos, influencia como os participantes compreendem o código.

Algumas **ameaças à validade externa** estão presentes no estudo. Dentre elas, são destacadas o fato: dos participantes serem inexperientes com a biblioteca *Code Contracts* e com a linguagem *C#*; da apresentação que antecede o estudo conter exemplos de contratos que servem de exemplos para o estudo; e da abordagem ser utilizada apenas com uma biblio-

teca e uma linguagem de programação. Nesse contexto, o estudo pode deixar de capturar particularidades relevantes. Entretanto, para aliviar estas ameaças, a população do estudo possui experiência com: lógica e outras linguagens de programação; e desenvolvimento de projetos em empresas de grande porte utilizando diversas tecnologias. Referente à apresentação que antecede o estudo, do ponto de vista experimental, é inviável o estudo não conter uma apresentação mostrando como escrever contratos. No entanto, pretende-se replicar este estudo com mais participantes experientes em contratos, a fim de aumentar a confirmação da generalização dos resultados.

4.4 Questões da Pesquisa

A seguir são respondidas as questões da pesquisa com base nos resultados e discussões da Seção 4.2.

QP1 *A escrita de contratos, de acordo com a abordagem proposta auxilia os desenvolvedores a compreenderem o método?*

Há indícios. Avaliado os três métodos da classe *DatabaseFactory*, os resultados indicam que dos participantes que acertam apenas uma questão, pelo menos um método ou parte dele é compreendido. O foco do estudo é de natureza qualitativa, nesse sentido, as observações evidenciam que a escrita de contratos para métodos mais internos do programa, tendem a auxiliar na compreensão de comportamentos do método. Isto é visível quando as chamadas internas são utilizadas para definir fluxos alternativos no código. Assim, considerando a existência de blocos condicionais, a compreensão tende a ser direcionada. Desse modo, predições são feitas antes mesmo do código ser compreendido na sua totalidade. Além disso, a consulta constante da abordagem, como forma de identificar quais passos devem ser seguidos, fortifica a necessidade do participante quanto a esta para auxiliá-lo na compreensão dos métodos. Todavia, acredita-se que um número maior de amostras considerando outros códigos possa mostrar uma variação nos resultados.

QP2 *A necessidade de compreender um programa favorece a escrita de contratos?*

Sim. Com base nas observações, a abordagem facilita a escrita de contratos à medida que provê quais características da classe e dos métodos devem ser avaliadas para determinar qual

tipo de contrato escrever. Acredita-se que os resultados não sofrem alterações para novos métodos. No entanto, se levado em consideração outras bibliotecas que permitam a utilização dos princípios de *DbC*, juntamente com outras linguagens e considerando *API*s externas, a compreensão pode ser ainda maior, haja vista que dificuldades podem ser minimizadas e métodos implementados por outras pessoas tendem a ser compreendidos.

4.5 Conclusões

O capítulo apresenta a avaliação da usabilidade da abordagem e a escrita de contratos para o programa. Neste sentido, o estudo adota métodos qualitativos que indicam o funcionamento do seu uso. No geral, os participantes reconhecem o auxílio desta, na definição de contratos, e não apresentam dificuldades em utilizá-la.

O estudo é importante uma vez que avalia a abordagem por meio da utilização por desenvolvedores atuantes no mercado de trabalho. Desse modo, é possível compreender como a mesma auxilia na compreensão e facilita a escrita de contratos em um programa real. Por fim, outro ponto relevante a ser destacado, é o fato do incentivo para a escrita de contratos.

Capítulo 5

Conclusão

Neste trabalho, foi apresentada uma abordagem sistemática que tem como objetivo auxiliar a compreensão de programas por meio da escrita de contratos. Um contrato é uma especificação formal feita nos métodos que verifica seus componentes e comportamentos e que garante a conformidade entre a implementação e a regra especificada.

Contratos são escritos em linguagens de programação que dão suporte a este tipo de verificação. A ideia foi proposta por Bertrand Meyer [37] na linguagem de programação Eiffel e considera a utilização destes durante atividades de *design* e implementação do programa. Embora desenvolvedores apresentem certa resistência para documentar programas ou escrever contratos [27; 46], este trabalho avalia, através do uso da abordagem proposta, a relação entre a compreensão e a escrita de contratos. Nesse sentido, com base no estado da arte envolvendo as áreas desta dissertação, alguns trabalhos propuseram abordagens diversificadas para compreender programas incluindo a utilização de *Design by Contract* [52; 1]. No geral, dentre todas as abordagens que objetivam compreender programas, continua sendo a mais utilizada a inspeção visual e manual dos métodos. Diante das conclusões dos trabalhos que se relacionam com o estudo, a não utilização das abordagens ocorre devido o desconhecimento da existência destas, o tempo que muitas vezes é curto e a falta de prática que também envolve a complexidade em aplicá-las. Além disso, a grande parte das abordagens envolve o aprendizado e a utilização de práticas que os desenvolvedores não costumam utilizar.

A avaliação da abordagem, que detalha passos sistematicamente para escrita de contratos, ocorreu por meio de um estudo em ambientes de desenvolvimento de *software* com 12

desenvolvedores. A abordagem considera avaliar o retorno, os parâmetros, e as variáveis de método e globais, provendo ao desenvolvedor suporte em relação a quais tipos de contratos escrever nos métodos. Esta também pode ser utilizada para qualquer linguagem de programação que dê suporte a *DbC*. Além disto, a validação dos contratos é efetuada pela execução dos testes de unidade que foram implementados pela equipe que desenvolveu o programa.

Os resultados, considerando a usabilidade da abordagem, apontam que onze dos 12 desenvolvedores classificam de muito fácil a fácil o seu uso. No entanto, apenas um classifica a abordagem como sendo difícil devido desconhecer os princípios de *DbC* e nunca ter utilizado a linguagem de programação *C#*.

Com base nas observações resultantes do protocolo *Think Aloud*, é possível inferir que partes do código que exigem a escrita de apenas um contrato são mais fáceis de compreender, devido o teste validar apenas uma única suposição. Além disso, os desenvolvedores leem o código para adquirir um conhecimento rápido e geral, pois necessitam deste para escrever contratos. Assim, a medida que o conhecimento aumenta em relação as rotinas dos métodos, contratos mais específicos, por exemplo, envolvendo regras de negócios tendem a ser escritos. Por fim, contratos representam uma alternativa a compreensão do programa, pois desenvolvedores utilizam-os para validar suposições que dificilmente seriam compreendidas apenas lendo o código.

Os questionamentos dos desenvolvedores geram suposições que posteriormente são transformadas em contratos. Essas suposições são feitas com mais facilidade pelos desenvolvedores no momento em que estes identificam estruturas presentes nos métodos, cujas quais possuem um conhecimento prévio. Assim, para programas que utilizam métodos conhecidos, a escrita de contratos torna-se descomplicada. Logo, a escrita de contratos em programas desconhecidos consome tempo dos desenvolvedores, no entanto, este é recuperado em fases posteriores do desenvolvimento, a exemplo da correção de *bugs*. Com o objetivo de manter os programas funcionando corretamente, desenvolvedores corrigem bugs. Nesse sentido, caso os contratos escritos durante a fase de compreensão sejam mantidos no programa, estes minimizariam os esforços dos desenvolvedores uma vez que o código estaria em conformidade com as especificações. Além disso, os contratos forçam os desenvolvedores, mesmo os inexperientes em *DbC*, a pensarem em regras que compõe o programa, fazendo-os compreendê-los. Em conclusão, apesar de pós-condição prover a compreensão

do comportamento dos métodos, esta possui a mesma importância de pré-condição, pois a abordagem sugere que ambas estejam presentes nos métodos.

5.1 Trabalhos Relacionados

Esta dissertação relaciona-se com trabalhos que buscam propor abordagens para auxiliar a compreensão de programas, bem como os que utilizam *Design by Contract* com o objetivo de examinar contratos escritos por desenvolvedores. A seguir, estes são relacionados com a abordagem deste trabalho.

5.1.1 Compreensão de Código

Nas últimas décadas, pesquisadores têm dedicado seu tempo estudando e propondo abordagens para compreensão de programas por meio de ferramentas e de análises manuais [44; 62; 48]. Perez et al. [44] propuseram uma abordagem que associa características para cada componente que compõe o programa e exibe essas associações por meio de um gráfico circular *sunburst*. Intitulada *Spectre Feature Comprehension* (*SFC*), a abordagem utiliza técnicas semelhantes as de localização de falhas em programas denominada *Spectrum-Based Fault Localization* (*SFL*). O objetivo é detectar falhas do programa de modo automático por meio de um algoritmo, que considera o código-fonte e os testes de unidade, gerando relatórios. Nesse sentido, o algoritmo busca associações entre os componentes (classes, métodos ou instruções) do programa por meio de uma análise dinâmica envolvendo os testes de unidade, para em seguida disponibilizar estas associações por meio de um gráfico circular em camadas. O gráfico descreve a hierarquia do programa associando componentes aos quais deseja avaliar, sendo a camada interna a raiz do programa, enquanto a mais externa pode até representar linhas de código sem relação direta. Cores são utilizadas para representar essa associação. Deste modo, a abordagem provê aos desenvolvedores auxílio a compreensão de programas, uma vez que considera os testes de unidade como um indicador neste processo. Contudo, a nossa abordagem diverge da deles a medida que busca compreender trechos do programa por meio da escrita de contratos.

Considerada, por pesquisadores, uma atividade de grande importância, a compreensão deve ocorrer antes de qualquer tarefa de manutenção do programa [62; 42]. Vassallo et

al. [62] desenvolveram uma ferramenta, denominada *mining sourCe cOde Descriptions from developErs diScussions* (*CODES*), que tem por objetivo documentar métodos de classes Java com base em discussões realizadas em um fórum na *internet*.

A ferramenta *CODES* extrai métodos a partir do código fonte do programa e busca informações a respeito destes no fórum de discussão *StackOverflow*[1]. Esta permite o desenvolvedor selecionar as classes que possuam pelo menos um método, e que gostaria de documentar. A ferramenta ainda busca informações dos métodos por meio de um banco de dados (*offline*). No entanto, para que o banco de dados contenha informações, é preciso que pesquisas prévias (*online*) tenham sido feitas. Apesar do trabalho ter o mesmo objetivo final que o nosso e possuir também na etapa de compreensão, pesquisas de *internet*, este possui diferenças entre as abordagens. Dentre essas, a ferramenta *CODES* utiliza unicamente a *internet*, mesmo que para alimentar o banco de dados, como meio de compreensão do programa, sendo que esta pode encontrar-se indisponível ao desenvolvedor em algum momento. Uma outra diferença é que pode ocorrer de não existirem comentários no fórum *StackOverflow* acerca dos métodos que pretende compreender. Por fim, como mostrado nos resultados desta dissertação, desenvolvedores tendem a considerar outras fontes de compreensão de programas além da documentação.

Apesar dos esforços em pesquisas envolvendo a compreensão de programas, pouco se sabe como os desenvolvedores a praticam. Pensando nisto, Roehm et al. [48] fizeram um estudo envolvendo desenvolvedores profissionais com o objetivo de entender como eles compreendem programas, e quais métodos e ferramentas eles utilizam para isto. O estudo consistiu na adoção das estratégias, nas informações necessárias e nas ferramentas utilizadas. Como resultados, concluíram que estratégias distintas são utilizadas e válidas em diferentes contextos. Além disso, perceberam que os desenvolvedores se colocam no papel de usuários finais como alternativa a leitura e depuração do código para compreenderem o comportamento do programa. Por fim, descobriram que as ferramentas do estado da arte, nas empresas observadas, não são utilizadas na prática pela indústria. O trabalho apresenta conclusões gerais interessantes, a exemplo da experiência ser um facilitador para familiarização do programa desconhecido. Nesse sentido, em conformidade com a conclusão do experimento, o estudo feito nessa dissertação considera que esta não apresenta relação direta com

[1] http://stackoverflow.com/

a compreensão, pois percebemos, em alguns casos, que desenvolvedores mais experientes compreenderam menos o programa do que desenvolvedores menos experientes. Contudo, isto não pode ser generalizado, pois acreditamos que a compreensão sofre influência de conhecimentos além dos envolvidos no contexto do programa. Por último, consideramos o trabalho proposto pelos autores relevante por observar as estratégias e ferramentas utilizadas no âmbito industrial, entretanto, nosso trabalho se difere por propor uma abordagem sistematizada, utilizando *Design by Contract*, que é desconhecida nesse meio.

5.1.2 Design by Contract

Vários estudos foram realizados envolvendo a ferramenta *Daikon* [25; 29; 5; 14; 13]. Entretanto, estes propõem abordagens automatizadas, por meio de ferramentas, que não capturam o comportamento dos desenvolvedores no momento que compreendem programas. Além disso, estas inferem em metodologias, às vezes, não utilizadas, no ambiente de trabalho destes. Ernst et al. [10; 11] propuseram *Daikon* para descobrir invariantes dinamicamente com base nos resultados obtidos pela execução do programa. Para isto, é preciso passar como entrada um programa escrito em *C*, *C++*, *Java* ou *Perl*, sendo possível estender a outras linguagens de programação. Os valores verdadeiros, gerados durante a execução do programa, são reportados formando um conjunto de invariantes com base no rastreamento. Diferente de *Daikon*, Lo et al. [29] propuseram uma abordagem com base em mineração de propriedades temporais, capturando ordens de lógicas sob a forma de autômatos. A abordagem considera o autômato do programa gerando um conjunto de regras de lógica temporal (*LTL*) de eventos múltiplos. Nesse sentido, o autômato expressa a visão geral especificada do programa, enquanto a mineração das regras de *LTL* expressam as propriedades do programa com base nos rastreamentos observados. Ambos os trabalhos, diferem do nosso por não prover ao desenvolvedor liberdade quanto a escrita de contratos. Em particular, a abordagem proposta por Lo et al. [29] motiva a nossa por implicar que uma pré-condição está para uma pós-condição. No entanto, o estudo realizado por esta dissertação observou que tal fato não ocorre sempre, devido existirem casos em que pré-condições inexistem para variáveis não inicializadas. Além disso, observamos que é mais comum pré-condições estarem para pós-condições em trechos do programa que apresentam blocos condicionais *if-else*.

Estudos recentes visam compreender como desenvolvedores escrevem contratos [45; 46;

49]. Polikarpova et al. [45] observaram que desenvolvedores possuem dificuldades e apresentam resistência em escrever contratos. Nesse sentido, propuseram realizar um estudo comparando os contratos escritos por desenvolvedores utilizando a linguagem *Eiffel* e os produzidos pela ferramenta *Daikon*. Para *Daikon* inferir invariantes, foi preciso escrever suítes de teste pois os programas utilizados não os continham. Tal ato pode ser um problema, pois a criação dos testes de unidade ocorreram por desenvolvedores que não fizeram parte do processo de desenvolvimento. Assim, podem ocorrer casos em que os testes escritos não refletem os reais requisitos do programa, como também não validam regras de negócio que a equipe de desenvolvedores oficial havia implementado. Nesse contexto, o estudo desta dissertação considera como premissa os testes de unidade escritos pela equipe de desenvolvimento oficial do programa, refletindo o real significado destes com o comportamento esperado do programa.

Um outro trabalho caracteriza os contratos que os desenvolvedores escrevem e poderiam escrever. Schiller et al. [49] apresentaram um estudo em diversos programas utilizando *Code Contracts*. Dentre estes, quatro foram avaliados profundamente com o objetivo de compreender a escrita de contratos. O estudo também apresenta duas ferramentas, uma chamada *Celeriac* que produz compatibilidade com Daikon para traçados de binários de execução com o *.NET*, e outra chamada *Code Inserter* (atual *Scout*) que é uma extensão para o *Visual Studio* cujo propósito é descobrir e inserir contratos utilizando o *Code Contracts* em códigos *C#*. Para avaliação, os contratos foram classificados em três categorias gerais - *Common-Case*, *Repetitive with Code* e *Application-Specific* - e posteriormente agrupados e quantificados de acordo com estas. Com base na classificação proposta, este estudo poderia ter alocado os contratos escritos de acordo essas. No entanto, tal ato é irrelevante, para este trabalho, uma vez que se buscou observar os contratos escritos pelos desenvolvedores sem o auxílio de ferramentas que os inferissem. Além disso, se considerássemos a categoria *Repetitive with Code* que representa a propriedade de contrato *Return Value*, essa caracterizaria 100% dos contratos escritos, ao contrário do trabalho de Schiller et al. que caracterizou apenas 8% para o programa *Labs Framework* e 30% para o programa *Quick Graph*. Acredita-se que isto ocorra devido a necessidade dos desenvolvedores em escrever contratos para compreender o programa.

Por fim, contratos têm sido utilizados em contextos distintos. Sohr et al. [52] propuse-

ram uma abordagem para assegurar o uso correto de bibliotecas de segurança de programas baseada nos princípios de *Design by Contract*. Eles observaram que os desenvolvedores ao fazerem o uso de bibliotecas envolvendo criptografia, controle de acesso e autenticação, podem introduzir vulnerabilidades no programa. Neste sentido, é preciso utilizar estas corretamente a fim de reduzir riscos de outras pessoas terem acesso aos dados. Por último, o trabalho presente nesta dissertação possui relação ao fazer uso dos princípios de *Design by Contract* para compreensão de programas. No entanto, diferentemente do trabalho de Sohr et al. este não utiliza ferramentas para inferir contratos.

5.2 Trabalhos Futuros

Em trabalhos futuros, pretendemos aprimorar os passos da abordagem para auxiliar desenvolvedores a compreender programas por meio de um estudo envolvendo outros projetos na linguagem *C#*. O objetivo é adquirir conhecimento de outros possíveis indicadores, a exemplo da utilização de padrões de projeto, que possam ser avaliados por meio de contratos, auxiliando os desenvolvedores a adquirirem conhecimentos mais geral.

Além disso, pretendemos aplicar a abordagem sistemática a outras linguagens de programação que dê suporte a *DbC*, buscando validá-la quanto ao seu uso em outros ambientes de desenvolvimento. Neste quesito, visamos incluir linguagens que suportem implicações diretamente, a exemplo de *JML* [23], pois a maioria das linguagens *.NET*, como *C#*, não possuem um operador que represente implicações diretamente. Assim, a inclusão deste operador pode fazer com que desenvolvedores escrevam mais contratos usando implicações.

Ainda no sentido de evoluir a abordagem, contratos envolvendo invariantes podem ser utilizados por esta, pois são capazes de auxiliar na validação das suposições dos desenvolvedores em relação às variáveis globais. O estudo realizado por esta dissertação não considerou a aplicabilidade destes, devido a considerar complexo para a maioria dos desenvolvedores que não possuem conhecimento em *DbC*. Assim, um estudo envolvendo invariantes deve ser feito. Além disso, o estudo deve ser mais longo para verificar se os contratos serão realmente úteis no desenvolvimento.

Consideramos importante automatizar a abordagem sistemática devido à possível redução na carga de trabalho dos desenvolvedores, uma vez que não há mais necessidade de

leitura dos passos descritos por esta. Sobretudo, porque a escrita de contratos pode ser difícil para alguns. Desse modo, o objetivo passa a ser em dedicado a agilizar a escrita de contratos que é considerada tediosa pelos desenvolvedores. Além disso, a automatização pode incluir sugestões de contratos para as suposições dos desenvolvedores. Com base no trabalho de Schiller et al. [49], envolvendo 90 programas *open-souce* escritos em *C#*, contratos que caracterizam implicações representam apenas 1%. Nesse sentido, acreditamos que esse número possa ser maior caso a abordagem proposta por este trabalho passe também a sugerir e dar suporte a escrita destes tipos de contratos.

Por fim, algumas questões de pesquisa futura podem ser levantadas. São elas:

- *Até que ponto a transformação de variável do método em global pode impactar no comportamento do código?* O objetivo é avaliar se as transformações das variáveis podem gerar mudança comportamental dos métodos; e
- *A escrita de contratos para métodos de APIs externas induzem a uma melhor compreensão do programa?* Aqui visamos compreender se bibliotecas externas que são utilizadas no programa podem favorecer a compreensão deste de maneira mais ampla. Sobretudo, podemos avaliar se as bibliotecas estão sendo utilizadas corretamente nos programas, impactando em uma melhor compreensão.

Bibliografia

[1] Sergio Areias, Daniela Carneiro da Cruz, and Jorge Sousa Pinto. Contract-based slicing helps on safety reuse. In *Proceedings of the 18th International Conference on Program Comprehension*, 2010.

[2] Mike Barnett, Rustan Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[4] Walter Bischofberger. Sniff (abstract): A pragmatic approach to a c++ programming environment. *SIGPLAN OOPS Messenger*, 1993.

[5] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2006.

[6] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 1983.

[7] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Ldiff: An enhanced line differencing tool. In *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[8] Massimiliano Di Penta, Richard Erick Kurt Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *Proceedings of the 15th International Conference on Program Comprehension*, 2007.

[9] Françoise Détienne. *Software design – cognitive aspects*. Springer, 2002.

[10] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[12] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 25th ACM Symposium on Applied Computing*, 2010.

[13] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[14] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[15] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[16] Shihong Huang and Scott Tilley. Towards a documentation maturity model. In *Proceedings of the 21st Annual International Conference on Documentation*, 2003.

[17] IEEE. *IEEE standard for software maintenance, IEEE Std 1219-1999*. IEEE Press, 1999.

[18] Frederick Phillips Brooks Junior. *The mythical man-month: Essays on software engineering*. Addison-Wesley Professional (Anniversary edition), 1995.

[19] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, 2001.

[20] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[21] Douglas Kramer. Api documentation from source code comments: A case study of javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, 1999.

[22] Reto Kramer. iContract - the java(tm) design by contract(tm) tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, 1998.

[23] Gary Leavens, Albert Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*. Springer, 1999.

[24] Gary Leavens, Albert Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Software Engineering Notes*, 2006.

[25] Caroline Lemieux. Mining temporal properties of data invariants. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[26] Timothy Lethbridge and Nicolas Anquetil. Architecture of a source code exploration tool: A software engineering case study. *Computer Science Technical Report of University of Ottawa*, 1997.

[27] Timothy Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE Computer Society Press*, 2003.

[28] Stanley Letovsky. Cognitive processes in program comprehension. In *Proceedings of the 1st Workshop on Empirical Studies of Programmers*, 1986.

[29] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 2008.

[30] Francesco Logozzo. Practical verification for the working programmer with codecontracts and abstract interpretation. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2011.

[31] Alex Loh and Miryung Kim. Lsdiff: A program differencing tool to identify systematic structural differences. In *Proceedings of the 32nd International Conference on Software Engineering*, 2010.

[32] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[33] Carma McClure. *The three Rs of software automation: Re-engineering, repository, reusability*. Prentice Hall, 1992.

[34] Nenad Medvidovic, Paul Grünbacher, Alexander Egyed, and Barry W. Boehm. Bridging models across the software lifecycle. *Journal of Systems and Software*, 2003.

[35] Bertrand Meyer. Genericity versus inheritance. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, 1986.

[36] Bertrand Meyer. Applying "design by contract". *Computer*, 1992.

[37] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[38] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[39] Richard Mitchell and Jim McKim. *Design by contract, by example*. Addison-Wesley, 2002.

[40] Hausi Müller and Karl Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, 1988.

[41] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, 1993.

[42] Sebastiano Panichella, Jairo Aponte, Massimiliano Di Penta, Andrian Marcus, and Gerardo Canfora. Mining source code descriptions from developer communications. In *Proceedings of the 20th International Conference on Program Comprehension*, 2012.

[43] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 1987.

[44] Alexandre Perez and Rui Abreu. A diagnosis-based approach to software comprehension. In *Proceedings of the 22nd International Conference on Program Comprehension*, 2014.

[45] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009.

[46] Nadia Polikarpova, Carlo Furia, Yu Pei, Yi Wei, and Bertrand Meyer. What good are strong specifications? In *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[47] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, 2002.

[48] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[49] Todd Schiller, Kellen Donohue, Forrest Coward, and Michael Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[50] John Sharp. *Microsoft Visual C# Step by Step*. Microsoft Press, 8th edition, 2015.

[51] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 1979.

[52] Karsten Sohr, Tanveer Mustafa, Philipp Hirch, and Markus Gulmann. Towards security program comprehension with design by contract and slicing. Technical Report TZI-Bericht-2016-80, Universität Bremen, Germany, 2016.

[53] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 1984.

[54] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and Vijay Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th International Conference on Automated Software Engineering*, 2010.

[55] Giriprasad Sridhara, Lori Pollock, and Vijay Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[56] Margaret Storey and Hausi Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the International Conference on Software Maintenance*, 1995.

[57] Margaret Storey, Kenny Wong, and Hausi Müller. How do program understanding tools affect how programmers understand programs. In *Proceedings of the 4th Working Conference on Reverse Engineering*, 1997.

[58] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: Past, present and future. *Software Quality Journal*, 2006.

[59] TakeFive Software GmbH. Sniff+ user's guide and reference. http://www.takefive.com, 1996.

[60] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, 2012.

[61] Scoot Tilley, Santanu Paul, and Dennis Smith. Towards a framework for program understanding. In *Fourth Workshp on Program Comprehension*, 1996.

[62] Carmine Vassallo, Sebastiano Panichella, Massimiliano Di Penta, and Gerardo Canfora. Codes: Mining source code descriptions from developers discussions. In *Proceedings of the 22nd International Conference on Program Comprehension*, 2014.

[63] Anneliese Von Mayrhauser and Marie Vans. From program comprehension to tool requirements for an industrial environment. In *Proceedings of the 2nd Workshop on Program Comprehension*, 1993.

[64] Anneliese Von Mayrhauser, Marie Vans, and Adele Howe. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance*, 1997.

[65] Mario Linares Vásquez, Luis Fernando Cortes-Coy, Jairo Aponte, and Denys Poshyvanyk. ChangeScribe: A tool for automatically generating commit messages. In *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[66] Susan Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 1986.

[67] Mickey Williams. *Microsoft visual c# .net*. Microsoft Press, 2002.

Apêndice A

Implementação da Classe *DatabaseFactory*

Neste apêndice é apresentado a implementação da classe *DatabaseFactory* do projeto *Archiver*. Nele está contido toda a implementação da classe, incluindo as variáveis globais e os três métodos utilizados pelo estudo.

```
...
namespace FreeSCADA.Archiver
{
  public static class DatabaseFactory
  {
    public const string SQLiteName = "System.Data.SQLite";

    enum ProcessorType
    {
      x86,
      x64,
      IPF
    }

    public static DbProviderFactory Get(string invariantName)
    {
      if (invariantName == SQLiteName)
      {
```

```
19        string sqlightAssembly = Path.Combine(AppDomain.CurrentDomain.
    BaseDirectory, "SQLite");
20        switch (GetProcessorArch())
21        {
22          case ProcessorType.x86:
23            sqlightAssembly = Path.Combine(sqlightAssembly, "x32");
24            break;
25          case ProcessorType.x64:
26            sqlightAssembly = Path.Combine(sqlightAssembly, "x64");
27            break;
28          case ProcessorType.IPF:
29            sqlightAssembly = Path.Combine(sqlightAssembly, "Itanium");
30            break;
31        }
32        sqlightAssembly = Path.Combine(sqlightAssembly, "system.data.
    sqlite.dll");
33
34        Assembly lib = Assembly.LoadFrom(sqlightAssembly);
35        foreach (Type t in lib.GetExportedTypes())
36        {
37          if(t.FullName == "System.Data.SQLite.SQLiteFactory")
38            return (DbProviderFactory)Activator.CreateInstance(t);
39        }
40
41        return null;
42      }
43      else
44      {
45        DataTable dt = GetAvailableDB();
46        DataRow[] rows = dt.Select(string.Format("InvariantName = '{0}'",
     invariantName));
47
48        if (rows.Length == 0)
49          return null;
50
51        return DbProviderFactories.GetFactory(rows[0]);
52      }
```

```
53      }
54
55      public static DataTable GetAvailableDB()
56      {
57          DataTable dt = DbProviderFactories.GetFactoryClasses();
58          dt.Rows.Add(new object[] { "SQLite Data Provider", ".Net Framework
    Data Provider for SQLite", SQLiteName, "System.Data.SQLite.
    SQLiteFactory, System.Data.SQLite" });
59          return dt;
60      }
61
62      private static ProcessorType GetProcessorArch()
63      {
64          using (System.Management.ManagementClass processors = new System.
    Management.ManagementClass("Win32_Processor"))
65          {
66              foreach (System.Management.ManagementObject processor in
    processors.GetInstances())
67              {
68                  int AddressWidth = int.Parse(processor["AddressWidth"].ToString
    ());
69                  int Architecture = int.Parse(processor["Architecture"].ToString
    ());
70
71                  if (AddressWidth == 32)
72                      return ProcessorType.x86;
73                  if (AddressWidth == 64 && Architecture == 9)
74                      return ProcessorType.x64;
75                  if (AddressWidth == 64 && Architecture == 6)
76                      return ProcessorType.IPF;
77              }
78          }
79          throw new NotSupportedException();
80      }
81  }
82 }
```

Apêndice B

Abordagem para Compreender Programas Escrevendo Contratos

A abordagem abaixo deve ser executada uma vez a cada escrita de contrato. É importante ressaltar a aplicação do fluxo a seguir:

1. escrever uma pré-condição;
2. executar o teste;
3. escrever uma pós-condição;
4. reexecutar o teste.

A ordem na escrita de uma pré-condição ou pós-condição pode ser alterada devido a assinatura do método ou variável global da classe. Entretanto, o fluxo de escrita de um contrato e execução do teste deve ser aplicado.

Os passos da abordagem são:

1. Em projetos que o desenvolvedor não conhece o programa, escolha um projeto qualquer (caso exista mais de 1 na *solution*) e busque o método que deseja compreender.

2. Escolhido o método, verificar a primeira chamada interna do método. Esta chamada deve ser de um método definido pelo programa.

2.1. Se a primeira chamada tem sobrecarga

2.1.1. O desenvolvedor deve optar pelo método que contêm o menor número de parâmetros. O método deve possuir código no escopo.

2.1.2. Caso existam dois ou mais métodos com apenas um parâmetro, deve-se começar pelo o que foi definido primeiro.

2.2. Se a primeira chamada não tem sobrecarga

2.2.1. Ir até o método raiz.

2.2.2. Escrever a pré-condição ou pós-condição de acordo com a assinatura do método (ponto 2.3).

2.2.3. Ao considerar suficiente os contratos escritos para o método raiz, deve-se voltar para o método que fez a chamada e escrever contratos neste.

2.3. Assinatura do método

PS:. Para as variáveis globais, só existirão pré-condições e pós-condições se estas forem utilizadas no escopo do método.

PS^2:. Caso julgue necessário, variáveis do escopo do método podem ser transformadas em variáveis globais.

2.3.1. Para métodos:

2.3.1.1. Com retorno e com parâmetros – pós-condição e pré-condição.

2.3.1.2. Com retorno e sem parâmetros – pós-condição.

2.3.1.3. Com retorno, sem parâmetros e com variáveis globais - pós-condição (para o retorno e para as variáveis globais) e pré-condição (variáveis globais).

2.3.1.4. Com retorno, com parâmetros e com variáveis globais - pós-condição (para o retorno e para as variáveis globais) e pré-condição (para os parâmetros e para as variáveis globais).

2.3.1.5. Sem retorno e sem parâmetros – nada.

2.3.1.6. Sem retorno e com parâmetros – pré-condição.

2.3.1.7. Sem retorno, sem parâmetros e com variáveis globais - pré-condição e pós-condição para as variáveis globais.

2.3.1.8. Sem retorno, com parâmetros e com variáveis globais - pré-condição (para os parâmetros e para as variáveis globais) e pós-condição (para as variáveis globais).

2.3.2. Definições de pré-condições e pós-condições para as variáveis globais bastam ser feitas apenas uma vez no método raiz.

3. Definir as possíveis pré-condições, sendo estas uma de cada vez seguindo o fluxo da abordagem

3.1. Para variáveis globais

3.1.1. Observar se há inicialização. Considerar a existência de pré-condições para variáveis globais que não são inicializadas.

3.1.2. Definir pré-condições de acordo com o seu tipo.

3.1.3. Observar onde a variável global é chamada e se existe alguma limitação.

4. Para cada pré-condição pertencente ao conjunto de Pré-condições

PS:. não é sempre que uma pré-condição está para uma pós-condição.

4.1. Caminhar no *trace* do método assumindo a pré-condição.

4.2. Gerar um conjunto de pós-condições com os resultados dos *traces*. O conjunto de pós-condições deve envolver argumentos dos métodos e variáveis globais da classe.

5. Para cada pós-condição resultante do *trace*

5.1. Escrever a pós-condição. Esta pode ser escritas com implicações pré-condição $\rightarrow$ pós-condição.

Apêndice C

Questionário Pós-Estudo para Avaliar a Usabilidade da Abordagem

Este apêndice contêm as questões elaboradas e aplicadas com o objetivo de coletar informações a respeito da compreensão dos métodos utilizados no estudo e a opinião dos desenvolvedores em relação a abordagem.

Antes da iniciarmos com o estudo, foi solicitado que os participantes lessem o termo de consentimento explicando qual seria a participação deles na pesquisa. Além disso, foi solicitado o preenchimento do nome, e-mail, tempo de experiência com desenvolvimento, como definem a experiência com programação *C#*, e como definem o conhecimento em *Design by Contrat*. A seguir as demais perguntas foram realizadas:

1. O método GetProcessorArch() verifica algumas informações do computador. Qual das alternativas abaixo corresponde à função do método? (A largura de endereço do sistema operacional e a arquitetura do processador, A largura de endereço do processador, A arquitetura do sistema operacional, A capacidade de cache do processador, Nenhuma das alternativas)

2. Qual a função desempenhada pelo método GetAvailableDB()? (Consultar informações do provedor System.Data.SQLite, Remover informações do provedor System.Data.SQLite, Adicionar informações ao criar um provedor, Atualizar informações do provedor Sysyem.Data.SQLite, Adicionar informações à cada provedor atribuído a variável SQLiteName)

3. A variável invariantName é importante para definir algumas coordenadas do código. Entretanto, o que ela representa? (uma constante, uma propriedade, o nome do banco de dados, uma linha do banco de dados, uma coluna do banco de dados)

4. Analisando a execução do método Get(...) e considerando a expressão invariantName == SQLiteName verdadeira, qual alternativa representa o fluxo do método? 1. Adicionar e carregar a .dll. 2. checar a arquitetura do processador e adicionar o prefixo correspondente. 3. adicionar a pasta SQLite ao diretório raiz. 4. retornar a instância do banco de dados criada a partir do nome da classe correspondente. (2, 1, 3, 4; 3, 2, 1, 4; 4, 2, 3, 1; 1, 3, 2, 4; 3, 4, 1, 2)

5. Você já havia escrito contratos? (sim ou não)

6. Como você classifica o nível de dificuldade para compreender o código proposto, caso não tivesse que escrever contratos? (Muito fácil, Fácil, Razoável, Difícil, Muito difícil)

7. Em sua opinião, qual nível de esforço você teve que dedicar para escrita de contratos? Justifique sua resposta. (Muito leve, Leve, Moderado, Intenso, Exaustivo)

8. Aponte ausência de passos na abordagem que dificultaram a escrita de contratos enquanto você compreendia o código.

9. Aponte benefícios que a abordagem auxiliou na escrita de contratos para compreensão do código.

10. Sugira melhorias para a abordagem.

11. Em resumo, qual sua opinião acerca da usabilidade da abordagem descrita no arquivo .pdf? (Muito fácil, Fácil, Razoável, Difícil, Muito difícil)