Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo de Oliveira Ferreira

Geração automática de suítes de teste da interface com usuário a partir de casos de uso

DISSERTAÇÃO DE MESTRADO

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio.

Orientador: Prof. Arndt von Staa

Rio de Janeiro
Agosto de 2013

Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo de Oliveira Ferreira

Geração automática de suítes de teste da interface com usuário a partir de casos de uso

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio.

Prof. Arndt von Staa
Orientador
Departamento de Informática – PUC-Rio

Prof. Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC-Rio

Prof. Hélio Cortes Vieira Lopes
Departamento de Informática – PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 23 de agosto de 2013

Eduardo de Oliveira Ferreira

Graduado em Engenharia da Computação na PUC-Rio em 2010. Sua principal área de atuação é Engenharia de Software.

Ficha Catalográfica

Ferreira, Eduardo de Oliveira

Geração automática de suítes de teste da interface com usuário a partir de casos de uso / Eduardo de Oliveira Ferreira ; orientador: Arndt von Staa. – 2013.

118 f : il. (color.) ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2013.

Inclui bibliografia

1. Informática – Teses. 2. Engenharia de Software. 3. Casos de uso. 4. Máquina de estados.5. Testes automatizados. 6. Dispositivos móveis. I. Staa, Arndt von. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Para Antonia, minha segunda mãe.

Agradecimentos

Aos meus pais, Jorge Manuel dos Santos Ferreira e Maria Albertina Guimarães de Oliveira, pelo amor, compreensão e confiança em minhas ideais e sonhos.

Ao meu irmão, Fábio de Oliveira Ferreira, pelo companheirismo e amizade nos momentos difíceis.

A minha querida avó, Antonia Gonçalves Guimarães, que faleceu ano passado.

Ao meu orientador, Arndt von Staa, por sua dedicação, orientação, confiança e paciência. Acredito que o caminho para a inovação começa ao permitir que as pessoas possam tentar inovar, respeitando suas ideias, opiniões e instintos.Fico eternamente grato pelo professor Arndt ter sido meu orientador.

Ao meu amigo João Marcos Albuquerque, por ter acreditado em minhas ideias e quebrado certos paradigmas meus.

À CAPES e à PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

A todos os amigos e familiares que de alguma forma contribuíram para a realização deste trabalho.

Resumo

Ferreira,Eduardo de Oliveira;Staa, Arndt von. **Geração automática de suítes de teste da interface com usuário a partir de casos de uso.**Rio de Janeiro, 2013. 118p.Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Espera-se que o desenvolvimento de suítes de teste a partir de modelos possa contribuir substancialmente para a redução do esforço humano e aumentar a eficácia dos testes gerados. Entende-se por eficácia (ideal) dos testes o percentual de defeitos existentes encontrados a partir desses testes. Uma grande parte dessas técnicas baseia-se em máquinas de estado e quase sempre estão voltadas para o teste de funcionalidade. Entretanto, existe a necessidade de se poder testar sistemas altamente interativos, tais como smartphones e tablets, a partir de uma descrição de sua interface humano-computador. O objetivo da dissertação é efetuar uma primeira avaliação de uma técnica voltada para a geração de suítes de teste visando o teste de interfaces gráficas. Para tal,desenvolvemos e avaliamos a eficácia de uma ferramenta, chamada *Easy*, que utiliza casos de uso tabulares e máquina de estados para a geração automática da suíte de testes. Os casos de uso são descritos em linguagem natural restrita. A partir dessa descrição, a ferramenta constrói uma máquina de estado e, a seguir, a utiliza para gerar cenários. Por construção os cenários estarão em conformidade com os casos de uso. Cada cenário corresponde a um caso de teste. Os cenários são apresentados ao usuário em linguagem natural restrita, permitindo a visualização destes antes da geração dos scripts finais de testes. Os scripts gerados são destinados a uma ferramenta de execução automatizada voltada para o teste de interfaces gráficas. Neste trabalho, utilizou-se a ferramenta UI Automation, responsável pela execução de testes em aplicações destinadas ao iOS, sistema operacional de iPhone, iPad e iPod Touch. A eficácia do processo foi avaliada em uma aplicação real, disponível na loja virtual de aplicativos App Store. Além disso, foram realizados testes de IHC afim de avaliar a influência no custo da produção da suíte de teste.

Palavras-chave

Engenharia de software; casos de uso; máquina de estado; testes automatizados; dispositivos móveis

Abstract

Ferreira, Eduardo de Oliveira; Staa, Arndt von (Advisor). **Automatic generation of user interface test suites specifiedby use cases.**Rio de Janeiro, 2013. 118p. MSc. Dissertation– Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

It is expected that the development of test suites from models can contribute substantially to reducing the human effort and to increase the effectiveness of the generated tests. Means for tests' effectiveness (ideal) the percentage of existing defects found by these tests. Most of these techniques is based on state machines and mostly directed to testing the functionality. However, there is a need to be able to test highly interactive systems, such as smartphones and tablets, from a description of its human computer interface. The goal of the dissertation is to make a first evaluation of a technique aimed to generate test suites for test of human computer graphic interface. For this purpose was developed and evaluated its effectiveness, a tool called Easy, using use cases tabular and state machine for the automatic generation of the suite tests. The use cases are described in natural language restricted. From this description, the tool builds a state machine, and then uses this to generate scenarios. By construction scenarios will be in accordance with the use cases. Each scenario corresponds to a test case. The scenarios are presented to the user in natural language, allowing the visualization of them before the generation of the final scripts tests. The generated scripts are intended to a running automated tool geared to testing graphical interfaces. In this work, we used the UI Automation tool, responsible for running tests on applications for the iOS, operational system for iPhones, iPads and iPod touchs. The effectiveness of the procedure was evaluated in a real application, available in the online store applications App Store. In addition, HCI tests were performed in order to evaluate the influence on the cost of production of the test suite.

Keywords

Software engineering; use cases; state machine; automated tests; mobile devices

Sumário

Lista de Figuras

Lista de Tabelas

1
Introdução

Teste é uma importante técnica de controle de qualidade de software. Porém, a utilização de testes de software nos projetos tende a ser custosa, muitas vezes responsável por 50% a 60% do total do custo de desenvolvimento [Memon, 2002]. Sabe-se que 50% dos softwares postos em uso no mercado contêm defeitos não triviais [Boehm e Basili, 2001]. Para aumentar a corretude dos softwares, foram desenvolvidas técnicas de controle de qualidade de software, como o *Test-driven development* (TDD) [Beck, 2004]. O TDD possui as seguintes vantagens [Staa, 2011]:

- Facilita o desenvolvimento incremental.
- Facilita corrigir eventuais problemas nas especificações.
- Reduz o esforço de teste ao levar-se em conta o custo de reteste após corrigir ou evoluir o módulo.
- Reduz o estresse do desenvolvedor.

Em [Janzen, 2008], afirma-se que o TDD não resume-se exclusivamente a automação de testes, mas também a uma prática de design de software. Observando as vantagens geradas pela utilização do TDD, fica claro que esta técnica é vantajosa no desenvolvimento de um produto, porém sua implementação em projetos é custosa, exigindo tempo e dedicação dos desenvolvedores. Outra desvantagem do TDD é ser fortemente voltado para testes de unidades (módulos). Em relação ao usuário, os comportamentos esperados por um software são um conjunto de interações entre módulos, e não somente a ação de uma unidade do sistema.

Outro ponto importante é a predominância atual de softwares utilizando interfaces gráficas para interagirem com o usuário. Sabe-se que 60% do código fonte de um softwareestão relacionados à construção de interface e tratamento de sua lógica [Memon, 2002], e que 66% das falhas encontradas em softwares com interface gráfica são defeitos na interface [Perry, 1987]. Os módulos que implementam os elementos gráficos e as interações do usuário com a interface são

mais custosos de serem testados do que módulos que implementam puramente lógicas de negócio. Uma das propostas para resolver estes problemas é partir para o desenvolvimento dirigido por comportamento, conhecido como *Behavior Driven Development* (BDD) [North, 2006].

O BDD surgiu como uma extensão ao TDD, tendo como objetivo identificar as funcionalidades de uma aplicação pelos seus comportamentos, utilizando linguagem natural. Essa abordagem proporciona uma especificação de requisitos mais simples e intuitiva tanto para clientes quanto para desenvolvedores. A linguagem utilizada por BDD é uma linguagem natural semiestruturada. Todavia, a linguagem proposta por [North, 2006] é capaz de representar diversos cenários, porém estes cenários podem não ser classificados como derivações de um mesmo evento, prejudicando a visualização do problema como um todo pelo usuário.

Em [Caldeira, 2010], foi criado uma ferramenta de geração de casos de teste BDD, a partir de casos de uso. Segundo [Jacobson, 1992], um sistema pode ser especificado por meio de casos de uso. A estratégia utilizada por [Caldeira, 2010] foi usar casos de uso para descrever casos de testes. Essa abordagem é interessante, pois facilita o uso da ferramenta pelo usuário devido ao caso de uso pertencer ao padrão UML, mundialmente utilizado no meio empresarial e acadêmico. Outro ponto importante é a utilização de tabela de decisão para a geração dos cenários de um caso de uso. Uma grande parte das técnicas de geração de casos de teste utiliza explicitamente ou implicitamente tabelas de decisão [Myers, 2004].

Uma tabela de decisão é uma ferramenta que tem como objetivo abordar as combinações de condições necessárias para a execução de uma ou mais ações [Myers, 2004]. Tabelas de decisão são divididas em duas partes: condições e ações. O campo de condições é responsável por listar as condições existentes e possíveis valores a serem tomados para a execução de um conjunto de ações. O campo de ações é responsável pelas ações a serem efetuadas a partir da combinação de condições. No caso da utilização para testes, as ações são encaradas como oráculos, respostas do sistema respectivas a um conjunto de condições ocorridas. As colunas da tabela simbolizam os cenários de um caso de testes [Lachtermacher, 2009].

A utilização de tabela de decisão mostra-se uma forte abordagem para a geração de massa de testes, gerando resultados satisfatórios. Além de [Caldeira,

2010], [Lachtermacher, 2009] também utilizou tabela de decisão para geração de testes, porém sem a utilização de casos de uso. Esses dois trabalhos criaram editores de tabela de decisão que permitem ao usuário criar cenários e passar valores para estes. É possível observar nessas referências que as ferramentas facilitam a geração de testes. Todavia, tabelas de decisão podem se tornar extremamente grandes para serem editadas, dependendo do número de condições a serem abordadas. Caso o usuário trabalhe com muitas condições, os cenários respectivos exigirão um grande esforço para que sejam completamente preenchidos. Seria interessante minimizar o esforço do usuário ao criar os cenários. Utilizar uma camada acima do método proposto por [North, 2006], utilizado em [Caldeira, 2010] e [Lachtermacher, 2009] ao descrever os casos de testes, onde cenários não precisassem ser descritos, mas sim gerados automaticamente e valorados.

Outra estratégia existente para geração automática de testes é a utilização de máquina de estados. Segundo [Yuan e Memon, 2010], modelos de máquinas de estados são os métodos mais populares usados para teste de software. Os comportamentos de um software são transformados em estados abstratos ou concretos. Esses estados formam um diagrama, semelhante a um grafo, onde é possível migrar de um estado ao outro. Um modelo bastante utilizado é o *Finite State Machine Models* (FSM). Contudo, máquina de estados pode se tornar complexa à medida que o número de comportamentos do sistema a ser testado aumenta.

Sabe-se que testes tendem a ser pouco eficazes devido à inadequação das ferramentas existentes [NIST, 2002]. Um desses fatores de inadequação é a interface com o usuário das ferramentas. Segundo [Norman, 1991] e [Norman, 1993], a tecnologia deve ser projetada com o objetivo de ajudar as pessoas a serem mais hábeis, eficientes e inteligentes. A interação humano-computador (IHC) nas ferramentas de geração de testes deveria ser mais explorada. Acredita-se que uma ferramenta de teste dotada de uma interface simples e objetiva possa diminuir a inércia dos desenvolvedores ao utilizarem testes codificados em seus softwares. Outro ponto importante é analisar o quanto a IHC interfere no custo de geração de testes ao usuário. Ferramentas com interfaces mal elaboradas dificultam e prejudicam os usuários ao criarem testes para as suas aplicações.

Em suma, existem diversos fatores que podem tornar uma ferramenta de geração de testes mais adequada ao usuário. Acredita-se que a utilização de casos de uso para descrever os casos de testes é uma abordagem promissora. A utilização de tabela de decisão pode ser uma abordagem bem sucedida, porém seria interessante uma abordagem que diminuísse ainda mais o esforço do usuário ao gerar cenários, valorá-los e criar testes. Acredita-se que a utilização de máquina de estados possa ser uma alternativa eficiente, caso esta máquina possa ser montada de forma automática a partir de uma especificação. A IHC das ferramentas de teste também deveria ser mais explorada, gerando assim artefatos mais simples e objetivos, capazes cada vez mais de serem utilizadas por desenvolvedores.

1.1 Objetivo

O objetivo da dissertação é desenvolver e avaliar a eficácia de uma ferramenta que utiliza máquina de estados para a geração automática de testes de interface gráfica a partir de casos de uso.

Optamos por utilizar casos de uso [Cockburn, 2000] para a descrição dos testes devido à possibilidade de redução da curva de aprendizado dos usuários da ferramenta, pois pertencem ao padrão UML, mundialmente utilizados no desenvolvimento de software. Além disso, os casos de uso são descritos com linguagem natural restrita, o que simplifica a compreensão por parte dos desenvolvedores e clientes, sendo este um dos objetivos do BDD, porém limita a liberdade do redator devido a restrições sintática e de tempos verbais. A existência de um padrão linguístico na descrição dos casos facilita a interpretação da ferramenta sobre a intenção do texto passado.

A máquina de estados foi a estratégia selecionada devido à sua facilidade de geração de cenários derivados de um caso de uso. A automatização da construção da máquina de estados e da geração de cenários é fundamental para a redução do esforço do usuário. Dessa forma, ao transformarmos automaticamente cenários em casos de testes codificados, o usuário agiliza a produção de testes de seu projeto ao não precisar implementá-los.

Outro ponto importante é a interface gráfica da ferramenta. Acreditamos que a simples utilização de máquina de estado no projeto não traria grandes vantagens sem o auxílio de uma interface gráfica que permitisse ao usuário acompanhar etapas da geração dos testes.

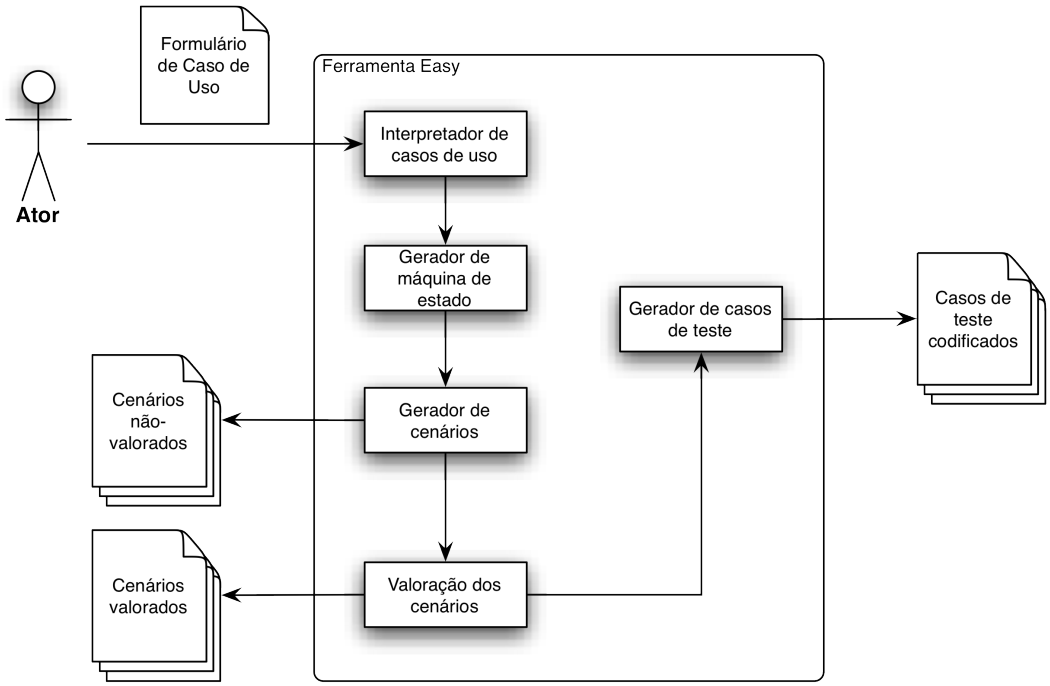


Figura 1: Etapas da ferramenta.

A ferramenta apresenta quatro etapas na produção dos testes. A primeira é o preenchimento do formulário de caso de uso. Ao finalizar o formulário, é iniciada a segunda etapa, onde são exibidos ao usuário os cenários gerados pela ferramenta a partir das informações contidas no caso de uso. Os cenários são descritos usando linguagem natural restrita, não contendo detalhes de programação. O objetivo dessa etapa é permitir ao usuário observar os cenários derivados de seu caso de uso, podendo selecionar quais destes deverão se transformar em casos de teste codificados. A terceira etapa é semelhante à segunda, porém apresenta os cenários com valores concretos a serem usados como valores de entrada no sistema. Muitos desses valores concretos são gerados aleatoriamente com o intuito de aumentar a abrangência dos testes, proporcionando diferentes valores de entrada ao sistema a ser testado. Nesta etapa, o usuário poderá editar os valores concretos e, semelhante à segunda etapa, selecionar quais cenários deverão continuar no processo de geração de testes. Por fim, a última etapa é a geração dos casos de testes codificados. Importante enfatizar que somente cenários que permaneçam até o fim do processo serão transformados em *scrips* de testes. Acreditamos que as

interfaces gráficas das ferramentas de testes sejam um dos pontos responsáveis pela redução do esforço do usuário na geração de testes. Também apresentamos na dissertação resultados de testes de IHC com usuários utilizando a ferramenta, avaliando as vantagem geradas pela interface gráfica na produção dos casos de testes. Na Figura 1 são ilustradas as etapas da ferramenta.

Um dos objetivos da ferramenta *Easy* é ser genérica em relação aos tipos de linguagens de programação dos *scripts* de testes gerados. A *Easy* foi modelada para permitir que desenvolvedores criem módulos de novas linguagens e consigam utilizar a ferramenta em seus contextos. Outra ponto importante é a possibilidade de inclusão de novos *widgets*. Dessa forma, a ferramenta tem capacidade de evolução em relação à medida que novos *widgets* forem criados e ao surgimento de novas linguagens. Atualmente, *Easy* tem capacidade de gerar *scripts* de testes de interface gráfica para aplicações para *iOS* que utilizem *UIKit* [Apple, 2007], biblioteca de interface gráfica padrão do *iOS*. O *iOS* é o sistema operacional utilizado por *iPhone*, *iPad* e *iPod Touch*.

Este documento está organizado em oito capítulos: no primeiro introduzimos o assunto e apresentamos as motivações e objetivos para este trabalho; no segundo mostramos o estado da arte; no terceiro apresentamos o processo realizado pela ferramenta na geração de testes; no quarto discutimos em detalhes as etapas da ferramenta e as soluções utilizadas; no quinto apresentamos experimentos realizados em aplicações reais; no sexto abordamos uma avaliação realizada com usuários reais; no sétimo são apresentadas as conclusões; no oitavo, último capítulo, comentamos sobre os trabalhos futuros.

2
Estado da arte

Existem diversos arcabouços de execução de testes automáticos para diferentes linguagens de programação inspiradas no método *Behavior Driven Development* (BDD). Podemos citar alguns: *JBehave* [North, 2006] para Java, *Cpp-Spec* [Berris, 2009] para C++, *RSpec* [Astels, 2006] para Ruby, *PHPSpec* [Brady, 2008] para PHP, *Cedar* [Labs, 2011] e *Kiwi* [Ding, 2011] para Objective-C. Essas ferramentas permitem aos desenvolvedores gerarem cenários utilizando linguagem natural, um grande avanço comparado aos arcabouços inspirados no método *Test-driven development* (TDD). Porém, cada ferramenta adota uma linguagem para a descrição dos cenários a serem testados, impondo ao usuário uma curva de aprendizado ao tentar utilizá-las. A utilização de um padrão conhecido, como casos de uso do UML, minimizaria o tempo de aprendizado.

Tentando minimizar o esforço do usuário, alguns arcabouçoscomeçaram a oferecer a possibilidade de gerar casos de testes em relação aos elementos de interfaces gráficas dos *softwares*. Podemos citar: *UI Automation* [Apple, 2011] para *UIKit* [Apple, 2007], biblioteca de elementos gráficos de interface para *iOS* [Apple, 2007]; e *UI Unit* [Galdino, 2010] para *Qt* [Nokia, 2007], biblioteca pra C++. Esses arcabouçospermitem aos desenvolvedores selecionarem diretamente os *widgets* envolvidos em uma ação, e simularem com mais realismo os comportamentos do usuário aos elementos de interface. A linguagem utilizada na descrição dos casos de testes é mais simples do que as adotadas pelas ferramentas sem suporte às interfaces gráficas, porém não se pode afirmar que sejam linguagens naturais ao usuário, estando ainda distantes da proposta do BDD [North, 2006]. Além disso, não há uma forma de identificar se alguns cenários são derivados de um mesmo evento. As ferramentas tratam os cenários como sendo completamente distintos, não esclarecendo que todos podem representar um mesmo evento. Outro problema é o esforço do usuário ao criar os cenários. Caso seja necessário criar diversos cenários de um mesmo evento, o usuário precisa implementar cada um deles. A geração dos cenários poderia ser automática em

relação à especificação de um evento, reduzindo assim atividades repetitivas pelo usuário.

Alguns arcabouçosque tratam de elementos de interface gráfica utilizam o recurso de *capture and replay* [Araujo, 2009] para geração dos testes. Podemos citar: *Squish* [Froglogic, 2008], *JRapture* [Steven, Chandra, Fleck e Podgurski, 2000], *Selenium IDE* [SeleniumHQ, 2008]. O recurso *capture and replay*, inicialmente, aparenta ser uma abordagem simples e rápida aos usuários. Para utilizar essa funcionalidade, o desenvolvedor precisa simular as ações do usuário para um evento, enquanto a ferramenta registra todas as ações ocorridas. Após finalizar o registro, é gerado um *script* responsável pelo teste. O problema ao utilizar ferramentas desse gênero é a dificuldade de coevolução desses *scripts*. Os *scripts* de testes gerados automaticamente são difíceis de serem editados pelos desenvolvedores, sendo que, em diversos momentos, torna-se necessário regravar os testes devido à necessidade de inclusão de novas ações. Resumindo, a manutenção de casos de testes gerados por *capture and replay* costuma ser custosa.

Buscando reduzir o trabalho com geração de cenários, a ferramenta criada por [Lachtermacher, 2009] permite que o usuário gere casos de teste de interface utilizando tabela de decisão. A ferramenta oferece um editor de tabela de decisão, permitindo que o usuário possa criar cenários com facilidade. Outra abordagem que também utiliza tabela de decisão é encontrada em [Caldeira, 2010]. Neste trabalho, o usuário descreve casos de teste por meio de casos de uso. Ao passar um formulário com o layout proposto por [Staa, 2010] e adaptado de [Cockburn, 2000], a ferramenta cria uma tabela de decisão com as condições e os oráculos descritos no caso de uso. Cabe ao usuário criar somente os cenários a serem testados. A utilização de tabela de decisão é uma abordagem forte para a geração de testes, porém a criação de cenários ainda pode ser custosa ao usuário. Caso sejam incluídas novas condições na tabela, o número de cenários aumenta exponencialmente. Ao aumentar o número de cenários, aumenta também o esforço do usuário ao passar valores de entrada para cada um deles. Por exemplo, se possuirmos uma tabela com 10 condições, onde cada uma delas pode assumir dois tipos de valores – verdadeiro ou falso – teremos 1024 cenários a serem editados pelo usuário. Não há dúvida quanto ao potencial de tabelas de decisão para geração de testes, porém a criação de cenários não deveria ficar

obrigatoriamente sob responsabilidade do usuário. Caso os cenários sejam gerados automaticamente, o usuário ganha uma considerável vantagem.

Outra abordagem para geração automática de casos de teste é a utilização de máquina de estados, sendo esta a mais utilizada por ferramentas de teste [Yuan e Memon, 2010]. Segundo [Staa, 2013], uma máquina de estado é composta de estados e arestas. Os estados representam pontos em que a aplicação em teste pode estar dependendo dos eventos ocorridos. As arestas são responsáveis por conectar estados e definem condições a serem satisfeitas que permitam a transição de um estado para outro. A utilização de máquinas de estados para testes é intuitiva, pois um teste consiste em um conjunto de eventos realizados pelo usuário ou sistema com o objetivo de se completar um processo. Esses eventos possuem uma cronologia que precisa ser respeitada, caso contrário o processo pode não ser executado corretamente. Segundo [Yuan e Memon, 2010], existem diversos modelos de máquinas de estados: *Finite State Machine Models* (FSM) [Barnett, 2003] [Hong, 1997] [Apfelbaum, 1997] [Farchi, 2002], *UML Diagram-based Models* [Lucio, 2004] e *Markov Chains* [Whittaker, 1997]. Esses modelos são geralmente derivações do FSM, mudando principalmente a abordagem de descrição dos estados em relação às ações ocorridas no sistema a ser testado. Existem também referências sobre a utilização de máquina de estados para testes de interface gráfica [White e Almezen, 2000] [Belli, 2001]. A estratégia utilizada neste caso diverge ligeiramente da apresentada por [Staa, 2013], pois trata cada estado de uma máquina como um evento de interface direcionado a um *widget* da interface gráfica de uma aplicação, sendo este evento realizado pelo usuário ou pelo próprio sistema. As arestas desse tipo máquina simplesmente indicam as direções possíveis que o ator do evento poderá seguir ao interagir com a interface gráfica da aplicação, não sendo obrigatória a presença de condições a serem satisfeitas para se percorrer uma aresta.

A utilização de máquina de estados para geração de testes também mostra-se uma abordagem eficaz, porém possui problemas semelhantes às outras ferramentas abordadas neste documento. A criação dos estados da máquina exige a utilização de linguagens específicas e criadas exclusivamente para as respectivas ferramentas. Desta forma, existe uma curva de aprendizado que o usuário deve enfrentar para utilizá-las. Além disso, muitas ferramentas não utilizam linguagens naturais para a descrição dos estados, prejudicando o entendimento de usuários

sem conhecimento em programação. Outro problema é a possibilidade do grande número de estados no grafo. Semelhante ao problema da utilização de tabelas de decisão quando o usuário utiliza diferentes tipos de cenários, o esforço necessário para a execução desta tarefa será grande. Máquina de estados parece ser uma abordagem intuitiva para a geração de casos de teste, porém também não soluciona o grande esforço do usuário com grandes números de cenários, caso sua construção seja manual.

Em relação à IHC das ferramentas de testes, não foram encontradas referências que abordem este tema. Interessante observar que os trabalhos relacionados à geração de testes automáticos sempre abordam o custo do usuário ao utilizar diferentes métodos relacionados à criação de testes. Todavia, a IHC de uma ferramenta pode ser um elemento definitivo para a diminuição do esforço envolvido ao gerar testes. A produção de diferentes abordagens para o problema de geração de testes é, sem sombra de dúvida, importante, porém deveriam existir mais estudos direcionados a interfaces de ferramentas de testes.

3
Processo

Neste capítulo será apresentado o processo de geração de testes da ferramenta. O processo inicia-se com a descrição de um formulário de casos de uso pelo usuário. Ao término do preenchimento do formulário, este é interpretado e passado para o gerador de máquina de estados da ferramenta, que constrói um grafo respectivo ao caso de uso passado. Tendo a máquina de estados montada, começa a identificação dos possíveis cenários a serem gerados. Todos os cenários são apresentados ao usuário por meio da interface gráfica da ferramenta. Nessa etapa o usuário poderá selecionar quais cenários deverão permanecer no processo de geração de testes. Em seguida, os cenários escolhidos são valorados com valores aleatórios ou passados pelo próprio usuário. Esses valores são utilizados como entradas no sistema a ser testado. Na etapa de valoração, o usuário também poderá escolher quais cenários, agora com valores de entrada definidos, deverão transformar-se em casos de testes codificados. Ao autorizar a produção dos testes, todos os cenários presentes até o fim do processo são transformados em *scripts* de testes.

Outra funcionalidade da ferramenta é a segmentação de casos de uso com muitos elementos de interface. Esta etapa consiste na transformação de casos de uso grandes em casos menores com dependências cronológicas entre si. Por exemplo, para que um caso de uso B seja executado, é necessário que outro caso A ocorra antes. Assim, B depende cronologicamente de A. Essa funcionalidade tem como objetivo reduzir o número de cenários de casos de uso que utilizem diversos elementos de interface. Acreditamos que esse recurso aumente a eficiência e objetividade dos testes.

O capítulo esta dividido em seis partes: a primeira é uma breve introdução do processo de geração de testes; na segunda abordamos como é realizada a descrição dos casos de uso; na terceira explicamos como ocorre a construção da máquina de estados em relação a um caso de uso; na quarta comentamos a geração de cenários; na quinta apresentamos a geração dos *scripts* de testes; e por

fim, na sexta parte, mostramos como deve ser feita a segmentação de casos de uso.

3.1
Descrição de casos de uso

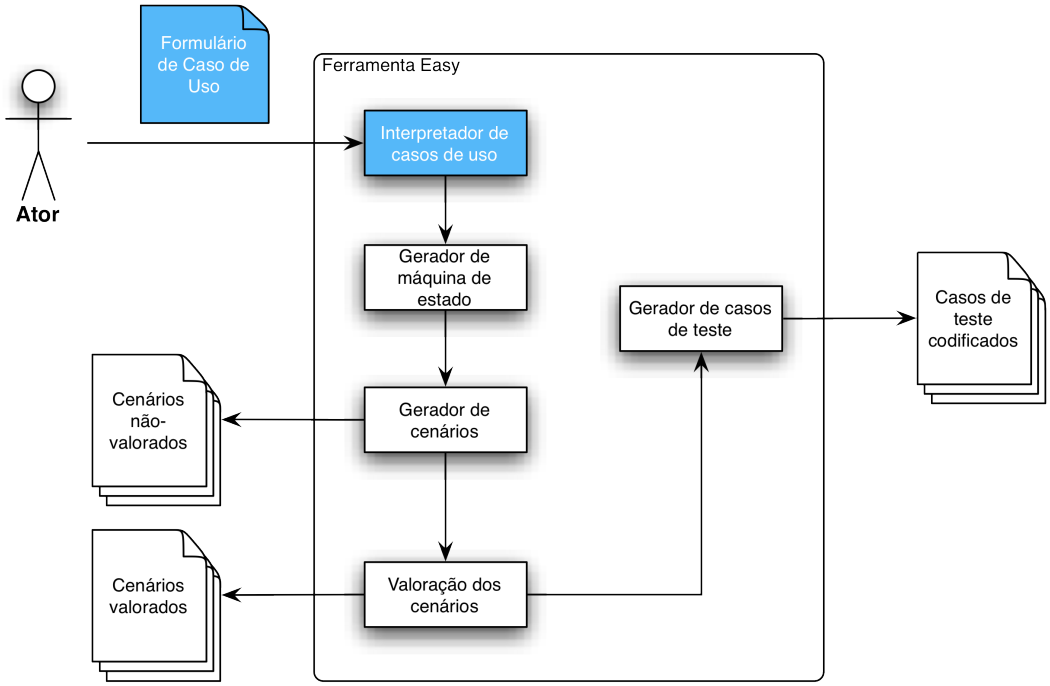


Figura 2: Etapa da ferramenta relacionada à descrição dos casos de uso.

A primeira etapa da ferramenta consiste na descrição dos casos de uso. Devido à existência de diversos modelos de descrição de casos de teste, acreditamos que a melhor opção seria utilizar um modelo já conhecido pelo usuário. Os casos de uso possuem grande potencial para descrição de testes e pertencem ao padrão UML, mundialmente utilizado no meio empresarial e acadêmico. Além disso, [Jacobson, 1992] afirma que um sistema pode ser especificado por meio de casos de uso, e em [Caldeira, 2010] é apresentada uma ferramenta de geração de casos de testes que utiliza casos de uso para descrevê-los. Sendo assim, acreditamos que o uso de casos de uso é benéfico e simples ao usuário para a geração de testes.

Os casos de uso devem ser descritos na interface gráfica da ferramenta como um formulário com o layout proposto por [Staa, 2010] e adaptado de [Cockburn, 2000]. O formulário utiliza uma linguagem natural restrita. Este formulário também foi utilizado em [Caldeira, 2010]. De todos os campos do formulário original, os utilizados pela ferramenta são:

- Nome
- Pré-condição
- Sistema
- Ator
- Elementos de interface
- Fluxo principal
- Fluxo alternativo
- Regras de negócio

O campo *Nome* especifica o nome do caso de uso. O campo *Pré-condição* é responsável por identificar se o caso de uso corrente depende de outro caso de uso para sua execução. O campo *Sistema* é onde especificamos o tipo de plataforma que os casos de testes serão executados, como, por exemplo, *iOS*. No campo *Ator* é passada uma referência sobre o usuário atuante no caso de uso. Normalmente utilizamos o substantivo *Usuário* neste campo, porém fica a critério do desenvolvedor o termo usado. Em *Elementos de interface* são cadastrados quais *widgets* participam do caso de uso. Como os casos de testes são direcionados à interface gráfica, esse campo é essencial para a ferramenta identificar os elementos de interface utilizados nos testes. O campo *Fluxo principal* é onde o desenvolvedor especifica o fluxo das ações do usuário aos elementos de interface do sistema a ser testado. Também é esperada uma resposta do sistema, a qual é utilizada como oráculo nos testes. No *Fluxo alternativo*, o desenvolvedor aponta possíveis alternativas ao fluxo principal. Qualquer ação do usuário que possa ter uma resposta do sistema diferente da esperada pelo fluxo principal deve ser informada nesse campo. Por fim, temos o campo *Regras de negócio*, onde o desenvolvedor especifica detalhes dos elementos de interface usados no caso de uso.

A linguagem utilizada no fluxo principal possui um padrão para facilitar a identificação dos elementos para a geração dos testes. Estes elementos são: ator da ação, o tipo de elemento de interface utilizado e o nome deste elemento. Este padrão é semelhante ao utilizado em [Caldeira, 2010] e apresentado nas Figuras 3 e 4.

<Número da ação><Ator><Prefixo><Ação><Sufixo><Nome do elemento na interface>

Figura 3: Padrão de descrição de um passo no fluxo principal.

1. Usuário preenche campo Nome.

Figura 4: Exemplo de passo no fluxo principal.

Para descrever os passos é necessário seguir um padrão de elementos textuais para a ferramenta ser capaz de identificar os tipos de *widgets* usados nos casos de uso. Cada *widget* possui ações, prefixos e sufixos que permitem ao usuário descrever as ações ocorridas dentro de um caso de uso. Na Tabela 1 apresentamos esses elementos textuais para três tipos de *widgets*.

Elementos de interface	Prefixo	Ação	Sufixo
Botão	não <*vazio*>	clica	<*vazio*> no botão em no na
Caixa de entrada de texto	<*vazio*>	preenche	<*vazio*> campo o campo o campo de texto o
Legenda	não <*vazio*>	mostra exibe	<*vazio*> o valor a mensagem mensagem

Tabela 1: Tabela com padrão de descrição dos elementos de interface.

Na Figura 4 podemos observar que *Usuário* é o ator da ação. O elemento de interface utilizado pelo ator é *Nome*, um *widget* do tipo *Caixa de entrada de texto* devido à utilização da palavra *preenche* no passo, segundo a Tabela 1. O fluxo principal também possui passos onde o ator é o próprio sistema testado. No intuito de todo teste possuir um oráculo para sua validação, o último passo de um fluxo principal deve ser sempre a resposta do sistema. Na Figura 5 apresentamos o padrão quando o ator é o sistema.

<Número da ação><Ator><Prefixo><Ação><Sufixo>*"***<Resposta do sistema>***"***

Figura 5: Padrão de descrição quando o ator é o sistema a ser testado.

1. Usuário clica no botão Excluir.
2. Sistema exibe “Exclusão bem sucedida”.

Figura 6: Exemplo de fluxo principal contendo uma resposta do sistema.

A Figura 6 é um exemplo de um caso de uso onde o usuário pretende excluir algum elemento. Após o usuário clicar no botão *Excluir*, o sistema retorna uma mensagem confirmando a exclusão. No passo 2, observamos que *Sistema* é o ator da ação. Também podemos observar que *Sistema* é um *widget* do tipo *Legenda*, pois foi utilizada a ação *exibe* em sua descrição. Em seguida, observamos a mensagem *“Exclusão bem sucedida”*. Essa mensagem será utilizada como oráculo nos casos de testes relacionados ao fluxo principal. A descrição do fluxo principal tende a ser simples e clara, sem a utilização de valores de entrada reais, para tornar a leitura do caso de uso simples, tanto para desenvolvedores quanto para clientes que não possuem conhecimento de programação. Todavia, o passo relacionado ao retorno do sistema, o último passo do fluxo principal, deve possuir um valor específico de resposta do sistema. Este passo deve ser o único do fluxo principal a possuir um valor específico. O uso de aspas (“) em uma expressão indica a utilização valores reais.

Para exemplificarmos como descrevemos um fluxo principal de um sistema, vamos utilizar a janela de autenticação apresentada na Figura 7. Na imagem é possível ver três elementos de interface através das quais o usuário poderá interagir com o sistema: as caixas de entrada de texto respectivas as legendas *Nome* e *Senha*, e o botão *Entrar*. Vamos considerar que o sistema tenha um quarto elemento de interface que informa a situação da autenticação ao usuário. Esse elemento seria do tipo *Legenda* e o chamaremos de *Resposta do Sistema*.

Figura 7: Exemplo de janela de autenticação.

Após identificarmos os tipos dos elementos de interface presentes na ferramenta, iniciamos a descrição do fluxo principal. Nos casos de uso, o fluxo principal é utilizado para definir a sequência de ações mais comuns ao sistema. No caso de uma janela de autenticação, é esperado que o usuário consiga identificar-se com o sistema. Qualquer eventualidade ou erro não deve ser expresso no fluxo principal, mas sim no fluxo alternativo. Sendo assim, na Figura 8 é apresentado o fluxo principal respectivo à janela de autenticação da Figura 7 no padrão aceito pela ferramenta, onde *Nome* e *Senha* são as caixas de entrada de texto, *Entrar* é o botão, *Sistema* é a legenda onde o sistema informa ao usuário sua situação e *Usuário* é o ator do caso de uso.

1. Usuário preenche o campo Nome.
2. Usuário preenche o campo Senha.
3. Usuário clica no botão Entrar.
4. Sistema exibe “Autenticação com sucesso”.

Figura 8: Exemplo de fluxo principal de uma janela de autenticação.

Caso exista alguma possibilidade alternativa de ação referente ao fluxo principal, devemos passá-la no campo de fluxos alternativos. Em um sistema de autenticação, é provável que ocorra a tentativa de entrada de usuários desconhecidos. Esse caso não pertence ao fluxo natural de uma tela de autenticação, mas sim é uma eventualidade ocorrida no sistema. No exemplo mostrado na Figura 8, o usuário consegue passar pela autenticação com sucesso. Porém, caso o usuário passe um nome desconhecido ao sistema, a autenticação não será concluída e o sistema informará ao usuário que o nome é desconhecido. Para esse caso, deve-se criar um fluxo alternativo. A Figura 9 apresenta uma alternativa ao passo *1* do fluxo principal da Figura 8, onde o usuário passa ao campo *Nome* um valor desconhecido pelo sistema. Assim como o fluxo principal, todo fluxo alternativo também precisa de uma resposta do sistema respectiva ao seu caso. Ao criar um fluxo alternativo, o desenvolvedor precisa informar qual será o retorno do sistema relativo ao passo alternativo. Na Figura 10 é mostrado um exemplo de retorno do sistema para o passo da Figura 9.

A1. Usuário preenche campo Nome com valor desconhecido.

Figura 9: Exemplo de passo no fluxo alternativo.

A1.R. Resposta do Sistema exibe “Usuário com nome desconhecido”.

Figura 10: Exemplo de retorno do sistema de um fluxo alternativo.

Um detalhe importante e ainda não mencionado é a expressão *com valor desconhecido* utilizada na Figura 9. Em prol de tornar a descrição dos casos de uso na ferramenta simples, assumimos que todo *widget* possui *valores abstratos*. Os *valores abstratos* são utilizados com o intuito de criar uma camada de abstração acima dos possíveis valores de entrada reais ou comportamento destinados aos *widgets*. Peguemos de exemplo o valor *desconhecido* utilizado na Figura 9. O elemento de interface *Nome* é do tipo *Caixa de entrada de texto*. Acreditamos que uma caixa de entrada de texto possa receber três tipos de valores abstratos: *conhecido*, *desconhecido* e *vazio*. O valor do tipo *conhecido* seria todos os casos de valores entrada conhecidos pelo sistema. No caso de uma janela de autenticação, seria o nome de um usuário já cadastrado no sistema. O valor *desconhecido* seria para valores de entrada que não são reconhecidos pela autenticação, como um usuário sem permissão. Por fim, o valor *vazio* é utilizado quando o usuário não preencher o campo. Importante observar que os *valores abstratos* tornam simples a leitura dos casos de uso aos desenvolvedores e clientes. Ao mencionar que o usuário irá preencher uma caixa de entrada de texto com valor *conhecido*, sabemos que, no mínimo, o sistema será capaz de reconhecer o valor de entrada passado pelo usuário. Na Figura 11 é apresentado o padrão de descrição quando passado um *valor abstrato* para uma caixa de entrada de texto.

<Número da ação><Ator><Prefixo><Ação><Sufixo><Nome do elemento na interface>
*com valor***<valor abstrato do elemento de interface>**

Figura 11: Padrão de descrição de caixa de entrada de texto no fluxo alternativo.

Todos os *widgets* cadastrados na ferramenta *Easy* possuem *valores abstratos*, sendo estes específicos a cada elemento de interface. Se pegarmos como exemplo um *widget* do tipo *Botão*, podemos atribuir a ele dois *valores abstratos*: *clicado* e *não clicado*. No caso de um interruptor, seus *valores abstratos* seriam: *ligado* e *desligado*. A *Caixa de entrada de texto* é um dos que possuem maior número de *valores abstratos*, devido à grande possibilidade de entradas a serem passadas pelo usuário. A Tabela 2 apresenta os *valores abstratos*

de um grupo de widgets que acreditamos serem os mais utilizados nos softwares disponíveis no mercado.

Elementos de interface	Valores abstratos
Botão	clicado não clicado
Caixa de entrada de texto	conhecido desconhecido vazio
CheckBox	selecionado não selecionado
Interruptor	ligado desligado
Legenda	exibe
RadioButton	selecionado não selecionado

Tabela 2: Tabela com valores abstratos respectivo aos widgets.

Outro ponto importante é que o padrão de descrição dos *widgets* no fluxo alternativo varia, nem sempre semelhante ao da Figura 11. Elementos de interface do tipo *Botão*, *Interruptores*, *CheckBox* e *RadioButton* podem ser descritos no fluxo alternativo sem a expressão com valor, utilizada nas caixas de entrada de texto. Na Figura 12 apresentamos um exemplo de um fluxo alternativo ao fluxo principal da Figura 8, onde o usuário não clica no botão *Entrar*. Neste caso, ao utilizarmos a expressão não clica no passo, indicamos que o usuário não irá interagir com o botão.

A2. Usuário não clica no botão Entrar.

Figura 12: Exemplo de passo no fluxo alternativo para o botão Entrar.

O único *widget* que possui somente um *valor abstrato* é a *Legenda*. Optamos por não incluir o *valor abstratonão exibe* devido a este não acrescentar uma vantagem na geração de teste como oráculo. A *Legenda* é um *widget* que não

interage com ações do usuário, sendo somente utilizado pelo sistema. Não é pressionado como um botão, nem editado como uma caixa de entrada de texto. Seu papel exclusivo é exibir conteúdos textuais na interface gráfica. Caso não seja necessária a exibição deste conteúdo, a legenda recebe um valor vazio ou é retirada da interface gráfica. O problema com o *valor abstrato não exibe* é o que este representa aos testes. Afirmar que uma legenda não exibe uma determinada mensagem, significa que poderá ser exibida qualquer informação, menos a apontada na descrição. Acreditamos que não existam vantagens à corretude do roteiro de teste em criar casos de testes que não possuem um oráculo bem definido.

Inicialmente, a utilização de *valores abstratos* pode parecer vaga ao desenvolvedor, principalmente utilizando *widgets* que possuam diversos tipos de comportamentos de interface. Na biblioteca *UIKit* [Apple, 2007], biblioteca de elementos gráficos de interface para *iOS* [Apple, 2007], a classe *UIButton* é responsável pela implementação dos botões. O *UIButton* é capaz de reconhecer diversos gestos realizados pelo usuário e atrelá-los a ações no sistema. Por exemplo, podemos citar dois tipos de eventos de interface utilizados em botões: *touch down* e *touch up inside*. O evento *touch down* ocorre quando o usuário simplesmente pressiona o botão. A qualquer toque do usuário, o sistema dispara uma ação. Em *touch up inside*, o evento somente é identificado caso o usuário pressiona o botão e o solte com o dedo em cima da região a qual o botão ocupa da interface gráfica. Para casos onde um elemento de interface tenha vários comportamentos para diferentes eventos, é necessário criar um *valor abstrato* para cada um desses eventos. Todavia, é importante que o desenvolvedor somente adicione *valores abstratos* a eventos possíveis de serem usados pelos usuários no sistema, e não para todos os eventos suportados pelo *widget*. Dessa forma é evitada a criação de testes que não estejam no contexto da aplicação. Na seção *Construção da máquina de estados* deste capítulo, explicamos a relação entre *valores abstratos* e máquinas de estados. No capítulo *Ferramenta* apresentamos detalhes de como a ferramenta interpreta a descrição dos fluxos e identifica o *valor abstrato* referente a cada passo. Após o preenchimento do formulário de casos de uso, inicia-se a construção da máquina de estados.

3.2
Construção da máquina de estados

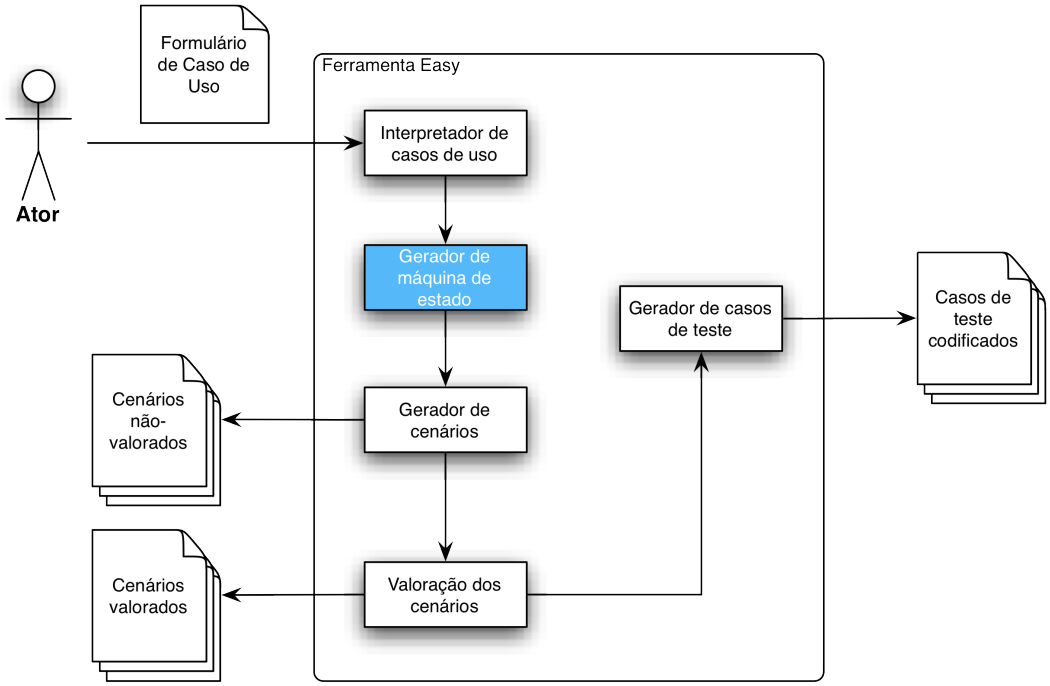


Figura 13: Etapa da ferramenta relacionada a geração de máquina de estados.

A segunda etapa da ferramenta é a geração da máquina de estado. Todo caso de uso possui um fluxo principal de eventos, onde o usuário e o sistema realizam ações. Ao passar o formulário de casos de uso à ferramenta, são identificados os *valores abstratos* relacionado ao fluxo principal. Antes de explicarmos o processos de construção da máquina de estado, é necessário esclarecer a relação dos *valores abstratos* com os fluxos do caso de uso. Anteriormente, comentamos que o fluxo principal é responsável pelas ações naturais ao sistema testado. Cada passo no fluxo representa um ou mais *valores abstratos* que os *widgets* podem assumir sem que o sistema aponte alguma irregularidade. Uma das utilidades do campo *Regra de negócio* é especificar se os *valores abstratos* dos elementos de interface pertencem, ou não, ao fluxo principal. Vamos utilizar como exemplo a tela de autenticação apresentada na Figura 7. Neste novo exemplo, o usuário não cria fluxos alternativos, mas especifica no campo *Regra de negócio* que os três *valores abstratos* da caixa de entrada de texto *Nome* pertencem ao fluxo principal. Isso significa que o sistema autoriza a entrada de usuários com nomes conhecidos, desconhecido e até quando o campo está vazio. As três possibilidades de valores são válidas e permitidas pelo sistema.

A construção da máquina de estado começa com a identificação dos *valores abstratos* respectivos ao fluxo principal. Para cada *valor abstrato* de um elementos de interface, é criado um estado na máquina. Em outras palavras, um novo nó no grafo é criado. O relacionamento entre os estados da máquina é definido pela cronologia dos passos descritos no fluxo principal. Por exemplo, se um passo A ocorre logo antes de um passo B, então o estado respectivo a A apontará para o estado responsável por B. Dessa forma, a máquina de estado é montada inicialmente em função do fluxo principal. Podemos dizer que o fluxo principal é a *espinha dorsal* do grafo. Na Figura 14 ilustramos a construção da máquina de estados do fluxo principal da tela de autenticação apresentado na Figura 8. Neste exemplo, cada elemento de interface possui somente um *valor abstrato* no fluxo principal. Nos passos 1 e 2, os *widgetsNome* e *Senha* utilizam o valor *conhecido*. No passo 3, o botão *Entrar* possui valor *clicado*. Por último, no passo 4 a legenda *Resposta do Sistema* exibe o valor concreto *"Autenticação com sucesso"*.

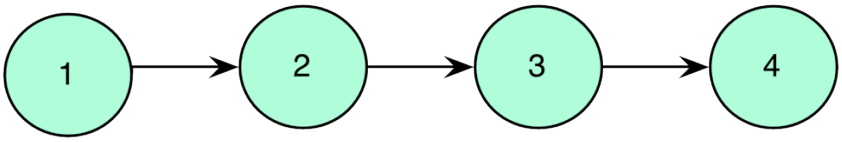


Figura 14: Máquina de estados com o fluxo principal.

Em seguida, a ferramenta observa os fluxos alternativos passados pelo usuário. Esses fluxos são eventos que, por alguma peculiaridade, geram comportamentos no sistema diferentes do descrito no fluxo principal. Todo fluxo alternativo deve especificar a qual passo do fluxo principal este é uma alternativa, qual a descrição do passo alternativo e a resposta do sistema para este evento. Sendo assim, a ferramenta adiciona dois novos estados à máquina de estados: um para o passo alternativo e outro para sua resposta do sistema. Durante a adição de um estado alternativo na máquina, são copiados todos os relacionamentos entre estados do nó do grafo para o qual o novo estado diz ser uma alternativa. Dessa forma, conseguimos criar a possibilidade da máquina ter diferentes caminhos a serem percorridos, permitindo a geração de diferentes cenários do caso de uso. Na Figura 15 ilustramos a máquina de estados da Figura 14 com quatro fluxos alternativos. Esses fluxos alternativos são em relação as caixas de entrada de texto *Nome* e *Senha* do fluxo principal da Figura 8. Os estados A1 e A2 são responsáveis pelos *valores abstratosdesconhecido* e *vazio* do elemento de

interface *Nome*, respectivamente. Os estados A3 e A4 pelos *valores abstratos desconhecido* e *vazio* de *Senha*, respectivamente. Estados que possuem a letra R em seu identificador, como o A1R, são as respostas do sistema respectivas aos estados alternativos com mesma numeração.

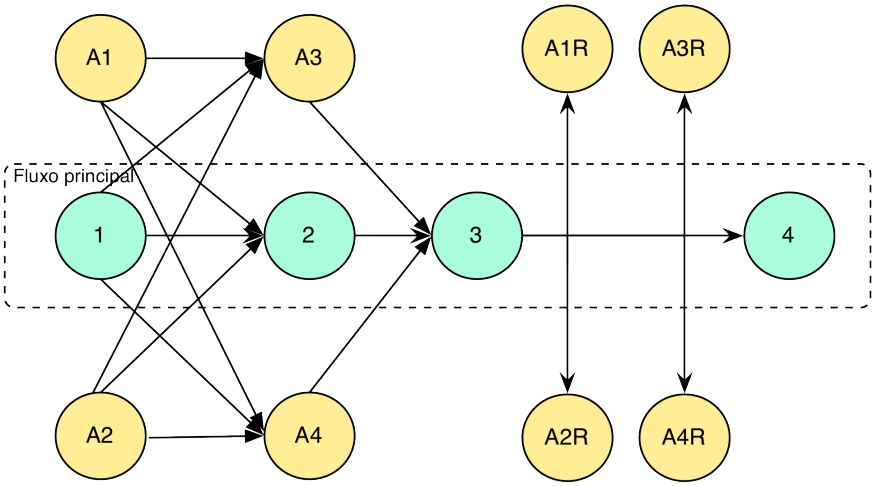


Figura 15: Máquina de estados com osfluxos alternativos.

Ao observarmos a máquina de estado representada na Figura 15, podemos observar a falta de um estado para o botão *Entrar*. O *valor abstrato não clicado* não foi adicionado à máquina devido ao desenvolvedor não ter criado um fluxo alternativo para esta possibilidade. Nem sempre o desenvolvedor dispõe de tempo para realizar um roteiro de teste com completeza, ou simplesmente não possui conhecimento sobre possíveis cenários que possam apresentar erros ao seu sistema. De qualquer forma, a produção de uma grande massa de teste é custosa. Devido a isso, a ferramenta *Easy* oferece aos desenvolvedores a opção de sugerir cenários não especificados no formulário de casos de uso. Apelidado de *fluxo sugerido*, são todos os caminhos possíveis de serem percorridos na máquina de estados que não foram mencionados no fluxo principal e alternativos. Acreditamos que esta funcionalidade possa apresentar cenários desconhecidos ao desenvolvedor, aumentando a completude do roteiro de testes.

Após a montagem da máquina com os fluxos passados no formulário, inicia-se a identificação dos elementos de interface que não tiveram todos os seus *valores abstratos* adicionados à máquina de estados. Para cada um desses valores é criado um estado referente ao fluxo sugerido. Diferente do fluxo principal e alternativo, esses estados não possuem respostas do sistema definidas, pois o usuário não tem conhecimento de sua existência. Assim, o desenvolvedor precisa

passar em um segundo momento os valores reais de resposta do sistema para os cenários do fluxo sugerido. No capítulo *Ferramenta* demonstramos como o usuário passa esses valores à ferramenta. Na Figura 16 apresentamos a máquina de estados com o fluxo sugerido, o estado S1, referente ao valor não clicado do botão *Entrar*.

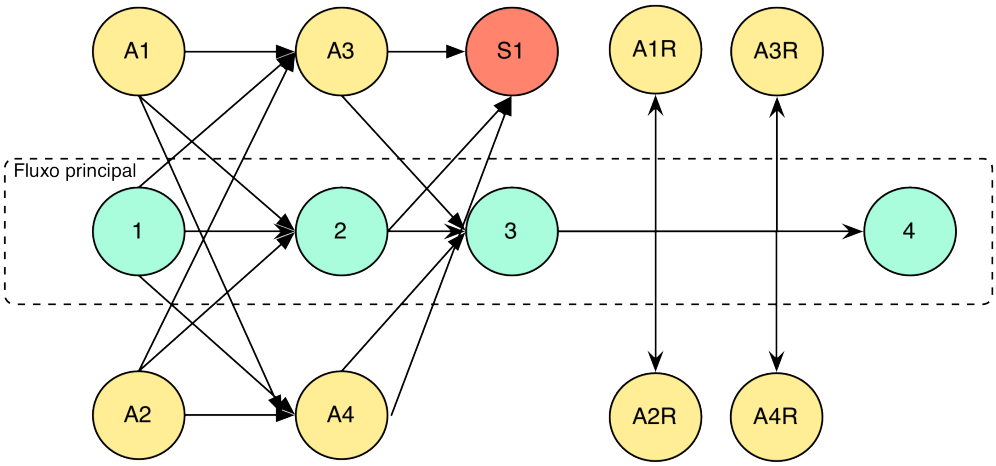


Figura 16: Máquina de estados com o fluxo sugerido

3.3 Geração de cenários

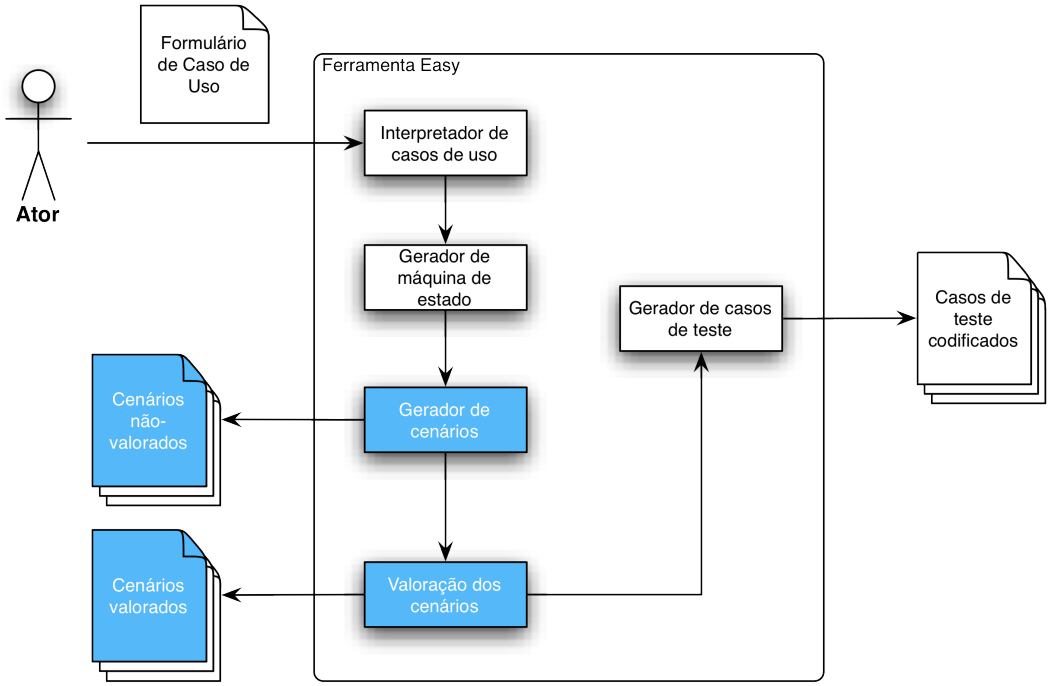


Figura 17: Etapa da ferramenta relacionada à geração de cenários.

A próxima etapa da ferramenta é a geração de cenários. Tendo a construção da máquina de estados finalizada, começamos a percorrê-la para definirmos os cenários. Semelhante à *busca em profundidade*, iniciamos pelos primeiros estados da máquina, ou as *cabeças do grafo*. Em um cenário, cada estado deve apontar somente para um próximo estado. Quando se encontra um estado que não possui referência para um próximo, significa que chegamos ao final da máquina. Neste instante identificamos o caminho percorrido como um cenário. Cada cenário possui somente um estado de cada elemento de interface utilizado no caso de uso. Na Figura 18 ilustramos como ocorre a geração de um cenário.

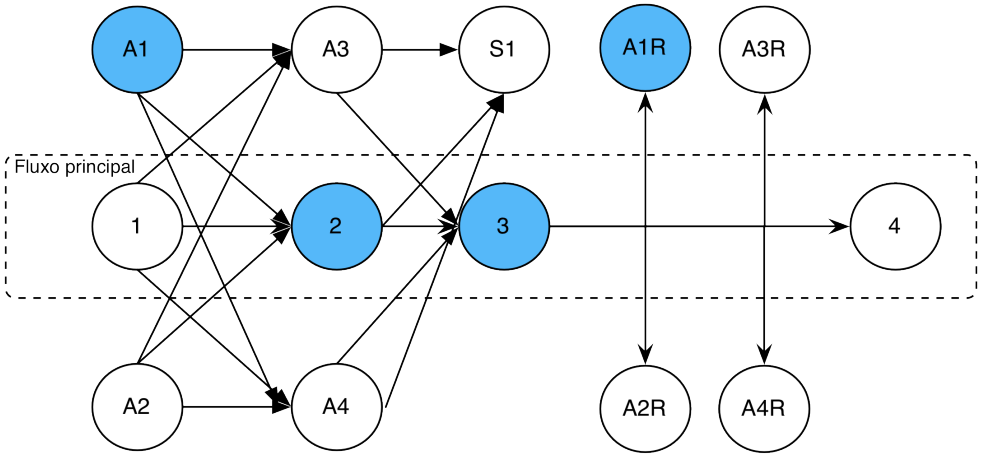


Figura 18: Exemplo de um cenário na máquina de estados.

A geração de cenários ocorre em duas partes: geração de *cenários abstratos* e *cenários concretos*. A diferença entre esses cenários é o tipo de valor que os elementos de interface possuem. Os cenários abstratos possuem somente estados com *valores abstratos*, os mesmos valores responsáveis pela construção da máquina de estados. Inicialmente, a ferramenta gera os cenários abstratos e apresenta ao desenvolvedor. Nesta etapa, o desenvolvedor pode observar os cenários derivados de seu caso de uso e selecionar quais são pertinentes ao seu roteiro de teste. Após escolher os cenários, inicia-se a valoração dos cenários abstratos, ou a criação dos cenários concretos. Os cenários concretos são cenários que possuem valores de entrada específicos, os quais serão utilizados nos casos de testes codificados. Esses valores específicos são gerados aleatoriamente pela ferramenta, porém o desenvolvedor pode editá-los manualmente. Nas Figuras 19 e 20 são apresentados exemplos de cenários abstratos e concretos, respectivamente. Os exemplos abaixo são relacionados à tela de autenticação citada anteriormente.

A1. Usuário preenche o campo Nome com valor desconhecido.

2. Usuário preenche o campo Senha com valor conhecido.

3. Usuário clica no botão Entrar.

A1R. Resposta do Sistema exibe “Usuário com nome desconhecido”.

Figura 19: Exemplo de cenário abstrato.

A1. Usuário preenche o campo Nome com valor “x2tyop4713”.

2. Usuário preenche o campo Senha com valor “senha1234”.

3. Usuário clica no botão Entrar.

A1R. Resposta do Sistema exibe “Usuário com nome desconhecido”.

Figura 20: Exemplo de cenário concreto.

O objetivo de criarmos esses dois tipos de cenários – abstratos e concretos – é facilitar o desenvolvimento de testes. Acreditamos que a lacuna entre a especificação dos casos de teste e a produção de teste codificados é muito grande e poderia ser melhor explorada. Ao permitir ao desenvolvedor acompanhar a geração dos testes, começando por uma forma simples e abstrata, e a cada etapa ficar mais específica e concreta, aumentaríamos a objetividade dos testes, deixando-os mais pertinentes aos sistemas a serem testados. Além disso, a descrição dos cenários pode ser compreendida por usuários sem conhecimento de programação, facilitando a comunicação entre clientes e desenvolvedores ao definirem os casos de testes. No capítulo *Ferramentas* mostramos como o desenvolvedor pode manipular os cenários, editar valores específicos e selecionar quais cenários deverão se transformar em casos de teste codificados.

3.4
Geração de scripts de teste

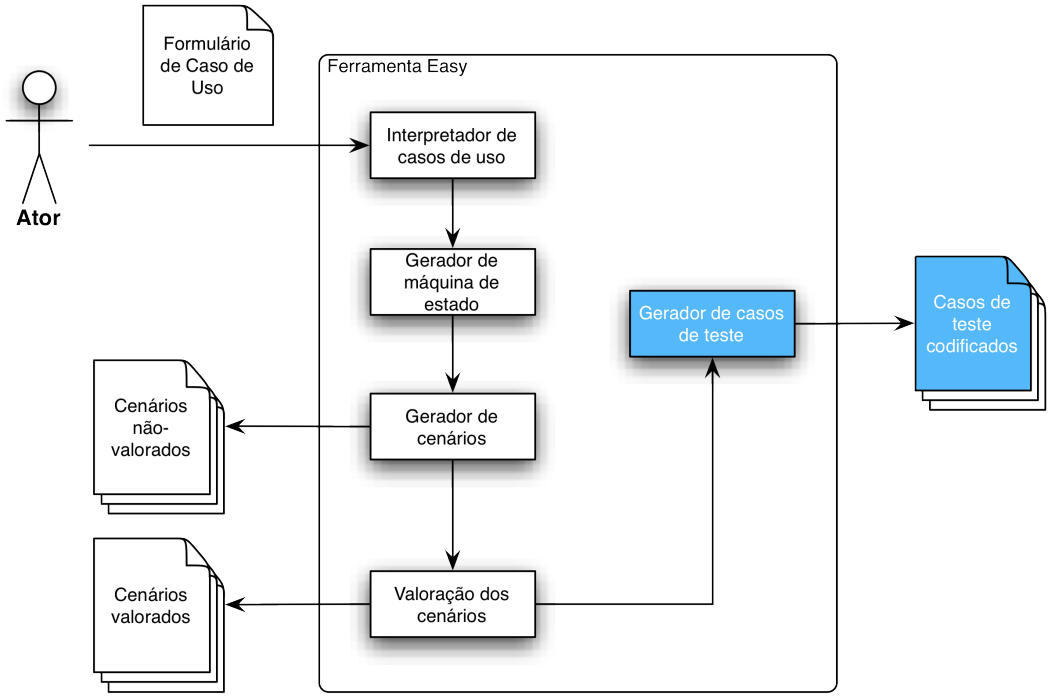


Figura 21: Etapa da ferramenta relacionada à geração de scripts de teste.

A última etapa da ferramenta é a criação dos *scripts* de teste. Na etapa passada, cenários foram gerados e valorados. O desenvolvedor pode selecionar quais cenários deveriam permanecer no processo e editar os valores reais incluídos neles. Após a conclusão dessa etapa, inicia-se a transformação dos cenários em casos de testes. O gerador de casos de teste interpreta as ações ocorridas em cada estado dos cenários e as converte em códigos de programação. Cada cenário irá gerar um arquivo de *script* de teste independente, permitindo ao desenvolvedor poder executar os testes individualmente. Os *scripts* gerados são destinados a testes de interface, sendo esse um dos objetivos iniciais da ferramenta *Easy*.

É importante salientar que a ferramenta é capaz somente de gerar casos de testes codificados, e não executá-los. Assim como comentado no capítulo *Estado da arte*, existem ferramentas que executam testes de interface. A vantagem de utilizar a *Easy* é gerar uma grande massa de teste rapidamente e com pouco esforço, para assim serem executados em ferramentas de teste de interface. A ferramenta *Easy* possui inicialmente suporte para a geração de *scripts* de testes para a ferramenta *UI Automation* [Apple, 2011], responsável por executar testes de interface de aplicações destinadas ao*iOS* [Apple, 2007]. O *iOS* é o sistema

operacional utilizado por *iPhone*, *iPad* e *iPod Touch*. No capítulo *Ferramenta* esclarecemos como é realizada a conversão de cenários em *scripts* de teste e sobre como adicionar o suporte a novas linguagens na ferramenta *Easy*.

3.5
Segmentação de casos de uso

Nesta seção comentamos um recurso da ferramenta *Easy* responsável por reduzir o esforço do desenvolvedor na geração de casos de testes. Durante a modelagem da ferramenta, observamos a possibilidade de casos de uso com diversos elementos de interface gerarem um grande número cenários. Dependendo do número de cenários produzidos, torna-se inviável para o desenvolvedor administrá-los. Vamos utilizar como exemplo um caso de uso que possua doze elementos de interface. Cada elemento tem dois *valores abstratos*. Sabemos que para cada *valor abstrato* de um elemento de interface, será criado um estado na máquina de estados. Dessa forma, cada elemento apresenta duas opções de se percorrer a máquina. Ao calcularmos o número de cenários possíveis a serem gerados, obtemos: $2^{12} = 4.096$. Um número dessa proporção torna extremamente custosa a geração dos testes ao desenvolvedor. Acreditarmos também que muitos desses testes seriam semelhantes e tornariam o processo repetitivo, testando diversas vezes uma mesma situação.

O ponto chave desse problema é a quantidade de elementos de interface em um caso de uso. Em outras palavras, o número de passos que o fluxo principal de um caso de uso deva possuir. Não é possível afirmar a existência de um número específico, pois existem diversos tipos de interface gráfica nos *softwares*. Porém, há como dividir o problema em diversas partes, tratando cada uma por vez. Inspirado na estratégia *dividir para conquistar*, acreditamos que a solução está na criação de uma hierarquia cronológica de casos de uso. Ao invés de criar um caso de uso responsável por uma ação por completo, criaremos diversos casos de uso específicos a cada etapa da ação. Por exemplo, ao efetuar uma compra na internet, é comum a existência das seguintes etapas: escolha do produto, definição do endereço de envio e forma de pagamento. Um único caso de uso seria capaz de descrever as ações do usuário ao sistema na realização de uma compra. Porém, novamente, o número de cenários seria muito grande. Sendo assim, ao invés de

um caso de uso, o desenvolvedor faria três casos, um para cada etapa de compra. Esses casos possuiriam uma ordem cronológica de execução, onde uma etapa somente poderia ser executada se sua antecessora já tivesse sido concluída. No nosso exemplo, o usuário somente poderá especificar as formas de pagamento – última etapa – caso tenha passado com sucesso pelas etapas de escolha de produto e definição o endereço de entrega. Acreditamos que a segmentação é específica a cada caso de uso. Inicialmente, optamos por dividir os casos de uso em relação as janelas envolvidas na ação a ser testada. Para cada janela, existirá um caso de uso respectivo.

Para percebermos a redução drástica no número de cenários com essa técnica, vamos utilizar novamente o exemplo do caso de uso com doze elementos de interface, porém agora iremos segmentá-lo em três casos. Cada um desses casos de uso terá quatro elementos de interface. Ao calcularmos os números de cenários, obtemos: $2^4 + 2^4 + 2^4 = 48$. Comparados aos antigos 4.096 cenários, os atuais 48 cenários representam um número extremamente menor e mais fácil para o desenvolvedor manipular. A redução no número de cenários não compromete o grau de corretude dos roteiros de testes. O processo de segmentação consiste na geração de máquinas de estados completas que se comunicam com arestas de corte. Uma aresta de corte é uma aresta que, se removida, torna o grafo desconexo. Ao certificarmos que uma etapa de um software esteja devidamente testada, podemos afirmar que não há necessidade de inclusão de todos os seus cenários aos testes das etapas seguintes. Os cenários de máquina de estados sucessores somente poderão ser executados por cenários de máquinas antecessoras que possuam arestas de corte. Assim, agilizamos o processo com uma bateria de teste menor e mais objetiva.

Ao preencher o campo *Pré-condição* do formulário de caso de uso, indicamos a dependência cronológica do atual caso com outro já cadastrado na ferramenta. Durante a geração de cenários, a ferramenta adiciona ao início dos cenários do caso de uso corrente um cenário do fluxo principal do caso de uso antecessor. Todos os cenários gerados possuirão o mesmo cenário do antecessor, sendo sempre este do fluxo principal. O motivo de escolhermos um cenário do fluxo principal é devido a este representar o fluxo natural de ações do usuário ao sistema. Assim garantimos que casos de uso dependentes sejam executados, pois ações realizadas nos casos de uso antecessores são esperadas pelo sistema. Na

Figura 22 ilustramos uma máquina de estados de um caso de uso que possui muitos elementos de interface e no qual não foi utilizada a segmentação. Na Figura 23 apresentamos um exemplo de como a segmentação funcionaria ao ser aplicada na máquina de estado da Figura 22. É possível observar na Figura 23 que as duas máquinas de estados se comunicam por arestas de corte.

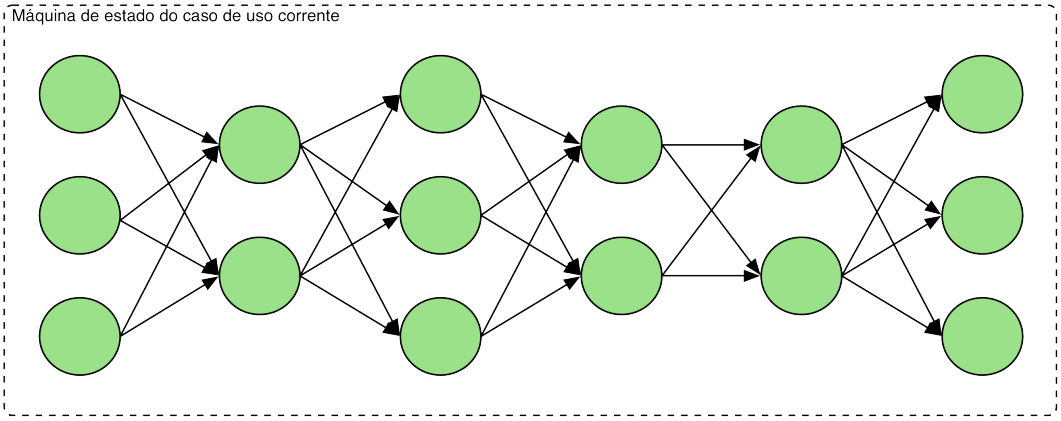


Figura 22: Exemplo de máquina de estados de um caso de uso não segmentado.

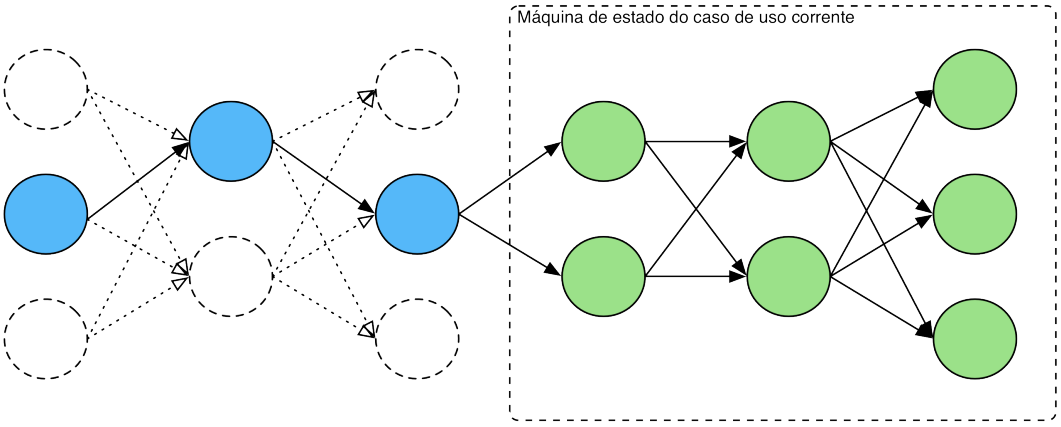


Figura 23: Exemplo de máquina de estados de um caso de uso segmentado.

4
Ferramenta

No capítulo anterior comentamos a lógica por trás do processo de geração de testes na ferramenta *Easy*, como o padrão utilizado na descrição dos casos de uso, a estratégia na montagem da máquina de estados, a geração de *cenários abstratos* e *concretos*, a criação de scripts de testes e a segmentação de casos de uso.

Neste capítulo esclarecemos como as etapas da ferramenta funcionam e como foram implementadas. Inicialmente, apresentamos como é realizada a descrição do caso de uso pela interface gráfica da ferramenta. Em seguida, mostramos como a ferramenta apresenta os cenários criados, permitindo ao desenvolvedor selecioná-los e editar valores de entrada e oráculos. São apresentados também como são gerados os scripts de testes, a implementação da segmentação dos casos de uso e os diagramas de classes da ferramenta.

Uma funcionalidade ainda não abordada é a capacidade do desenvolvedor cadastrar novos elementos de interface na ferramenta *Easy*. Essa funcionalidade permite a geração de testes para *widgets* recém desenvolvidos. Também é possível registrar elementos de interface utilizados por outras linguagens de programação. Dessa forma, a ferramenta mostra-se adaptável aos objetivos do desenvolvedor, independendo de novos elementos de interface ou diferentes linguagens de programação utilizadas em suas aplicações.

O capítulo divide-se em cinco partes: na primeira apresentamos como o desenvolvedor descreve casos de uso na ferramenta; na segunda explicamos a geração de cenários e como a ferramenta apresenta-os ao desenvolvedor; na terceira comentamos detalhes da implementação da segmentação de casos de uso; na quarta esclarecemos como o desenvolvedor cadastra novos elementos de interface; e por último, na quinta parte, apresentamos os diagramas de classes da ferramenta.

4.1
Descrição de casos de uso

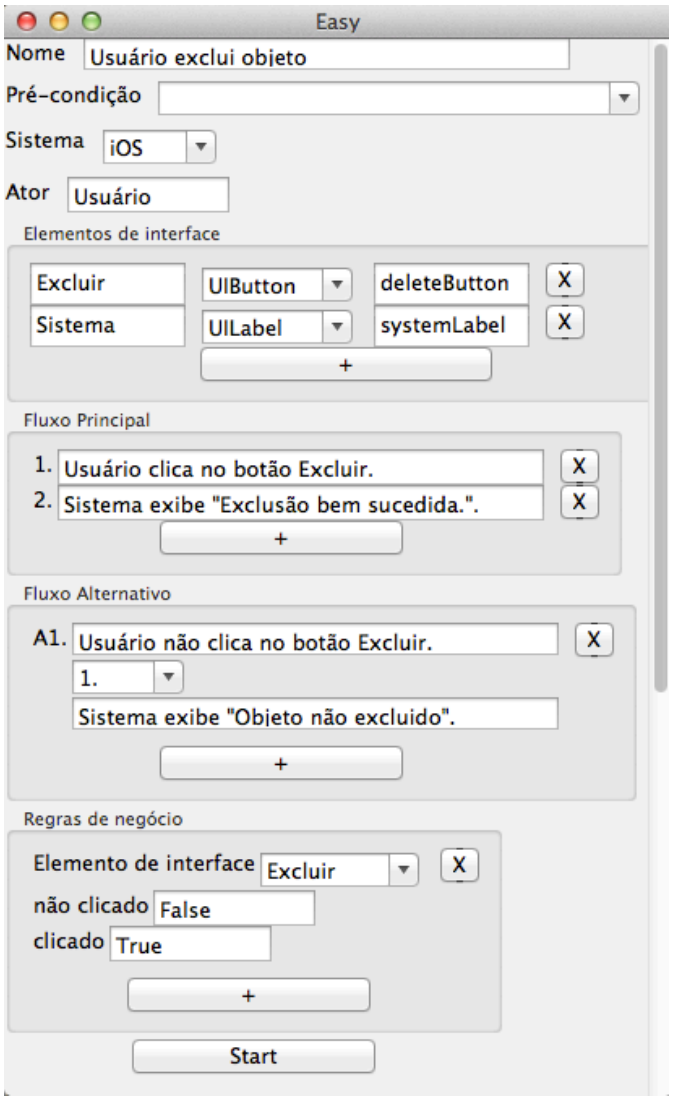


Figura 24: Tela de formulário de casos de uso.

Nesta seção será explicado como o desenvolvedor descreve casos de uso pela interface gráfica da ferramenta. Na Figura 24 podemos observar os campos do formulário de caso de uso. O campo *Nome* é onde o desenvolvedor especifica o nome do caso de uso corrente. O campo *Sistema* é responsável por identificar o tipo do sistema aos quais os casos de teste são destinados. Em outras palavras, o desenvolvedor está informando à ferramenta a linguagem em que os casos de testes deverão ser gerados. A ferramenta possui atualmente suporte a casos de teste para aplicações destinadas a *iOS*, sistema operacional utilizado por dispositivos móveis como *iPhone*, *iPad* e *iPod Touch*. Para este caso, o desenvolvedor deverá selecionar a opção *iOS* no campo *Sistema*. O próximo

campo é o *Pré-condição*, responsável por definir o nome de um caso de uso que seja pré-requisito à execução do caso de uso corrente. Ao utilizar esse campo, a ferramenta aciona o processo de segmentação de casos de uso, comentado no capítulo anterior. Como nem todos casos de uso possuem uma pré-condição, o preenchimento deste campo não é obrigatório. Em seguida temos o campo *Ator*, onde o desenvolvedor especifica o substantivo que representa o ator do caso de uso. A ferramenta sugere a utilização da palavra *Usuário*, porém o desenvolvedor poderá escolher outra a seu critério.

A próxima etapa do formulário é onde se descreve os elementos de interface a serem utilizados no caso de uso – o campo *Elementos de interface*. A Figura 25 apresenta o campo *Elementos de interface* sendo preenchido com os elementos da janela de autenticação da Figura 7, exemplo utilizado no capítulo *Processo*. Nesse exemplo os campos *Nome* e *Senha* são as caixas de entrada de texto, *Entrar* é o botão e *Sistema* é a legenda que informa a situação do sistema ao usuário.

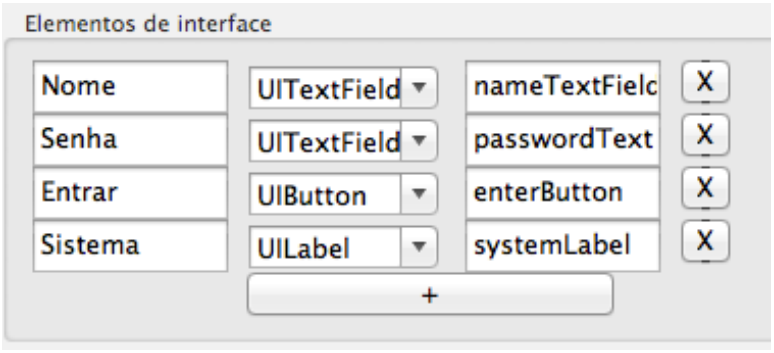


Figura 25: Exemplo do campo Elementos de interface.

O campo *Elementos de interface* é dividido em linhas, onde cada uma representa um elemento de interface independente. Para cadastrar um novo elemento de interface é necessário preencher três campos: um nome respectivo ao elemento de interface no caso de uso, a classe do *widget* utilizado pelo elemento e o nome do elemento na aplicação a ser testada. Assim como o campo *Ator*, o desenvolvedor poderá escolher um nome para identificar o elemento de interface no caso de uso. É importante comentar que o desenvolvedor deve escolher um nome que facilmente identifique o respectivo elemento de interface, pois esse nome é utilizado nos demais campos do formulário – como *Fluxo principal*, *Fluxo alternativo* e *Regra de negócio* – e nos cenários a serem criados. O segundo campo é onde se especifica a classe da biblioteca de interface gráfica utilizada pelo elemento de interface. Ao selecionar o tipo de sistema no campo *Sistema*, a

ferramenta apresenta neste campo todas as classes de *widgets* possíveis de serem usados com o sistema escolhido. Esse campo é responsável por informar a ferramenta como deve-se tratar o elemento de interface em questão, como por exemplo o padrão linguístico a ser adotado em sua descrição no fluxo principal, seus *valores abstratos* e como deve ser criado os *scripts* de testes. As classes selecionadas na Figura 25 são da biblioteca *UIKit* [Apple, 2007], biblioteca de elementos gráficos de interface para *iOS*. O terceiro campo é o nome do elemento de interface na aplicação a ser testada. Para que os *scripts* de testes realizem testes de interface gráfica, é necessário ter conhecimento de como esses elementos de interface são identificados em suas aplicações. Assim é possível encontrá-los e simular comportamentos de interface do usuário direcionados a esses elementos.

Em seguida temos o campo *Fluxo principal*, onde é descrito o fluxo natural de ações do usuário e sistema. Na Figura 26 apresentamos o campo com o fluxo principal da tela de autenticação da Figura 7. Podemos encontrar a mesma descrição do fluxo na Figura 8 no capítulo *Processo*.

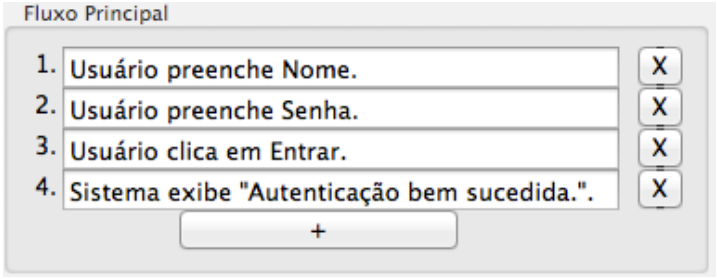


Figura 26: Exemplo do campo Fluxo principal.

O campo *Fluxo principal* é divido em linhas que representam os passos do fluxo natural de eventos no sistema. Cada linha possui dois campos: um número e um campo para a descrição do passo. O número indica a posição cronológica do passo no fluxo e serve de referência para os fluxos alternativos, onde o usuário precisa indicar o número de um passo do fluxo principal. O segundo campo da linha é o de descrição do passo. Cada passo deve tratar exclusivamente de uma ação no sistema, podendo esta ser realizada pelo usuário ou pelo próprio sistema. A descrição do passo deve seguir o padrão apresentado na seção *Descrição de casos de uso* do capítulo *Processo*. É importante enfatizar que o fluxo principal deve possuir somente um único passo em que o ator da ação é o sistema. É obrigatório que este passo seja o último do fluxo. Todos os passos restantes devem descrever ações que o usuário realiza na aplicação a ser testada. Na Figura

26 podemos observar que somente o último passo do fluxo – com o ator *Sistema* – é destinado a uma ação executada pelo sistema; os demais são realizados pelo usuário.

O próximo campo do formulário é o *Fluxo alternativo*. Seu objetivo é registrar eventualidades que possam ocorrer em relação ao fluxo natural de ações do caso de uso. Em outras palavras, fluxos que sejam diferentes do fluxo principal. Na Figura 27 mostramos um exemplo do campo *Fluxo alternativo* sendo preenchido com um fluxo alternativo em relação ao elemento de interface *Nome*.

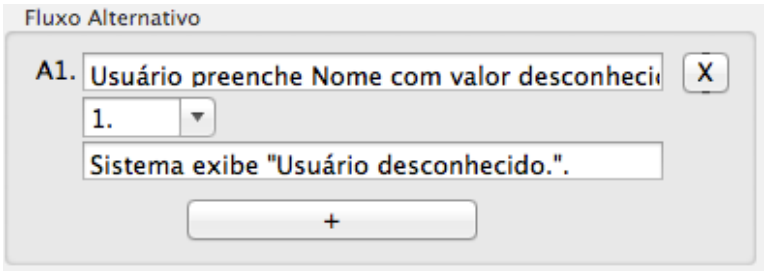


Figura 27: Exemplo do campo Fluxo alternativo.

Ao iniciar o cadastro de um fluxo alternativo, o desenvolvedor precisa preencher três campos: a descrição do passo alternativo, o número do passo do fluxo principal que este afirma ser uma alternativa e a respectiva resposta do sistema ao passo alternativo. A descrição do passo alternativo serve para especificar o ponto de divergência com um dos passos do fluxo principal. Essa descrição precisa seguir o padrão apresentado na Figura 11 do capítulo *Processo*. O segundo campo é o número do passo do fluxo principal para o qual este afirma ser uma alternativa. O terceiro campo é a resposta do sistema quando o fluxo alternativo é executado. Como comentado no capítulo *Processo*, a resposta do sistema serve de oráculo nos *scripts* de teste. Como o fluxo alternativo é uma possibilidade diferente do fluxo principal, é necessário um retorno do sistema específico para essa alternativa. O preenchimento do campo *Fluxo alternativo* não é obrigatório, permitindo assim ao desenvolvedor gerar casos de testes informando somente o fluxo principal.

Por fim, o último campo do formulário é o *Regras de negócio*. O objetivo desse campo é permitir ao desenvolvedor passar dados específicos aos elementos de interface utilizados no caso de uso. Na Figura 28 são apresentadas as regras de

negócio dos elementos de interface cadastrados no campo *Elementos de interface* da Figura 25.

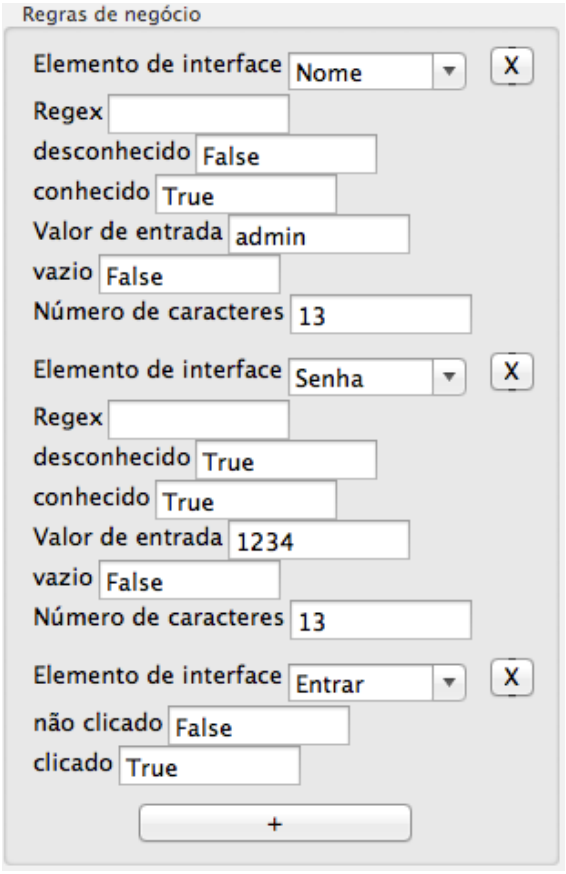


Figura 28: Exemplo do campo Regras de negócio.

Ao observarmos a Figura 28, podemos perceber que as regras de negócio cadastradas possuem diferentes campos. Na verdade, os campos variam em relação ao tipo – classe – do elemento de interface. Uma regra de negócio é dividida em três partes: o nome do elemento de interface ao qual a regra é destinada, campos relacionados aos *valores abstratos* do elemento e campos adicionais exclusivos do tipo de *widget* selecionado. Inicialmente, quando o usuário pretende cadastrar uma regra de negócio, o único campo presente é o de seleção de nome de elementos de interface. Os nomes disponíveis são dos elementos cadastrados no campo *Elementos de interface* do formulário. Na Figura 29 é ilustrado o início do preenchimento de uma regra de negócio, onde o usuário ainda não selecionou um elemento de interface.

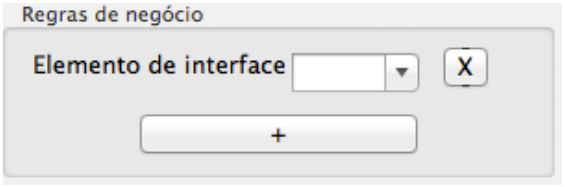


Figura 29: Exemplo do campo Regras de negócio sem um elemento de interface.

Ao selecionar um elemento, a ferramenta adiciona campos respectivos ao tipo de *widget* do elemento de interface escolhido. Esses campos são as outras duas partes ainda não comentadas de uma regra de negócio: *valores abstratos* e campos exclusivos do *widget*. Os campos relacionados aos *valores abstratos* têm o objetivo permitir que o desenvolvedor indique quais valores devem pertencer ao fluxo principal. Anteriormente, comentamos que o fluxo principal possui somente *valores abstratos* que pertencem ao fluxo natural de eventos na aplicação testada. Ao cadastrar um fluxo alternativo, estamos informando que um determinado *valor abstrato* de um elemento de interface não pertence ao fluxo principal. Dessa forma, o desenvolvedor pode especificar em uma regra de negócio quais *valores abstratos* devem participar do fluxo principal. Os campos dos *valores abstratos* somente podem receber dois tipos de valores: *True* ou *False*. Ao passar o valor *True* indicamos que o *valor abstrato* pertence ao fluxo principal. O oposto acontece caso seja passado o valor *False*. Para esclarecermos melhor essa funcionalidade, vamos utilizar o exemplo apresentado na Figura 28. No exemplo temos uma regra de negócio para o elemento de interface *Nome*. Podemos observar que os campos *conhecido*, *desconhecido* e *vazio* possuem valores *True*, *False* e *False*, respectivamente. Essa configuração indica que somente o *valor abstrato conhecido* pertence ao fluxo principal. Os demais estão no fluxo alternativo ou sugerido.

A vantagem de utilizar esse recurso é permitir que mais de um *valor abstrato* pertença ao fluxo principal. Por exemplo, em uma aplicação do tipo *despertador*, o usuário pode criar alarmes para acordar. O processo de criação de um alarme envolve o preenchimento de alguns campos, como nome, horário de despertar, etc. Nesse exemplo iremos focar no campo responsável pelo nome do alarme. Essa aplicação não possui nenhuma restrição sobre o nome do alarme, permitindo assim a entrada de qualquer valor para a criação de um alarme. No caso do usuário não passar nenhum nome, deixando o campo de nome vazio, a aplicação preenche-o com um valor padrão – por exemplo *Alarme*. Sendo assim,

ao descrever o caso de uso *Criar Alarme* na ferramenta *Easy*, o desenvolvedor poderá criar uma regra de negócio respectiva ao elemento de interface de nome do alarme e preencher os campos relacionados aos *valores abstratos* do elemento – *conhecido*, *desconhecido* e *vazio* – com o valor *True*. Dessa forma, garantimos que todos os *valores abstratos* pertençam ao fluxo principal, pois a aplicação aceita qualquer tipo de valor como nome de um alarme. Importante comentar que é obrigatório ao menos um *valor abstrato* estar cadastrado no fluxo principal. No mínimo um campo deverá receber o valor *True*.

A última parte das regras de negócio são os campos exclusivos dos *widgets*. Devido aos *widgets* possuírem diferentes funções nas interfaces gráficas, é necessário em alguns momentos passar parâmetros específicos do elemento de interface. Por exemplo, a Figura 28 contém uma regra de negócio do elemento de interface *Nome* sendo cadastrada. Dos campos dessa regra, vamos observar três deles: *Regex*, *Valor de entrada* e *Número de caracteres*. O campo *Regex* permite ao desenvolvedor passar uma expressão regular, caso o elemento de interface tenha alguma restrição ao formato de seus valores de entrada. O campo *Valor de entrada* serve quando a caixa de entrada de texto precisa receber um valor de entrada específico para o funcionamento correto da aplicação, como por exemplo *softwares* que possuem autenticação de usuários. Na Figura 28 pode-se observar que o desenvolvedor passou o valor *admin* como valor de entrada para o elemento de interface *Nome*. Importante informar que o valor de entrada passado neste campo será utilizado em cenários relacionados ao fluxo principal que utilizem o *valor abstrato conhecido* de caixas de entrada de texto. O campo *Número de caracteres* tem o objetivo de especificar quantos caracteres os valores de entrada do respectivo elemento de interface devem possuir. Ao informar esse dado, a ferramenta é capaz de gerar cenários com valores de entrada aleatórios no tamanho especificado.

Os três campos recém explicados são exclusivos de *widgets* caixa de entrada de texto. Na Figura 28 podemos observar que a regra de negócio do elemento de interface *Senha* possui também esses campos, porém a regra do elemento *Entrar* não. Isso ocorre devido ao elemento *Senha* ser uma caixa de entrada de texto – semelhante à *Nome* – e *Entrar* ser um botão. De todos os *widgets* registrados na ferramenta *Easy*, somente as caixas de entrada de texto possuem elementos exclusivos nas regras de negócio. Caso haja a necessidade de inclusão de novos

campos exclusivos a certos *widgets*, o desenvolvedor poderá cadastrá-los na ferramenta editando a classe responsável pelo elemento de interface. Neste capítulo – *Ferramenta* – esclarecemos como o desenvolvedor edita e adiciona novos *widgets* na ferramenta *Easy* na seção *Cadastro de novos elementos de interface*. O preenchimento do campo *Regras de negócio* não é obrigatório, podendo o desenvolvedor gerar casos de testes sem utilizá-lo. Após o preenchimento do formulário, o desenvolvedor deve apertar o botão *Start* para iniciar a geração de cenários.

4.2
Geração de cenários

Nesta seção será explicado como a ferramenta apresenta os cenários de um caso de uso ao desenvolvedor. Antes de apresentarmos detalhes da interface gráfica dessa etapa, precisamos comentar o motivo de sua existência. Os casos de uso têm a capacidade de gerar grandes números de cenários. Quanto maior o número, maior o esforço do desenvolvedor para administrá-los. No capítulo *Estado da arte* apresentamos diversas estratégias direcionadas à geração automática de testes. Um dos objetivos principais dessas abordagens é a redução do custo de criação de testes aos desenvolvedores. Todavia, nenhuma das referências encontradas aponta a interface gráfica das ferramentas de testes como um dos fatores responsáveis pela diminuição do esforço dos desenvolvedores. Por esse motivo, optamos por avaliar a relevância de uma interface gráfica ao desenvolvimento de casos de testes. Uma das etapas do desenvolvimento da ferramenta *Easy* foi a elaboração de uma interface gráfica que pudesse atender as necessidades do usuário e facilitar a geração de testes.

A interface gráfica divide-se em três etapas: formulário de caso de uso, visualização de *cenários abstratos* e *cenários concretos*. A interface do formulário de casos de uso foi apresentada na seção anterior, onde explicamos como o desenvolvedor deve utilizá-la. Ao passar o caso de uso à ferramenta, inicia-se o processo de geração de cenários. Esses cenários são expostos em duas etapas cronológicas. A primeira apresenta *cenários abstratos* ao desenvolvedor. A segunda os *cenários concretos*. Anteriormente, comentamos a existência desses dois tipos de cenários: *abstratos* e *concretos*. Esses cenários servem para

descrever com linguagem natural restrita as derivações dos casos de usos. Os *cenários abstratos* utilizam os *valores abstratos* dos elementos de interface em suas descrições. Os *cenário concretos* são mais específicos do que *cenários abstratos*, apresentando valores de entrada reais na descrição dos cenários. O intuito dessas duas etapas na interface gráfica da ferramenta é permitir ao desenvolvedor acompanhar a geração dos casos de testes de início ao fim. Ao apresentarmos inicialmente cenários, de forma abstrata e com linguagem natural restrita, e gradativamente incluirmos informações específicas até chegarmos aos *scripts* de testes, permitimos que o desenvolvedor tenha conhecimento do perfil dos casos de testes antes da geração dos próprios testes. Esse processo permite que o desenvolvedor tenha mais controle do conteúdo da massa de teste criada, sem comprometer a agilidade e eficiência do processo de geração automática de testes. Durante a modelagem da ferramenta *Easy* percebemos que as ferramentas de geração automática de casos de testes possuem uma grande lacuna entre a descrição dos casos até a geração dos *scripts* de testes. Essa lacuna poderia ser preenchida por meio de uma interface gráfica, assim como é realizado na ferramenta *Easy* com os *cenários abstratos* e *concretos*.

Após esclarecermos alguns pontos importantes, vamos explicar como a ferramenta apresenta os cenários ao desenvolvedor. Inicialmente, a ferramenta apresenta os *cenários abstratos* do caso de uso passado. O objetivo dessa etapa é mostrar todos os cenários possíveis de serem gerados sem abordar detalhes de valores de entrada reais. O desenvolvedor deve somente preocupar-se em ter conhecimento dos cenários possíveis e selecionar quais devem continuar no processo de geração de testes. Na Figura 30 apresentamos a interface gráfica da etapa de exposição dos *cenários abstratos*, onde os cenários presentes são derivados do caso de uso descrito na Figura 24, na seção *Descrição de casos de uso* deste capítulo.

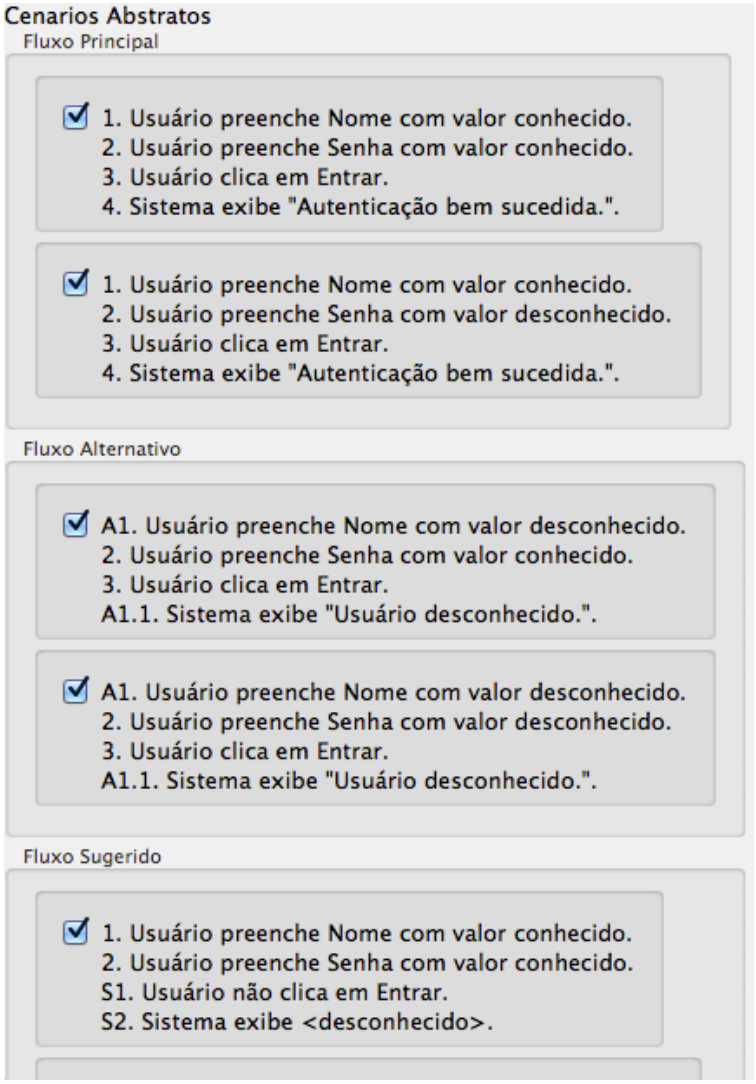


Figura 30: Janela de apresentação dos cenários abstratos.

Ao observarmos a Figura 30, podemos identificar que a janela organiza os cenários em três partes: cenários do fluxo principal, do fluxo alternativo e do fluxo sugerido. Os cenários do fluxo principal correspondem às possibilidades de se percorrer a máquina de estado, segundo o fluxo principal descrito no caso de uso. Do ponto de vista da ferramenta, são os caminhos da máquina de estado que possuem *valores abstratos* cadastrados no fluxo principal. Podemos observar que todos os cenários contidos nesta região da interface possuem a mesma resposta do sistema. Os cenários do fluxo alternativo são os caminhos possíveis de serem percorridos com as alternativas ao fluxo principal cadastradas pelo desenvolvedor no caso de uso. Todo cenário do fluxo alternativo possui um passo alternativo e um retorno do sistema respectivo a esse passo. Os demais passos são semelhantes aos cenários do fluxo principal. Os cenários do fluxo sugerido são todos os cenários possíveis de serem gerados pela máquina de estado, porém que não se

encontram no fluxo principal e fluxos alternativos. Uma característica desses cenários é não possuírem uma resposta do sistema definida. Isso ocorre devido a esses cenários serem possibilidades ainda não conhecidas pelo desenvolvedor. Sendo assim, a ferramenta *Easy* não é capaz de prever comportamento da aplicação sem que o desenvolvedor informe. Pode-se observar na Figura 30 que os cenários do fluxo sugerido possuem o valor <*desconhecido*> no passo de resposta do sistema. Novamente, o objetivo dessa etapa da interface não é especificar valores de entrada reais e retornos do sistema, mas sim permitir que o desenvolvedor tenha conhecimento dos possíveis cenários e selecione os mais relevantes.

A seleção de cenários é uma funcionalidade que permite ao desenvolvedor escolher quais cenários devem continuar no processo de geração de testes. Todos os cenários apresentados na interface gráfica possuem um *checkbox*. Ao selecionar um cenário, o desenvolvedor está informando a ferramenta que ele deve permanecer no processo. Caso contrário, a ferramenta remove o cenário do roteiro de testes. Por padrão, todos os cenários estão inicialmente selecionados. Durante o desenvolvimento da ferramenta *Easy*, percebemos que nem sempre o desenvolvedor precisa utilizar todos os cenários gerados, principalmente os pertencentes ao fluxo sugerido. Assim, o desenvolvedor pode facilmente desconsiderar cenários pouco pertinentes, focando exclusivamente nos mais importantes. Entretanto, é aconselhável utilizar sempre o maior número de cenários possíveis em prol de garantir a completude da massa de testes. Após selecionar os cenários relevantes, o desenvolvedor deve clicarno botão *Gerar Cenários Concretos* para dar continuidade ao processo de geração de casos de testes.

A próxima etapa da interface gráfica é a apresentação dos *cenários concretos*. Após o desenvolvedor selecionar os *cenários abstratos* pertinentes, inicia-se o processo de valoração destes, transformando-os em *cenários concretos*. Na Figura 31 é apresentado a janela de *cenários concretos*, onde os cenários presentes são os selecionados na Figura 30.

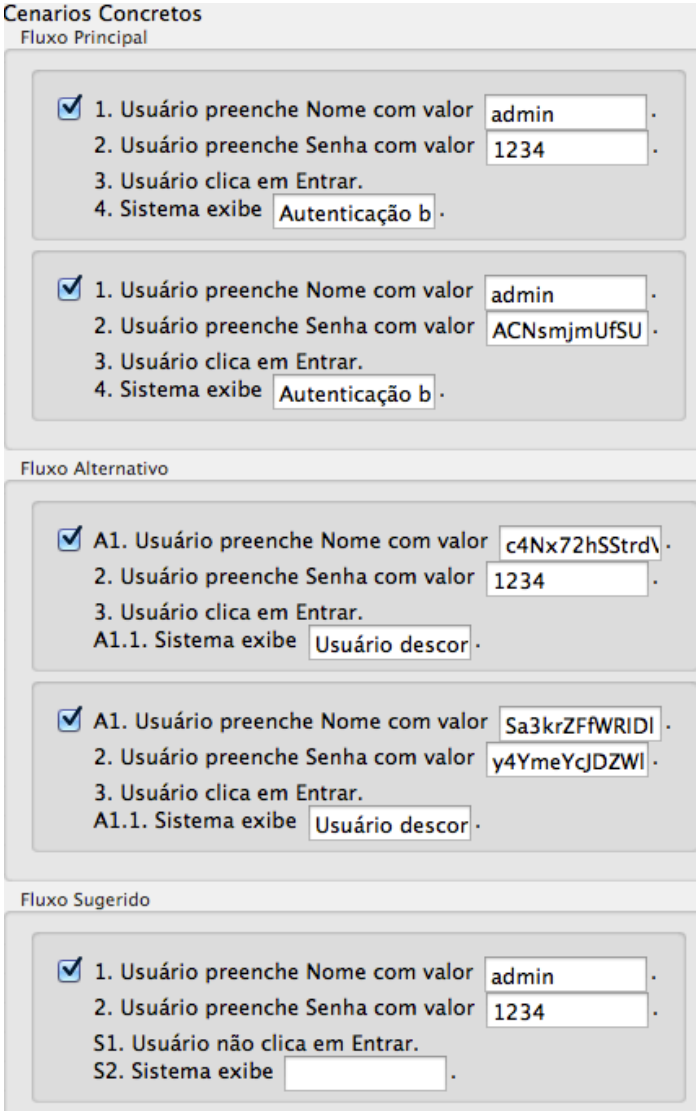


Figura 31: Janela de apresentação dos cenários concretos.

O objetivo dessa etapa da interface é apresentar mais detalhes sobre os cenários do que na janela de *cenários abstratos*. Os *cenários abstratos* selecionados na etapa anterior passam por um processo de valoração, onde são adicionados a eles valores reais de entrada, gerados automaticamente pela ferramenta, que serão utilizados nos *scripts de testes*. Após a valoração, a ferramenta apresenta esses cenários valorados – *cenários concretos* – ao desenvolvedor, para que este tenha conhecimento do perfil dos testes a serem criados. A janela de *cenários concretos* também permite que o desenvolvedor edite os valores de entrada e as respostas do sistema. As respostas do sistema dos cenários do fluxo sugerido devem ser preenchidos nessa etapa. Na Figura 31 podemos observar que todos os cenários possuem campos para a entrada de valores reais. Após certificar-se de que os cenários estão devidamente

preenchidos, o desenvolvedor deve clicarno botão *Concluir* para iniciar a geração dos *scripts* de testes e concluir assim, o processo de produção de testes da ferramenta *Easy*.

A interface gráfica ainda permite que o desenvolvedor possa transitar entre as três etapas – formulário de caso de uso, *cenários abstratos* e *cenários concretos* – durante o processo de geração de testes. Na parte inferior das janelas de *cenários abstratos* e *cenários concretos* há os botões *Voltar Formulário* e *Voltar Cenários Abstratos*, respectivamente, possibilitando que o desenvolvedor possa voltar para a etapa antecessora à corrente. Essa funcionalidade permite que o desenvolvedor possa corrigir erros e editar elementos que não estejam de acordo, sem a necessidade de reiniciar o processo de geração dos testes. Na Figura 32 é ilustrado o grau de liberdade do desenvolvedor entre as etapas da ferramenta.

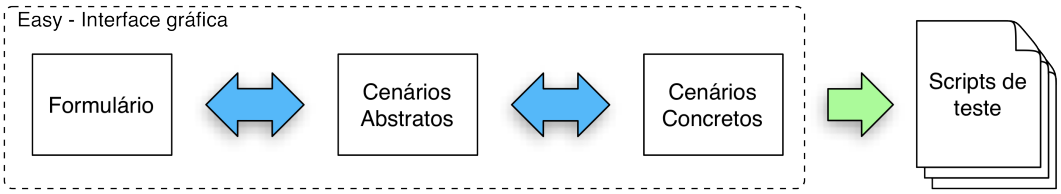


Figura 32: Ilustração da transição entre etapas da ferramenta pelo usuário.

4.3 Segmentação de casos de uso

Nessa seção abordamos a implementação da segmentação de casos de uso pela ferramenta. No capítulo *Processo* explicamos que a segmentação é um processo que conecta máquinas de estados respectivas a casos de uso por arestas de corte. Ao preenchermos o campo *Pré-condição* do formulário de caso de uso, informamos que um caso de uso possui outro como antecessor. Ao finalizar o processo de geração de teste pela ferramenta, sabemos que os cenários gerados são transformados em *scripts* de testes. No caso desses cenários serem de um caso de uso que possua um antecessor, estes devem ser somente executados após a execução um cenário do fluxo principal do caso de uso antecessor. Ao executarmos um cenário do fluxo principal, garantimos que cenários seguintes conseguirão ser executados, pois o fluxo principal é responsável pelo fluxo natural de eventos esperados pelo sistema.

Para realizarmos a segmentação dos casos de uso, a ferramenta possui duas etapas. A primeira é armazenar em uma base de dados todas as informações de um caso de uso passado à ferramenta. Esse processo ocorre, quando concluída a geração de testes de um caso de uso. Uma das informações armazenadas na base de dados é um *script* de teste do fluxo principal, que é fundamental para a execução da segunda etapa do processo. A segunda etapa somente é executada para casos de uso que tenham antecessores – campo *Pré-condição* do formulário foi preenchido. Ao gerar os *scripts* de teste, a ferramenta acessa a base de dados e busca pelo caso de uso passado no campo *Pré-condição*. Ao encontrá-la, é copiado o *script* do fluxo principal, salvo na primeira etapa do processo de segmentação. Com o *script* do caso de uso antecessor em mãos, a ferramenta adiciona-o no início de todos os *scripts* a serem gerados para o caso de uso corrente. Assim garantimos que todos*scripts* de teste serão executados, pois dentro de si está contido o *script* do fluxo principal de seu caso de uso antecessor. Na Figura 33 fazemos um paralelo entre cenários e um *scripts* de teste. Na figura possuímos três elementos: um cenário do fluxo principal do caso de uso A, um cenário de qualquer fluxo do caso de uso B e o *script* de teste respectivo ao cenário do caso de uso B. O caso de uso B possui A como seu antecessor.

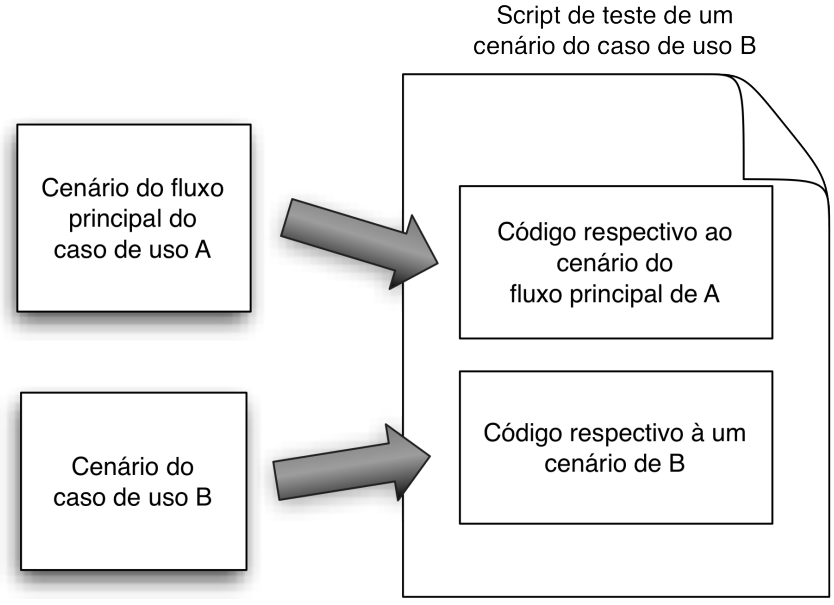


Figura 33: Ilustração da segmentação de casos de uso nos scripts de testes.

4.4
Cadastro de novos elementos de interface

Nesta seção explicamos como o desenvolvedor registra novos elementos de interface na ferramenta *Easy*. Um dos objetivos iniciais da modelagem da ferramenta era permitir ao desenvolvedor utilizar a geração automática de testes para diferentes aplicações e linguagens de programação. Para tornar isso possível, a arquitetura da ferramenta necessita ser abstrata do ponto de vista dos elementos de interface cadastrados, não estando presa a uma linguagem específica. É importante enfatizar que é extremamente recomendado todo usuário da ferramenta *Easy* ter conhecimento de seu funcionamento interno. Durante a utilização da ferramenta na geração de casos de testes para aplicações reais, observamos que em certos momentos foi necessário o cadastro de elementos de interface específicos a algumas aplicações. Tendo o desenvolvedor esse conhecimento em mãos, é possível contornar certas peculiaridades problemáticas das interfaces gráficas das aplicações e ser capaz te continuar a produção de casos de teste.

Para o desenvolvedor incluir novos elementos de interface, é necessário que tenha conhecimento de algumas classes da ferramenta e como estas são utilizadas durante o processo de geração de testes. A primeira classe a ser explicada é a *System*. Na Figura 34 apresentamos seu diagrama de classe.

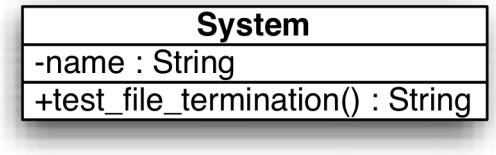


Figura 34: Diagrama de classe de System.

O papel da classe *System* é permitir o cadastro de linguagens ou sistemas de que a ferramenta *Easy* terá suporte. O primeiro passo é a criação de uma classe respectiva a linguagem ou sistema em questão, tendo a classe *System* como superclasse. Ao criá-la, é necessário passar o nome da linguagem ou sistema que esta classe representa e sobrescrever o método *test_file_termination*. O nome deve ser passado ao parâmetro *name* do construtor da classe *System*. O nome passado será apresentado com uma das opções do campo *Sistema* do formulário de caso de uso. O método *test_file_termination* é responsável por especificar o formato dos arquivos de testes a serem gerados. Sabemos que a ferramenta *Easy* gera um *script* de teste para cada cenário e esses *scripts* são destinados às ferramentas

responsáveis por executarem testes de interface. Ao sobrescrever o método *test_file_termination* estamos informando o formato que os *scripts* de teste devem possuir. Este método deve retornar somente uma *string* com o formato dos arquivos de testes – *txt*, *xml*, etc. Para esclarecer esse processo, vamos utilizar como exemplo a classe *iOS* presente na ferramenta *Easy*. No construtor da classe, preenchemos o parâmetro *nome* com o valor *iOS*. No método *test_file_termination* retornamos o valor *js*, formato de arquivos *JavaScript*. Isso ocorre devido aos *scripts* destinados à ferramenta *UI Automation* precisarem ser escritos em *JavaScript*.

A segunda classe é a *SystemWidget*, responsável pelo registro dos elementos de interface na ferramenta *Easy*. Semelhante ao modo de utilização da classe *System*, o desenvolvedor deverá criar uma classe para cada elemento de interface a ser cadastrado, tendo *SystemWidget* como superclasse. Além disso, é necessário sobrescrever os métodos de *SystemWidget* com as características do elemento de interface em questão. Na Figura 35 apresentamos o diagrama de classe *SystemWidget*.

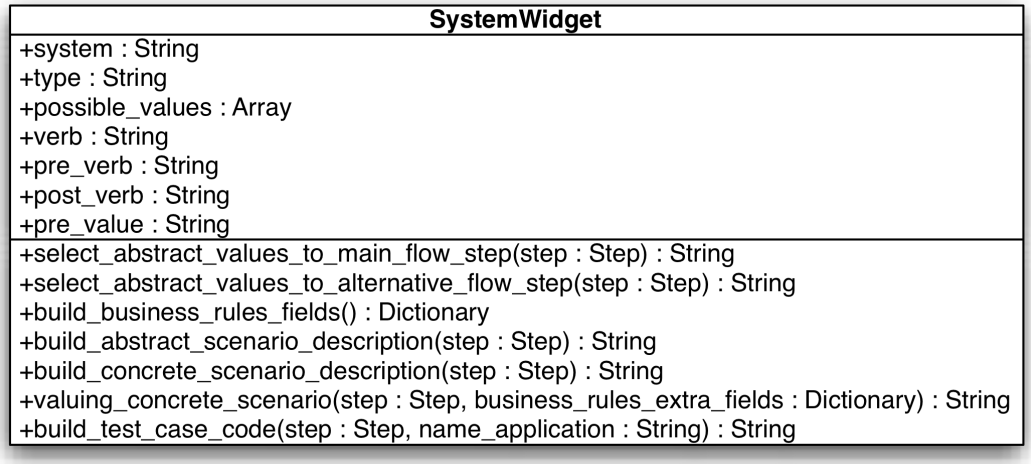


Figura 35: Diagrama de classe de SystemWidget.

Antes de entrarmos em detalhes sobre a classe *SystemWidget,* é necessário explicarmos alguns pontos. Anteriormente, comentamos que o fluxo principal do formulário de casos de uso pode somente possuir um passo para cada elemento de interface. O mesmo ocorre ao criar um fluxo alternativo, onde cada alternativa é direcionada a um único passo. Os cenários criados e apresentados ao desenvolvedor também possuem seus passos, responsáveis por especificar os *valores abstratos* ou *concretos* dos elementos de interface. Os passos de um fluxo

ou cenários são fundamentais para direcionarem os objetivos dos testes nas aplicações, pois esclarecem as ações a serem realizadas pelo usuário ou sistema aos elementos de interface. Devido à importância dos passos nos fluxos e cenários, a ferramenta *Easy* possui uma classe chamada *Step* para o armazenamento de informações relacionadas aos passos durante o processo de geração dos casos de testes. Na Figura 36 é apresentado o diagrama da classe *Step*.

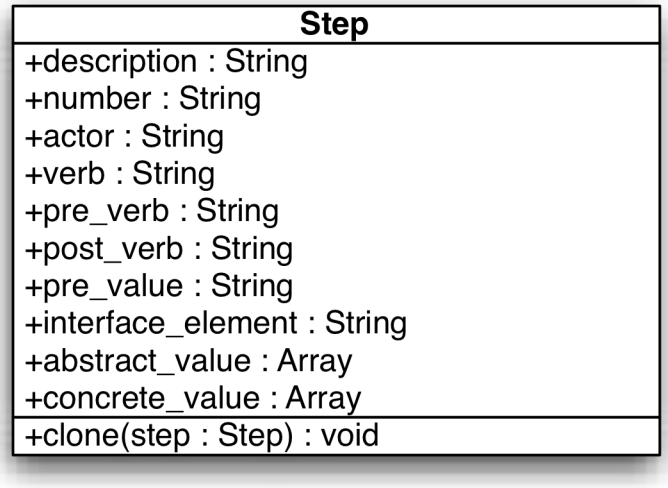


Figura 36: Diagrama de classe de Step.

O objetivo da classe *Step* é permitir ao desenvolvedor acessar informações dos passos de forma simples e objetiva. A ferramenta é responsável por separar as informações contidas na descrição dos passos e disponibilizá-las ao desenvolvedor. Das características expostas no diagrama da Figura 35, os atributos são os mais pertinentes no momento. Segue abaixo a descrição dos atributos da classe *Step*:

- *description*: responsável pela descrição do passo. A expressão que descreve o que ocorre no passo.
- *number*: o número do passo.
- *actor*: nome do ator do passo.
- *verb*: a ação realizada no passo.
- *pre_verb*: o prefixo utilizado na descrição.
- *pos_verb*: o sufixo utilizado na descrição.
- *pre_value*: expressão que antecede o valor abstrato ou concreto no passo. Podemos citar como exemplo a expressão com valor utilizado para caixas de entrada de texto.

- *interface_element*: nome do elemento de interface do passo.
- *abstract_value*: valor abstrato utilizado no passo.
- *concrete_value*: valor concreto utilizado no passo.

Podemos observar na Figura 34 que a maioria dos métodos da classe *SystemWidget* possui um parâmetro do tipo *Step*. Isso ocorre devido à ferramenta ser abstrata do ponto de vista dos elementos de interface, precisando assim do auxílio das classes desses elementos para interpretar cada passo dos fluxos e cenários. No início desta seção esclarecemos que o objetivo da ferramenta *Easy* é ser independente de linguagens de programação. As classes respectivas aos elementos de interface são responsáveis por identificar o que ocorre nos passos da ferramenta. Para deixarmos mais claro a relação entre passos e classes de elementos de interface, vamos observar a Tabela 3. Nesta tabela mostramos em quais etapas da ferramenta são utilizados os métodos das classes de elementos de interface.

Etapas da ferramenta	Métodos de *SystemWidget*
Formulário de casos de uso	*select_abstract_values_to_main_flow_step* *select_abstract_values_to_alternative_flow_step* *build_business_rules_fields*
Geração de *cenários abstratos*	*build_abstract_scenario_description*
Geração de *cenários concretos*	*build_concrete_scenario_description* *valuing_concrete_scenario*
Geração de *scripts* de teste	*build_test_case_code*

Tabela 3: Relação das etapas da ferramenta com a utilização dos métodos das classes dos elementos de interface.

As etapas da ferramenta formulário de casos de uso, geração de *cenários abstratos*, *cenários concretos* e *scripts* de testes – são dependentes das classes que representam os elementos de interface. Na primeira etapa – formulário de casos de uso –, a ferramenta pega os passos do fluxo principal e alternativo descritos pelo desenvolvedor e passa às classes dos elementos de interface criados no campo *Elementos de interface*. Dessa forma, a ferramenta identifica quais *valores abstratos* dos elementos de interface cada passo do fluxo principal e alternativo está utilizando. A segunda etapa – geração de *cenários abstratos* – necessita que as classes construam a descrição dos passos dos cenários com os seus *valores abstratos*. A terceira etapa – geração de *cenários concretos* – precisa das classes

para transformar *valores abstratos* em *concretos* e reconstruir a descrição dos passos, agora incluindo o *valores concretos* ao invés do *valores abstratos*. Por fim, a última etapa – geração *scripts* de testes – depende do módulo para adaptar as informações acumuladas nos passos no formato esperado pelos*scripts* de testes a serem gerados. Compreendendo a dependência da ferramenta em cada uma de suas etapas, podemos comentar as características da classe *SystemWidget*. Abaixo descrevemos os métodos de *SystemWidget*:

- *select_abstract_values_to_main_flow_step*: define o valor abstrato dos passos do fluxo principal descrito no formulário de caso de uso. No caso de mais de um valor abstrato, o método deve retornar um vetor com os valores desejados.
- *select_abstract_values_to_alternative_flow_step*: define os valores abstratos dos passos dos fluxos alternativos descritos no formulário de caso de uso. Cada passo alternativo criado deve possuir somente um valor abstrato.
- *build_business_rules_fields*:adiciona campos exclusivos aos elementos de interface nas regras de negócio do formulário. O retorno do método deve ser um dicionário, onde sua chave/valor deve ser o nome do campo exclusivo e um valor inicial ou vazio, respectivamente.
- *build_abstract_scenario_description*: constrói a descrição de passos de cenários abstratos. O retorno desse método é uma *string* com a descrição do passo.
- *build_concrete_scenario_description*: constrói a descrição de passos de cenários concretos. O retorno desse método é uma *string* com a descrição do passo.
- *valuing_concrete_scenario*: define o valor concreto de passos de cenários concretos. Além do passo, o método também recebe os campos exclusivos do elemento de interface do passo corrente, os quais foram editados pelo desenvolvedor no campo Regras de negócio do formulário. O retorno do método deve ser do tipo *string*.

- *build_test_case_code*: transforma passos de cenários concretos no formato esperado pelos scripts de testes a serem gerados. O retorno desse método é uma *string*.

A ferramenta *Easy* foi desenvolvida utilizando a linguagem de programação *Python*, sendo necessário o desenvolvedor possuir certo conhecimento sobre a linguagem. Ao iniciar o cadastro de uma nova linguagem ou sistema na ferramenta *Easy*, deve-se criar um arquivo em *Python* contendo a classe responsável pelo sistema – subclasse de *System* – e as responsáveis pelos elementos de interface – subclasse de *SystemWidget*. No caso do suporte ao*iOS*, foi desenvolvido um arquivo *ios.py*. Após finalizar o desenvolvimento das novas classes, deve-se instanciá-las no arquivo *system.py* da ferramenta. Os elementos de interface são cadastrados na ferramenta durante sua inicialização.

4.5
Diagrama de classes

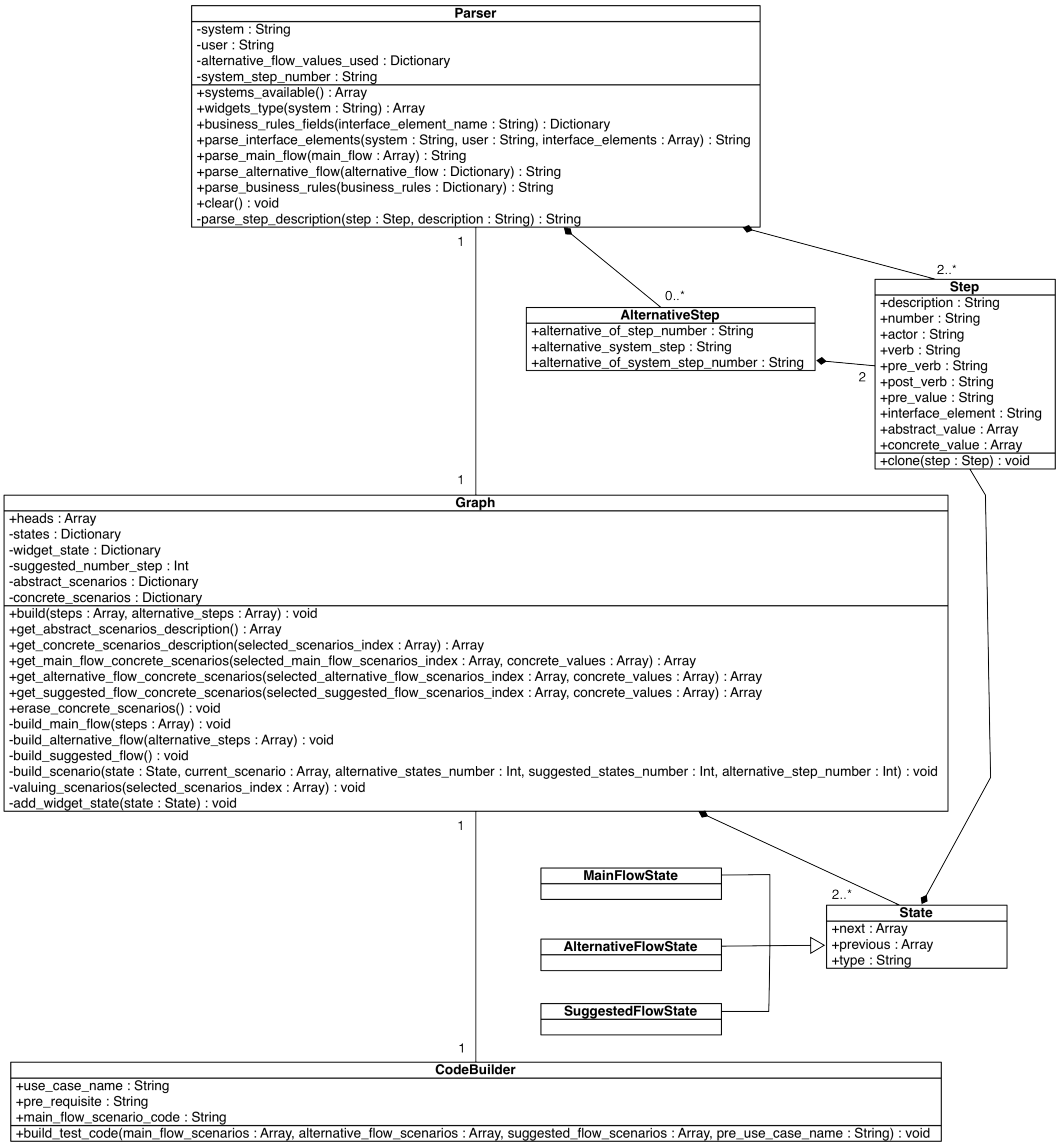


Figura 37: Diagrama das classes responsáveis pelo processo da ferramenta Easy.

Na Figura 37 apresentamos o diagrama de classes das principais classes da ferramenta *Easy*. A classe *Parser* é responsável por interpretar as informações passada pelo usuário ao formulário da ferramenta. A classe *Step* e *AlternativeStep* têm o objetivo de encapsular as informações dos passos presentes no fluxo principal e alternativo, respectivamente. Em seguida, a classe *Graph* coleta os dados identificados em *Parser* e constrói a máquina de estados. Para identificar os diferentes fluxos na máquina, utilizamos as classes *MainFlowState*, *AlternativeFlowState* e *SuggestedFlowState*. Por fim, a classe *CodeBuilder* recebe os cenários da classe*Graph* e gera os *scripts* de testes.

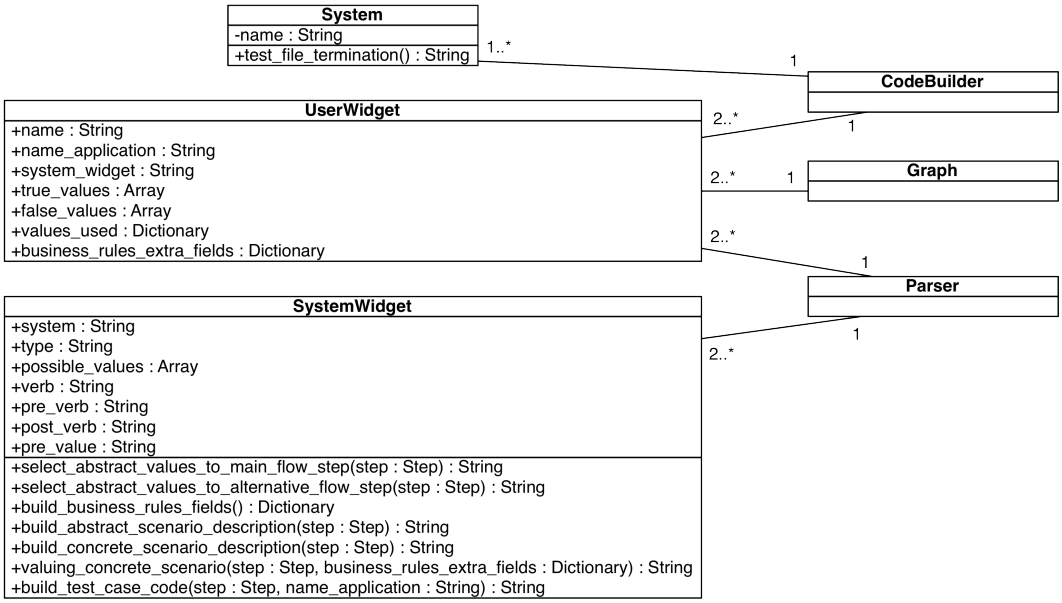


Figura 38: Diagrama de classes do módulo widget.

Na Figura 38 apresentamos as classes pertencentes ao módulo *widget* da ferramenta. A classe *UserSystem* é responsável por fornecer os dados sobre os elementos de interface cadastrados no formulários da ferramenta. A função das classes *System* e *SystemWidget* foi explicada na seção anterior, *Cadastro de novos elementos de interface*. É importante observar que essas classes auxiliam as principais classes do sistema.

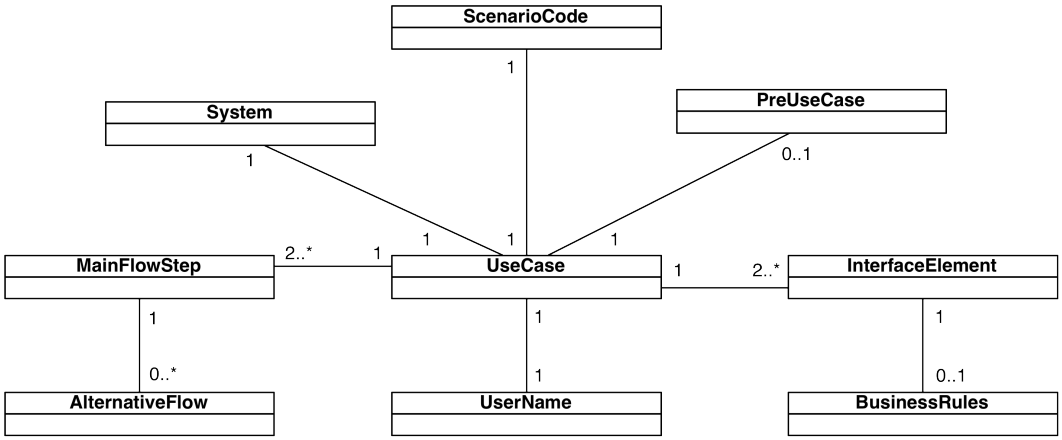


Figura 39: Diagrama das tabelas da base de dados.

A Figura 39 mostra as tabelas presentes na base de dados da ferramenta. O objetivo é armazenar as informações de um caso de uso passado à ferramenta. Com isso, permitimos ao desenvolvedor reutilizar casos de uso, sem a necessidade de preenchê-los novamente e a realização da funcionalidade de segmentação de casos de uso. A tabela *UseCase* é responsável por salvar o nome dos casos de uso, *PreUseCase* por criar o relacionamento entre casos de uso que possuam pré-

requisito, *System* por armazenar o sistema usado no caso, *UserName* pelo nome do protagonista atuante nos fluxos, *InterfaceElement* pelos elementos de interface utilizados nos casos de uso, *MainFlowStep* por armazenar a descrição dos passos do fluxo principal, *AlternativeFlow* por salvar a descrição do passo alternativo e o número do passo do fluxo principal para o qual este é uma alternativa, *BusinessRules* pelas regras de negócio utilizadas pelo usuário e, por fim, *ScenarioCode* que salva a codificação de *scripts* do fluxo principal dos casos de uso para, posteriormente, serem usados na segmentação de casos de uso.

5
Prova de conceito

Neste capítulo apresentamos os resultados, dificuldades e soluções encontradas ao aplicar o processo de geração de casos de testes proposto a duas aplicações: um exemplo de janela de autenticação e um sistema real em uso no mercado, e o aplicativo para dispositivos móveis *Intensity Alarm* [Ferreira, 2013]. Este capítulo é dividido duas seções, onde cada uma é respectiva a uma das aplicações testadas pela ferramenta *Easy*. Em cada seção iniciamos com uma breve introdução sobre aplicação em teste, em seguida apresentamos as funcionalidades da aplicação a serem testadas e como o processo foi aplicado a elas; também demonstramos um exemplo da utilização da ferramenta *Easy* na aplicação e finalizamos com resultados e considerações sobre os experimentos. No final de cada seção é apresentado um comparativo entre o tempo gasto na criação de *scripts* de testes manualmente e utilizando a ferramenta *Easy*.

5.1
Janela de autenticação

A primeira aplicação a ser testada com o processo proposto é um exemplo de janela de autenticação, o mesmo utilizado nos capítulos anteriores para auxiliar na explicação do funcionamento da ferramenta *Easy*. A escolha desse exemplo deu-se através da constante necessidade dos sistemas reais em autenticar seus usuários. Além disso, acreditamos que ao começarmos por um exemplo conhecido, facilitamos o entendimento da aplicação do processo em sistemas reais.

A aplicação em questão foi desenvolvida exclusivamente para a elaboração desta análise. Ela consiste em uma única janela que possui quatro campos: uma caixa de entrada de texto para o nome do usuário, uma caixa de entrada de texto para a senha, um botão que inicia a autenticação ao ser clicado e uma legenda onde o sistema apresenta o resultado da autenticação. Essa aplicação foi desenvolvida em *Objective-C* para dispositivos móveis do tipo *iPhone.*

Figura 40: Janela de autenticação da aplicação de exemplo.

5.1.1. Aplicação do processo

A aplicação de exemplo utilizada possui como única funcionalidade a autenticação de usuários. Para iniciarmos o processo, construímos a descrição do caso de uso respectivo a funcionalidade "Realizar autenticação". A Figura 41 apresenta o caso de uso da funcionalidade "Realizar autenticação" segundo o padrão apresentado em [Staa, 2013].

Caso de uso	Realizar autenticação	
Resumo	Usuário tenta autenticar-se no sistema para poder utilizá-lo.	
Escopo	Usuário inicializa a aplicação em um smartphone e realiza autenticação para poder utilizá-la.	
Atores	Usuário	Obter autorização para utilizar a aplicação.
	Sistema	Permitir somente usuário cadastrados a utilizarem o sistema.
Pré-condições	Usuário inicializa a aplicação em um smartphone.	
Fluxo principal	1. O usuário preenche o campo Nome. 2. O usuário preenche o campo Senha. 3. O usuário clica no botão Entrar. 4. O sistema valida as informações passadas. 5. O sistema apresenta um mensagem confirmando a autenticação.	
Fluxos alternativos	Evento E1/1: O usuário preenche o campo Nome com valor vazio. • E1.1 O sistema exibe a mensagem "Nome do usuário está vazio.".	

	• E1.2 Repete a partir de 1. Fim evento E1.
	Evento E2/1: O usuário preenche o campo Nome com valor não cadastrado. • E2.1 O sistema exibe a mensagem “Nome do usuário desconhecido.”. • E2.2 Repete a partir de 1. Fim evento E2.
	Evento E3/2: O usuário preenche o campo Senha com valor vazio. • E3.1 O sistema exibe a mensagem “Senha do usuário está vazio.”. • E3.2 Repete a partir de 2. Fim evento E3.
	Evento E4/2: O usuário preenche o campo Senha com valor não cadastrado. • E4.1 O sistema exibe a mensagem “Senha do usuário desconhecido.”. • E4.2 Repete a partir de 2. Fim evento E4.
Pós-condições	O usuário está autenticado e pode acessar a aplicação.
Regras de negócio	Nome 1. Precisa estar cadastrado. 2. Não pode ser vazio. Senha 1. Precisa estar cadastrada e relacionada a um nome. 2. Não pode ser vazia.

Figura 41: Descrição do caso de uso “Realizar autenticação”.

Após concluirmos a descrição do caso de uso, passamos este à ferramenta *Easy*. Anteriormente, comentamos que o processo de geração de *scripts* de testes da ferramenta *Easy* possui quatro etapas. A primeira é a descrição do caso de uso na interface gráfica da ferramenta *Easy*. A segunda é a geração dos *cenários abstratos* apresentando-os ao usuário. A terceira é a transformação de *cenários abstratos* em *cenários concretos*. Por fim, na quarta e última etapa, são gerados os *scripts* de testes. Na Figura 42 é apresentado o formulário de casos de uso da ferramenta *Easy* preenchido com o caso de uso “Realizar autenticação”, apresentado na Figura 41. Na Figura 43 e 44 são mostrados os *cenários abstratos* e *concretos* pela interface gráfica da ferramenta, respectivamente. A Figura 45 apresenta os *scripts* de testes resultantes do processo. Importante comentar que a ferramenta gera um arquivo de *scripts* para cada caso de teste. Para facilitar a visualização, a Figura 45 possui todos os *scripts* em um mesmo arquivo.

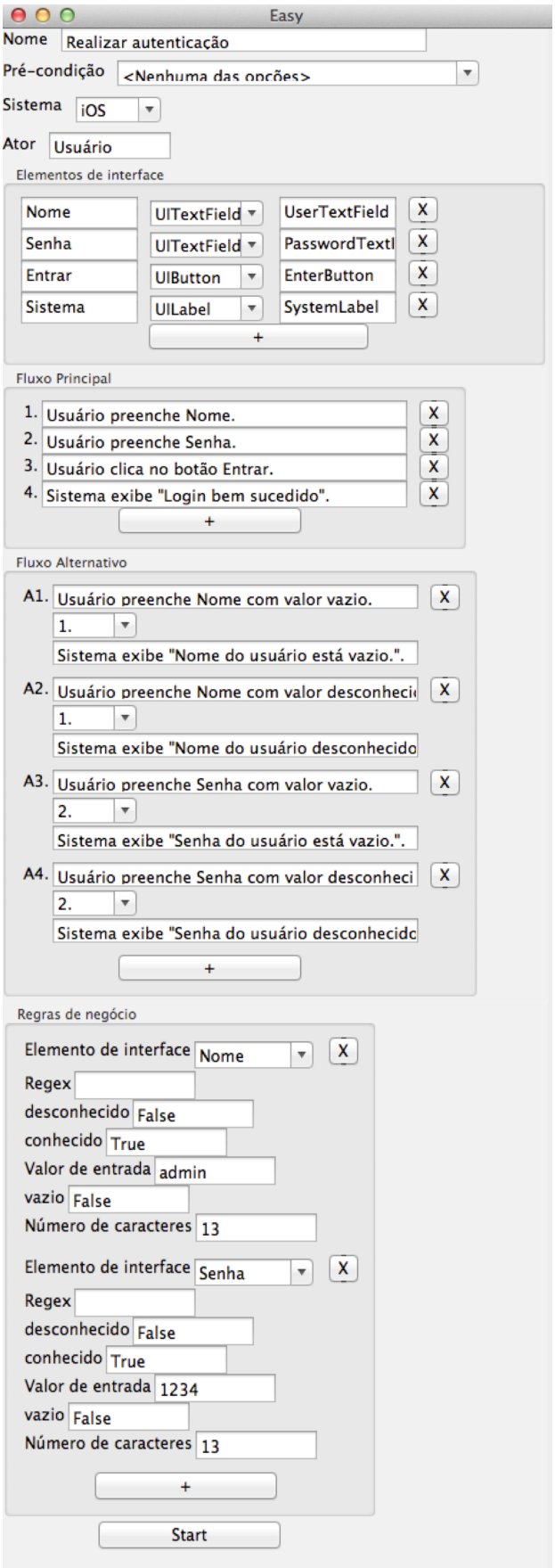
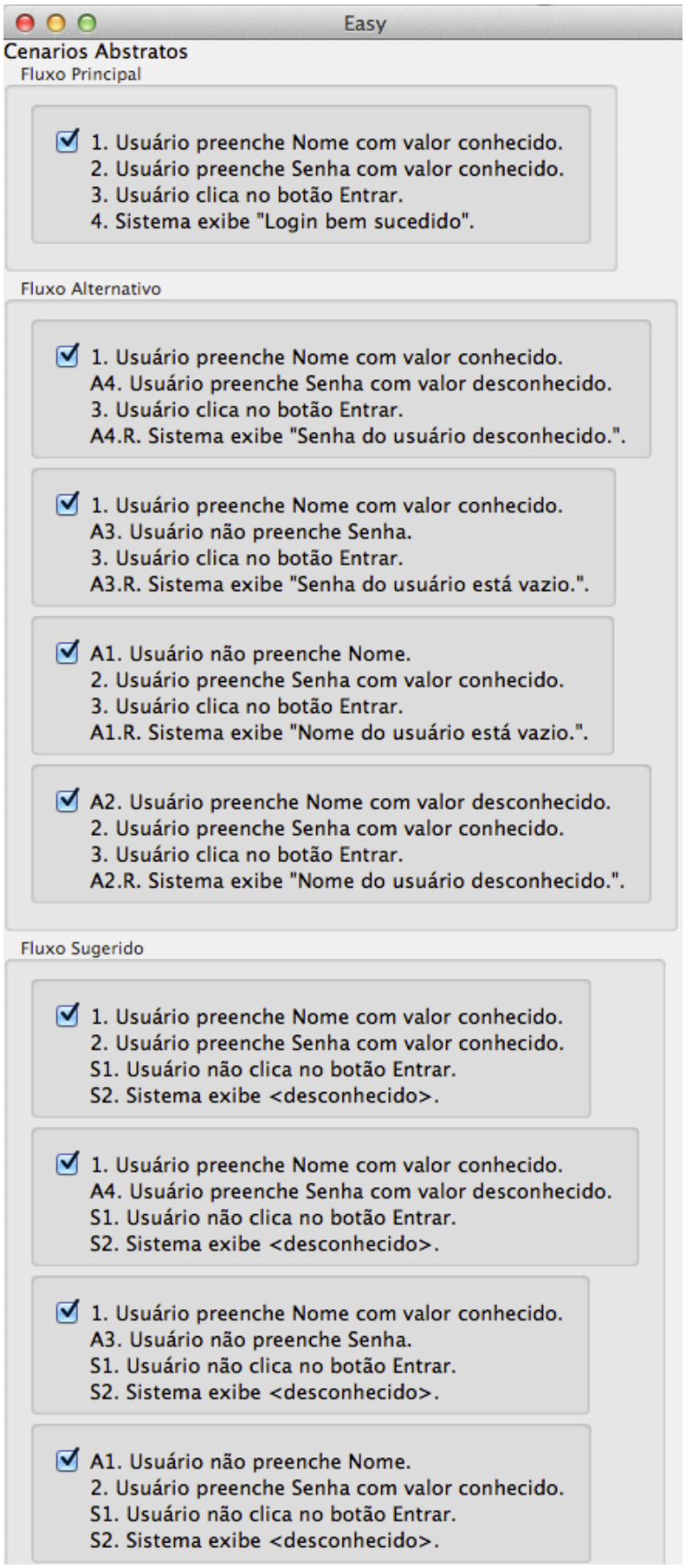


Figura 42: Formulário de casos de uso preenchido com “Realizar autenticação”.



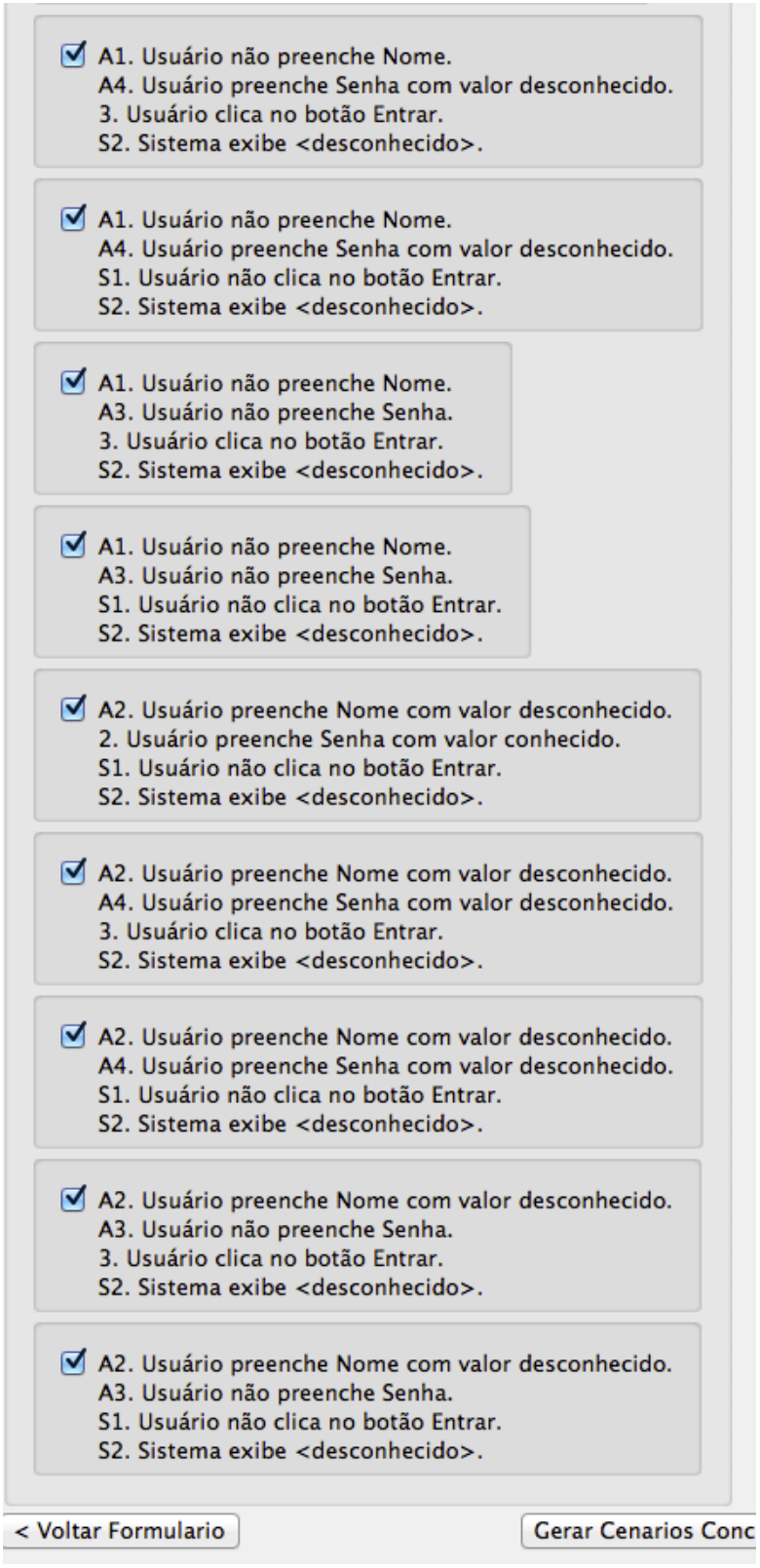
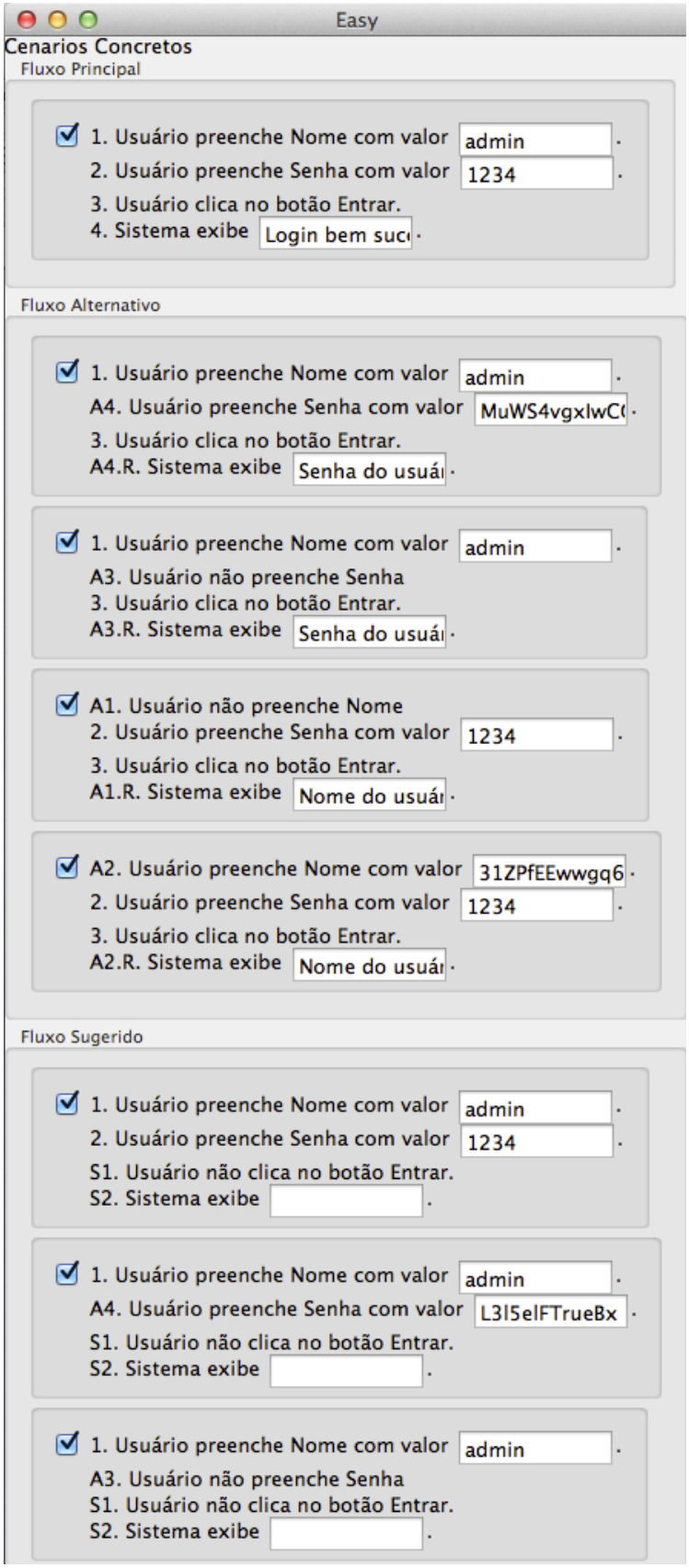


Figura 43: Cenários abstratos do caso de uso “Realizar autenticação”.



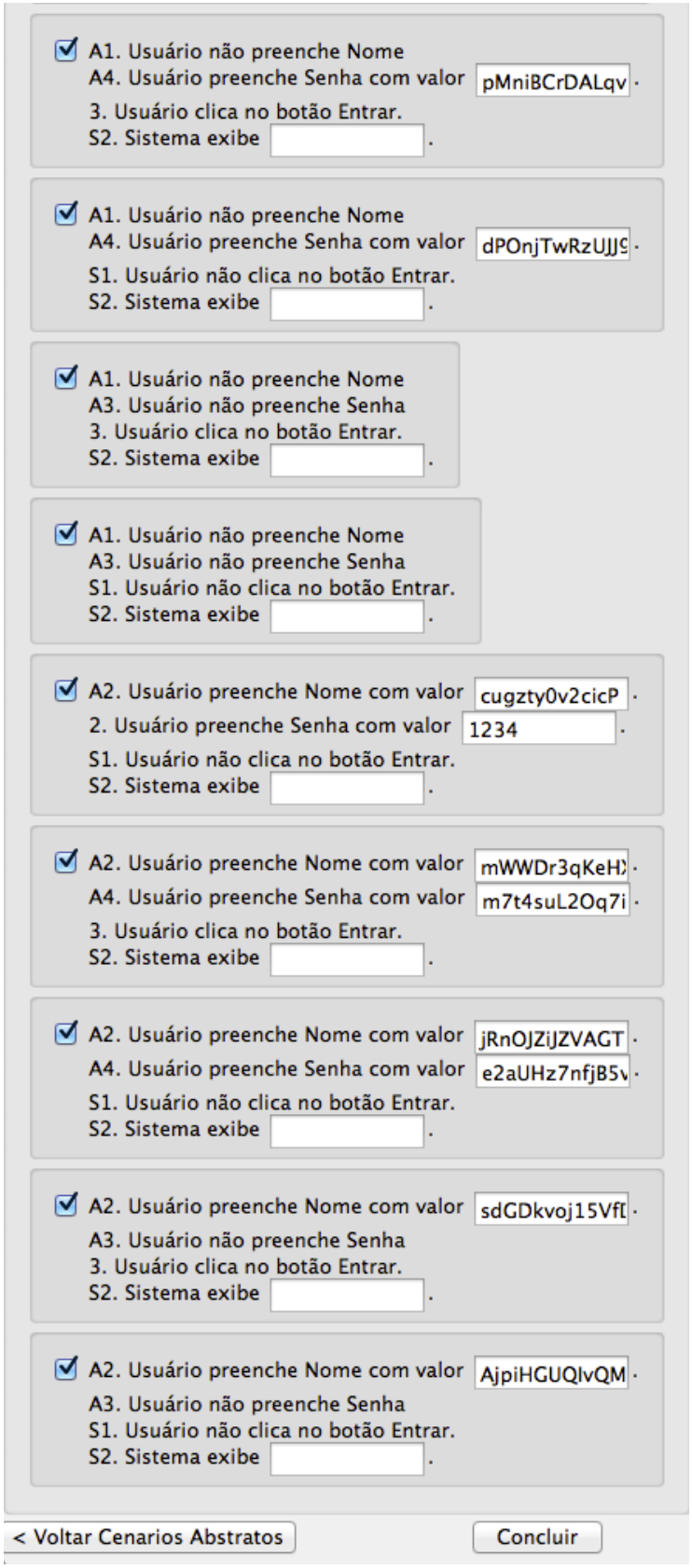


Figura 44: Cenários concretos do caso de uso “Realizar autenticação”.

```
//////////////////////
// FLUXO PRINCIPAL //
//////////////////////

// CENARIO M1
UIALogger.logMessage("Usuário preenche Nome com valor \"admin\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("admin");

UIALogger.logMessage("Usuário preenche Senha com valor \"1234\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("1234");

UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe \"Login bem sucedido\".");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Login bem sucedido") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}

//////////////////////////
// FLUXO ALTERNATIVO //
//////////////////////////

// CENARIO A1
UIALogger.logMessage("Usuário preenche Nome com valor \"admin\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("admin");

UIALogger.logMessage("Usuário preenche Senha com valor \"MuWS4vgxIwCQe\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("MuWS4vgxIwCQe");

UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe \"Senha do usuário desconhecido.\".");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Senha do usuário desconhecido.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}

// CENARIO A2
UIALogger.logMessage("Usuário preenche Nome com valor \"admin\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("admin");

UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe \"Senha do usuário está vazio.\".");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Senha do usuário está vazio.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}

// CENARIO A3
UIALogger.logMessage("Usuário preenche Senha com valor \"1234\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("1234");

UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe \"Nome do usuário está vazio.\".");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Nome do usuário está vazio.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}

// CENARIO A4
UIALogger.logMessage("Usuário preenche Nome com valor \"31ZPfEEwwgq6G\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("31ZPfEEwwgq6G");

UIALogger.logMessage("Usuário preenche Senha com valor \"1234\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("1234");

UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe \"Nome do usuário desconhecido.\".");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Nome do usuário desconhecido.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}

//////////////////////
// FLUXO SUGERIDO //
//////////////////////

// CENARIO S1
UIALogger.logMessage("Usuário preenche Nome com valor \"admin\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("admin");

UIALogger.logMessage("Usuário preenche Senha com valor \"1234\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("1234");

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Sem mensagem") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
```

```
// CENARIO S2
UIALogger.logMessage("Usuário preenche Nome com valor \"admin\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("admin");

UIALogger.logMessage("Usuário preenche Senha com valor \"L3l5elFTrueBx\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("L3l5elFTrueBx");

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Sem mensagem") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S3
UIALogger.logMessage("Usuário preenche Nome com valor \"admin\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("admin");

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Sem mensagem") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S4
UIALogger.logMessage("Usuário preenche Senha com valor \"pMniBCrDALqvj\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("pMniBCrDALqvj");

UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Nome do usuário está vazio.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S5
UIALogger.logMessage("Usuário preenche Senha com valor \"dPOnjTwRzUJJ9\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("dPOnjTwRzUJJ9");

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Nome do usuário está vazio.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S6
UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Nome do usuário está vazio.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S7
UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Sem mensagem") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S8
UIALogger.logMessage("Usuário preenche Nome com valor \"cugzty0v2cicP\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("cugzty0v2cicP");

UIALogger.logMessage("Usuário preenche Senha com valor \"1234\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("1234");

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Sem mensagem") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S9
UIALogger.logMessage("Usuário preenche Nome com valor \"mWWDr3qKeHXpn\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("mWWDr3qKeHXpn");

UIALogger.logMessage("Usuário preenche Senha com valor \"m7t4suL2Oq7iV\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("m7t4suL2Oq7iV");

UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Nome do usuário desconhecido.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
```

```
// CENARIO S10
UIALogger.logMessage("Usuário preenche Nome com valor \"jRnOJZiJZVAGT\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("jRnOJZiJZVAGT");

UIALogger.logMessage("Usuário preenche Senha com valor \"e2aUHz7nfjB5v\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["PasswordTextField"].setValue("e2aUHz7nfjB5v");

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Sem mensagem") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S11
UIALogger.logMessage("Usuário preenche Nome com valor \"sdGDkvoj15VfD\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("sdGDkvoj15VfD");

UIALogger.logMessage("Usuário clica no botão Entrar.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["EnterButton"].tap();

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Nome do usuário desconhecido.") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
// CENARIO S12
UIALogger.logMessage("Usuário preenche Nome com valor \"AjpiHGUQIvQMJ\".");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["UserTextField"].setValue("AjpiHGUQIvQMJ");

UIALogger.logMessage("Sistema exibe <desconhecido>.");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["SystemLabel"].value() == "Sem mensagem") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
```

Figura 45: Scripts de testes do caso de uso “Realizar autenticação”.

5.1.2. Resultados e considerações

Nesta seção comentamos os resultados, dificuldade e soluções ao utilizarmos a ferramenta *Easy* no caso de uso “Realizar autenticação”. Para avaliarmos a eficiência do método proposto, produzimos em paralelo o mesmo número de *scripts* de testes por dois métodos: manualmente e usando a ferramenta *Easy*. No processo manual, iniciamos listando os cenários derivados do caso de uso “Realizar autenticação”. Em seguida, produzimos os *scripts* de testes manualmente segundo os padrões aceitos pela ferramenta *UI Automation*. Após conclusão do processo manual, começou-se o da ferramenta *Easy*. A Tabela 4 faz um comparativo do tempo gasto nos dois processos. Os tempos apresentados na Tabela 4 são resultantes da geração de 18 *scripts* de testes por cada processo.

Etapas	Manual (min.)	Easy (min.)
Listagem de cenários	8	-
Codificação	31:30	-
Preenchendo formulário	-	4:35
Manipulando *cenários abstratos*	-	0:28
Manipulando *cenários concretos*	-	3:11
Total	39:30	8:14

Tabela 4: Comparação do tempo gasto entre o processo manual e a ferramenta Easy.

Ao compararmos o tempo gasto nos dois processos, fica clara a agilidade da produção de testes com a ferramenta *Easy*. Os problemas enfrentados no processo manual resumem-se a volume de informações, organização, falta de atenção do desenvolvedor e formulação superficial do suíte de teste. Durante a listagem dos cenários, notou-se a necessidade de atenção do desenvolvedor na identificação de possíveis cenários e em evitar a repetição destes. A formulação do suíte de teste pode ser superficial dependendo do número cenários, pois o desenvolvedor poderá não conhecer todas as possibilidades de cenários. Além disso, listar os cenário antes de iniciar a produção dos *scripts* de testes contribuiu com a redução de possíveis erros humanos por déficit de atenção, uma prática de organização que nem sempre será utilizada por desenvolvedores. A segunda etapa – codificação dos *scripts* – também deixou clara a possibilidade do desenvolvedor gerar testes contendo erros de programação. Devido aos cenários possuírem certas semelhanças, os *scripts* gerados são parecidos e podem prejudicar o desenvolvedor a identificar erros na codificação dos scripts. Outro problema identificado é o número de cenários, que influencia diretamente o custo de produção dos testes. Quanto maior o número, maior o esforço do desenvolvedor ao implementá-los.

A ferramenta *Easy* mostrou-se eficiente na geração de *scripts* de testes. A descrição dos casos de testes com casos de uso reduziu o esforço do desenvolvedor na criação dos testes, principalmente tendo em mãos a descrição do caso de uso, como o apresentado na Figura 38. A geração automática de cenários foi um grande diferencial entre os dois processos. Enquanto no processo manual o desenvolvedor preocupa-se em garantir que todos os cenários a serem tratados estão corretamente definidos, na ferramenta *Easy* essa dificuldade não existiu. A

apresentação dos cenários pela interface gráfica e com linguagem natural restrita facilitou a identificação pelo desenvolvedor dos conteúdos dos cenários, proporcionando a opção de selecionar quais deles deveriam transformar-se em *scripts* de testes.

Sabe-se que a aplicação de exemplo testada possui uma interface gráfica simples, o que ajudou com o pequeno tempo gasto na utilização da ferramenta *Easy*. Caso a ferramenta não tivesse suporte a algum elemento de interface utilizada na aplicação teste, o usuário teria de cadastrá-lo na ferramenta, o que consumiria mais tempo. É importante comentar que os tempos coletados não foram influenciados pela curva de aprendizado do desenvolvedor com os dois processos. A análise somente começou após o desenvolvedor ter conhecimento de como gerar os casos de testes, em ambos processos – criar os *scripts* manualmente e operar a ferramenta *Easy*.

5.2 Intensity Alarm

A segunda aplicação a ser testada com o processo proposto chama-se *Intensity Alarm* [Ferreira, 2013]. O *Intensity Alarm* é um aplicativo do tipo despertador, destinado a dispositivos móveis *Apple*, como *iPhone*, *iPad* e *iPodTouch*. Lançado em Março de 2013, o *Intensity Alarm* foi baixado em mais de 50 países e alcançou 46º posição no ranque da categoria *Health & Fitness* da *App Store Brasil*. Atualmente o aplicativo é utilizado por centenas de usuários diariamente e com atualizações mensais de novos conteúdos e funcionalidades.

O *Intensity Alarm* possui as funcionalidades básicas de um despertador como criar, editar e excluir alarmes. Além disso, oferece serviço de meteorologia baseado na posição atual do usuário; suporte às línguas Português e Inglês; compatível a partir dos modelos *iPhone 3GS*, *iPod Touch* (3º geração) e *iPad*;e compilável em versões a partir *iOS* 4.3. O aplicativo possui também características peculiares, como a presença de elementos de interface não existentes na biblioteca *UI Kit* da *Apple*, desenvolvidos exclusivamente para a aplicação, e uma interface gráfica dinâmica, que apresenta diferentes comportamentos gráficos dependendo do horário em que o usuário está utilizando a aplicação.

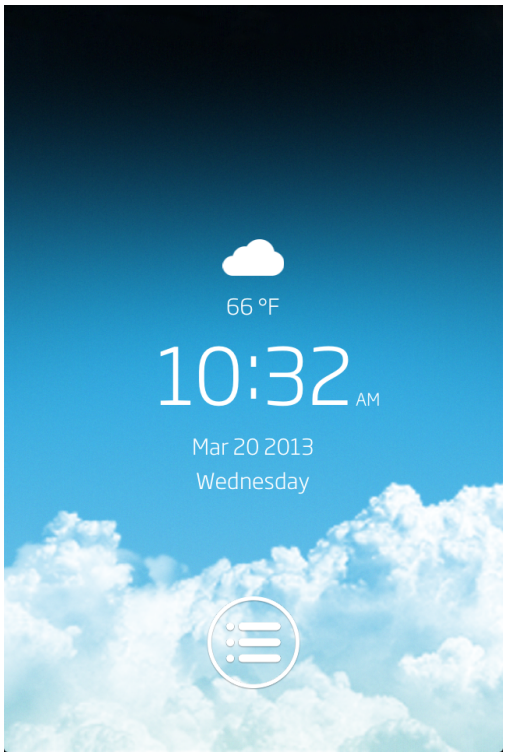


Figura 46: Janela inicial da aplicação Intensity Alarm.

5.2.1. Aplicação do processo

Para iniciarmos a aplicação do processo proposto, decidimos selecionar as principais funcionalidades da aplicação, as quais são mais utilizadas pelos usuários. Antes da aplicação do processo, identificou-se a existência de elementos de interface que não possuíam suporte pela ferramenta *Easy*. Alguns destes são exclusivos do *Intensity Alarm*, não sendo encontrados na biblioteca de interface gráfica *UI Kit* ou em outras aplicações do mercado. Sendo assim, foi necessário cadastrá-los na ferramenta *Easy* antes de iniciar a geração de testes. As funcionalidades escolhidas para a avaliação foram:

- Criar alarme
- Editar alarme
- Excluir alarme
- Iniciar alarme
- Cancelar alarme

Uma característica da maioria dessas funcionalidades é a participação de diferentes janelas da aplicação para a conclusão de uma função. Devido ao tamanho reduzido da tela dos dispositivos móveis, as aplicações tendem a dividir

suas funcionalidades em janelas independentes, ao invés de utilizar abas ou pequenas janelas que sobrepõem a janela principal. É sempre priorizada a atividade corrente do usuário para facilitar a usabilidade da aplicação nos dispositivos móveis. Sendo assim, ao testar uma funcionalidade, é comum a participação de diferentes telas e, por consequência, dos elementos de interface contidos nelas. Os problemas e soluções da utilização de diferentes telas são comentados em *Resultados e considerações* desta seção.

Outro ponto observado antes de iniciar a aplicação do processo são as respostas do sistema ao decorrer das ações realizadas pelo usuário. Muitos aplicativos, inclusive o *Intensity Alarm*, tendem minimizar a utilização de textos e priorizam o uso de imagens – pictogramas na maioria – para representarem os elementos de interface na aplicação. O minimalista das interfaces gráficas reduz a dependência linguística dos usuários. Sendo assim, o sistema pode utilizar diferentes artifícios para comunicar ao usuário sobre a conclusão de uma funcionalidade. Esses artifícios são os oráculos a serem analisados pelos testes de interface gráfica. A utilização de diferentes elementos de interface como oráculos de testes pode ser problemática ao desenvolvedor na elaboração dos *scripts* de testes.

Após identificar as características da aplicação citadas anteriormente e o cadastro dos novos elementos de interface na ferramenta *Easy*, iniciamos a descrição dos casos de uso das funcionalidades escolhidas. Em seguida, passamos os casos de uso para a ferramenta *Easy* e iniciamos a geração de cenários. Em alguns casos de uso, o número de cenários foi extremamente grande para o desenvolvedor manipulá-los, mesmo por intermédio da interface gráfica da ferramenta *Easy*. Nestes casos foi utilizado o recurso de *Segmentação de casos de uso* da ferramenta para reduzir a quantidade de cenários. Para esclarecermos melhor como foi realizado o processo, a subseção *Caso de uso "Criar alarme"* é apresentada a seguir.

5.2.1.1. Caso de uso "Criar alarme"

Iniciamos a avaliação do processo descrevendo o caso de uso da funcionalidade "Criar alarme". Essa funcionalidade é umas das mais importantes do *Intensity Alarm*, obrigando o usuário a percorrer a maioria das janelas presentes na

aplicação. Sendo assim, alguns elementos de interface citados no caso de uso servem exclusivamente para permitir que o usuário consiga chegar à janela de criação de alarme do aplicativo. Na Figura 47 apresentamos a descrição do caso de uso “Criar alarme”.

<table>
<tr><td>Caso de uso</td><td colspan="2">Criar alarme</td></tr>
<tr><td>Resumo</td><td colspan="2">Usuário cria um alarme para despertá-lo.</td></tr>
<tr><td>Escopo</td><td colspan="2">Usuário inicializa a aplicação em um smartphone, vai até a lista de alarmes do aplicativo e inicia o processo de criação de um alarme.</td></tr>
<tr><td rowspan="2">Atores</td><td>Usuário</td><td>Ajustar o despertador.</td></tr>
<tr><td>Sistema</td><td>Permitir criação e a persistência de alarmes.</td></tr>
<tr><td>Pré-condições</td><td colspan="2">Usuário inicializa a aplicação em um smartphone.</td></tr>
<tr><td>Fluxo principal</td><td colspan="2">1. O usuário clica no botão Alarmes.
2. O usuário clica no botão Criar Alarme.
3. O usuário preenche o campo Nome.
4. O usuário ajusta o Relógio.
5. O usuário clica no botão Sons.
6. O usuário clica no botão Tema.
7. O usuário seleciona Intensidade.
8. O usuário clica no botão Selecionar Tema.
9. O usuário clica no botão Salvar Alarme.
10. O sistema persiste as informações passadas.
11. O sistema exibe na lista de alarmes o alarme criado.</td></tr>
<tr><td rowspan="5">Fluxos alternativos</td><td colspan="2">Evento E1/1: O usuário não clica no botão Alarmes.
• E1.1 O sistema exibe a tela de entrada.
• E1.2 Repete a partir de 1.
Fim evento E1.</td></tr>
<tr><td colspan="2">Evento E2/2: O usuário não clica no botão Criar Alarmes.
• E2.1 O sistema exibe a tela de lista de alarmes.
• E2.2 Repete a partir de 2.
Fim evento E2.</td></tr>
<tr><td colspan="2">Evento E3/5: O usuário não clica no botão Sons.
• E3.1 O sistema exibe a tela de criação de alarme.
• E3.2 Repete a partir de 5.
Fim evento E3.</td></tr>
<tr><td colspan="2">Evento E4/8: O usuário não clica no botão Selecionar Tema.
• E4.1 O sistema exibe a tela de temas.
• E4.2 Repete a partir de 8.
Fim evento E4.</td></tr>
<tr><td colspan="2">Evento E5/9: O usuário não clica no botão Salvar Alarme.
• E5.1 O sistema exibe a tela de criação de alarme.
• E5.2 Repete a partir de 9.
Fim evento E5.</td></tr>
</table>

Pós-condições	O usuário criou o alarme e pode utiliza-lo para despertar.
Regras de negócio	Nome 1. Pode receber qualquer valor de entrada. 2. Pode ser vazio. Relógio 1. Qualquer valor de horário é permitido. 2. Valor default é permitido. Tema 1. Somente os temas de som “Praia”, “Velho Oeste”, “Games” e “Cartoon” podem ser selecionados. Intensidade 1. As duas intensidade (baixa e alta) são permitidas.

Figura 47: Descrição do caso de uso “Criar alarme”

Após concluir a descrição do caso de uso, iniciamos o processo ao passar o caso à ferramenta *Easy*. Durante o preenchimento do formulário de casos de uso da ferramenta, percebemos que havia um problema com o fluxo alternativo. Devido à funcionalidade “Criar alarme” obrigar o usuário a percorrer diversas janelas para concluí-la, o sistema utiliza mais de um elemento de interface para informar eventualidades ao usuário. Em outras palavras, os oráculos dos fluxos alternativos são gerados por elementos de interface diferentes do utilizado no fluxo principal. O problema está na exigência da ferramenta *Easy* em permitir somente um elemento de interface encarregado pela resposta do sistema. De qualquer forma, a geração de *scripts* de testes não foi prejudicado, porém essa situação indica que a ferramenta precisa permitir a utilização de mais de um elemento de interface responsável pelo oráculo em um caso de uso. Discutimos com precisão essa situação em *Resultados e considerações* desta seção.

Após o preenchimento do formulário de casos de uso e iniciada a geração de cenários, percebemos um grande número de cenários criados, totalizando 1.536 unidades. Um número de cenários desta proporção torna inviável ao desenvolvedor administrar a geração de massa de testes. Devido a isso, optamos por utilizar o recurso de *Segmentação de casos de uso* da ferramenta para reduzir o número de cenários a serem gerados. O caso de uso “Criar alarme” foi dividido em duas partes. A primeira – “Criar alarme - Parte 1” – é responsável pela duas primeiras linhas do fluxo principal do caso “Criar alarme”. A segunda parte – “Criar alarme - Parte 2” – trata dos demais elementos de interface utilizados no caso “Criar alarme”. A segunda parte possui a primeira como pré-requisito. Assim, o objetivo do caso “Criar alarme - Parte 1” é garantir que testes cheguem

até a janela de criação de alarmes. Em relação ao oráculo, os dois casos de uso possuem elementos de interface diferentes. O primeiro utiliza a legenda responsável pelo título *Novo Alarme* na tela de criação de alarmes. Caso um *script* de teste desse caso seja executado por completo e, por fim, a aplicação encontrar-se na janela de criar alarmes, então o teste foi bem sucedido. A segunda parte mantém o mesmo elemento utilizado no caso "Criar alarme". Utilizando o recurso *Segmentação de casos de uso*, reduzimos o número de cenários para 384. Nas Figuras 48, 49, 50 e 51 apresentamos o formulário, *cenários abstratos*, *cenários concretos* e *scripts* de testes do caso de uso "Criar alarme - Parte 1", respectivamente. Nas Figuras 52, 53, 54 e 55 apresentamos o formulário, *cenários abstratos*, *cenários concretos* e *scripts* de testes do caso de uso "Criar alarme - Parte 2", respectivamente. Devido ao grande número de cenários e *scripts* de testes, as imagens relacionadas ao caso de uso "Criar alarme - Parte 2" mostram somente parte do que realmente foi gerado.

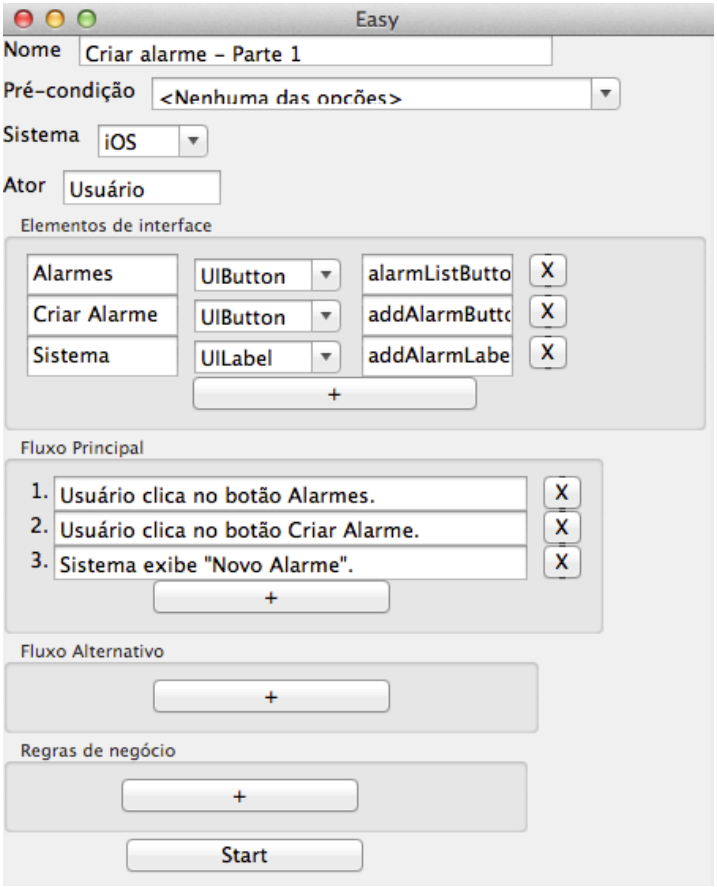


Figura 48: Formulário de caso de uso preenchido com "Criar alarme - Parte 1".

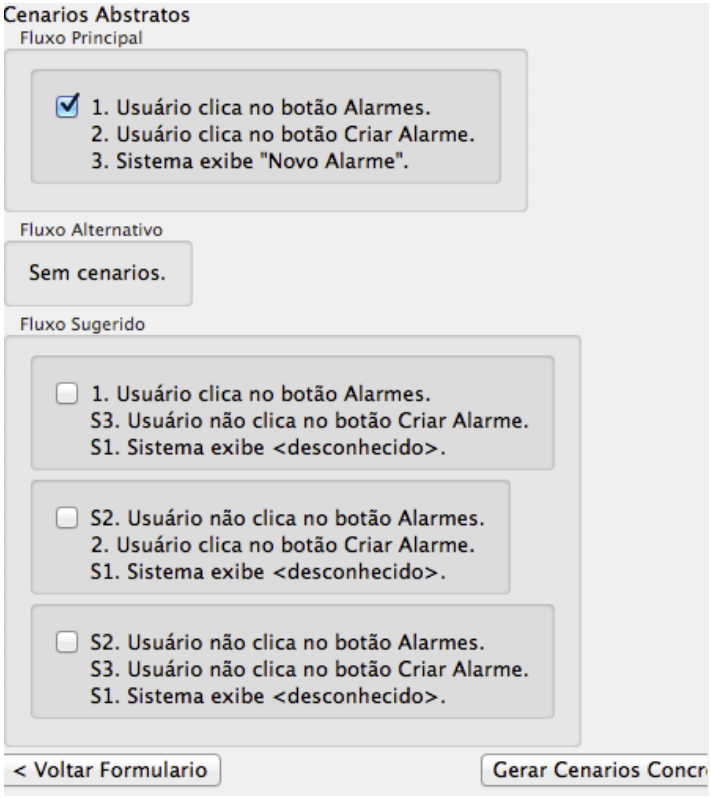


Figura 49: Cenários abstratos do caso de uso “Criar alarme - Parte 1”.

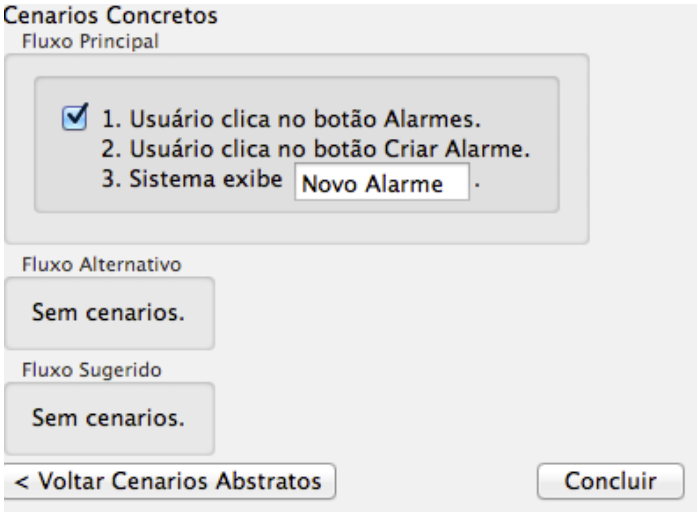


Figura 50: Cenários concretos do caso de uso “Criar alarme - Parte 1”.

```
UIALogger.logMessage("Usuário clica no botão Alarmes.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["alarmListButton"].tap();

UIALogger.logMessage("Usuário clica no botão Criar Alarme.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["addAlarmButton"].tap();

UIALogger.logMessage("Sistema exibe \"Novo Alarme\".");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["addAlarmLabel"].value() == "Novo Alarme") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}
```

Figura 51: Scripts de testes do caso de uso “Criar alarme - Parte 1”.

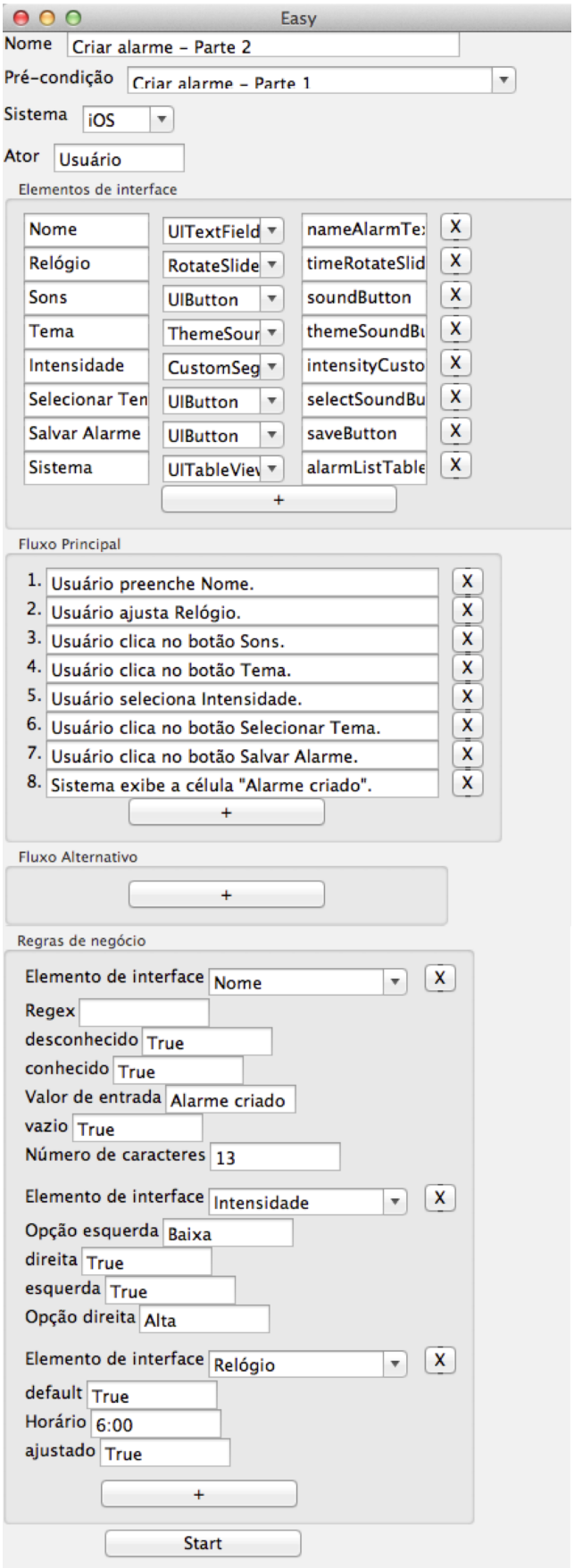


Figura 52: Formulário de casos de uso preenchido com “Criar alarme - Parte 2”.

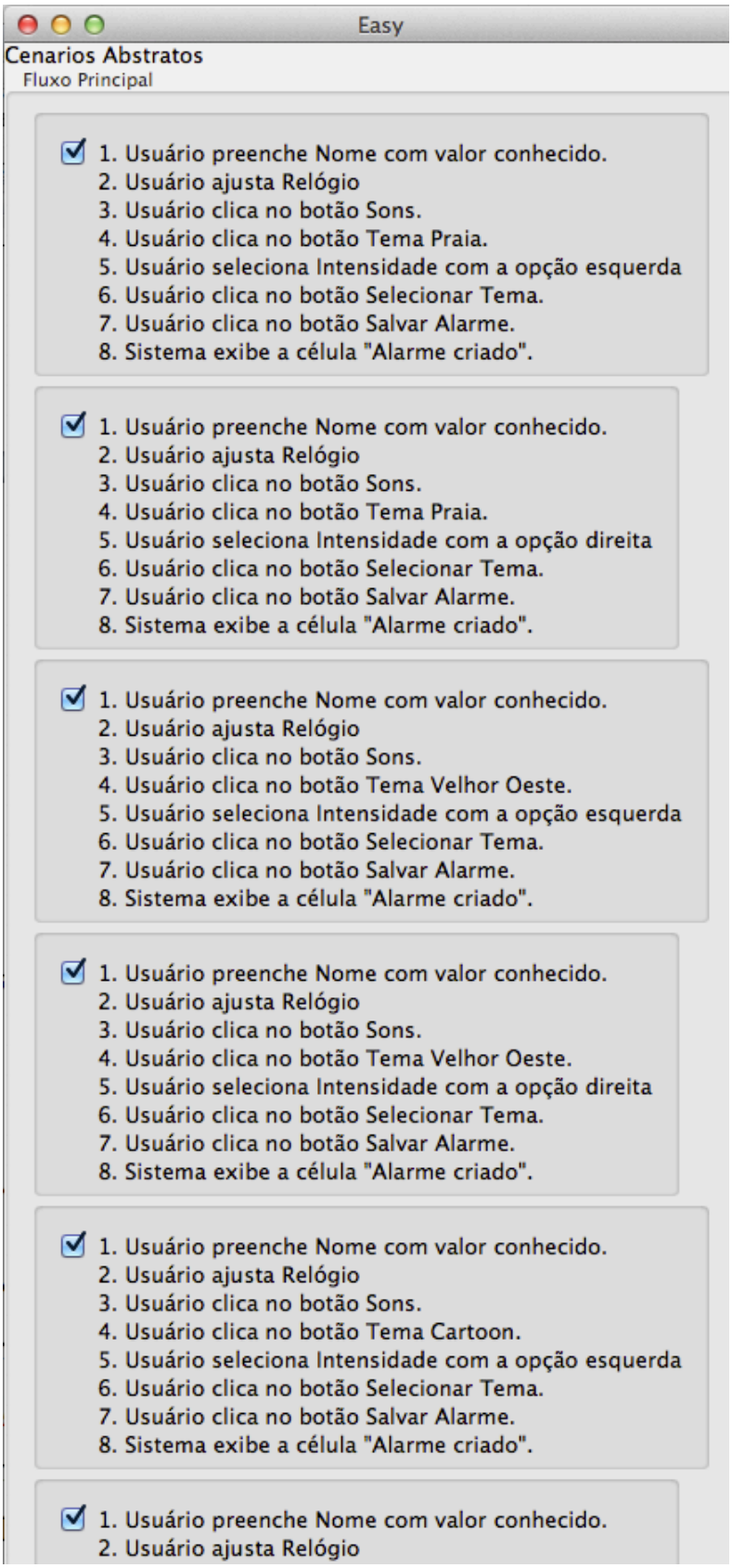


Figura 53: Cenários abstratos do caso de uso “Criar alarme - Parte 2”.

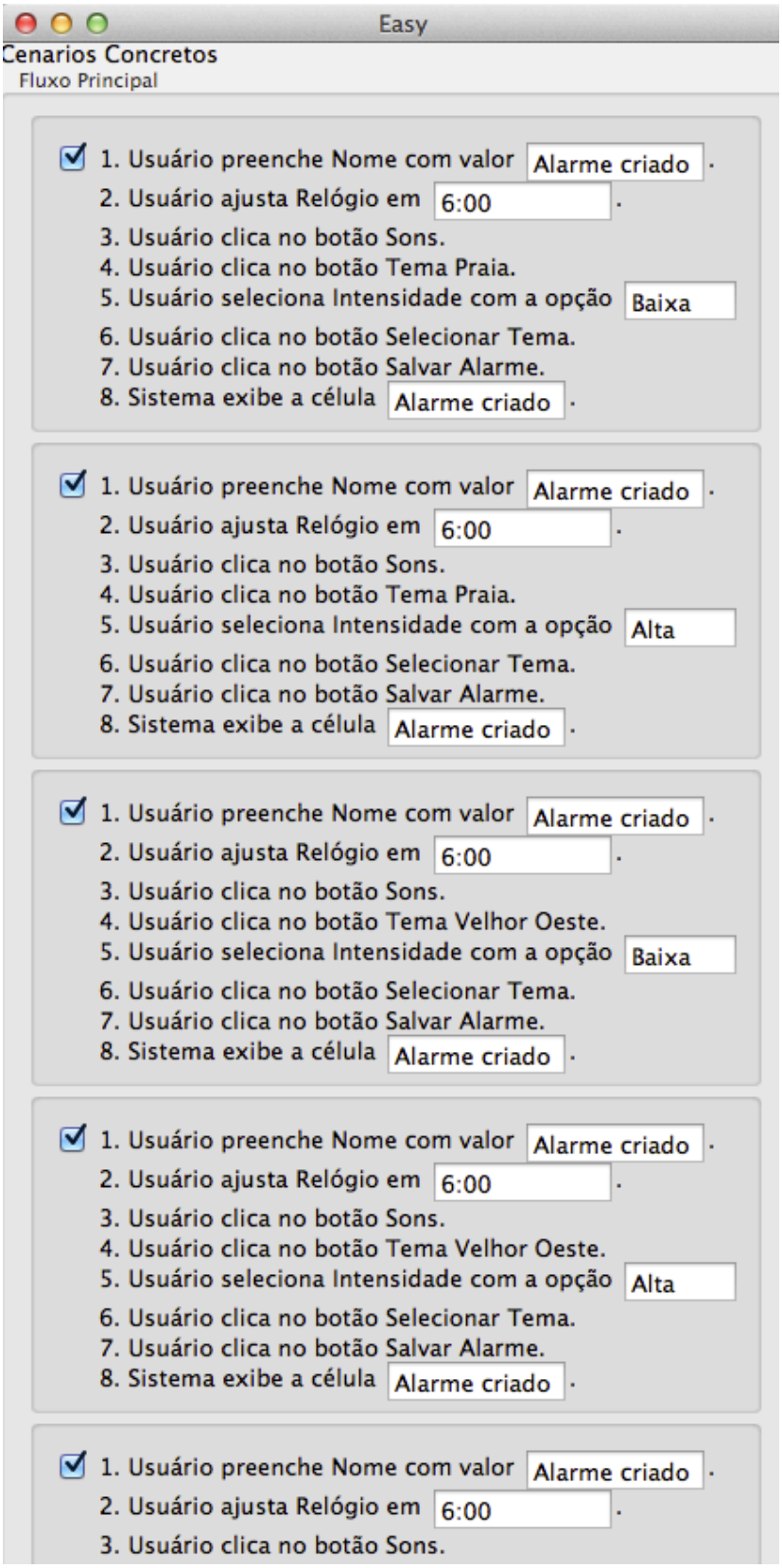


Figura 54: Cenários concretos do caso de uso “Criar alarme - Parte 2”.

```
UIALogger.logMessage("Usuário clica no botão Alarmes.");
able or type UIALogger
get().frontMostApp().mainWindow().buttons()["alarmListButton"].tap();

UIATarget.localTarget().delay(2);

UIALogger.logMessage("Usuário clica no botão Criar Alarme.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["addAlarmButton"].tap();

UIATarget.localTarget().delay(2);

UIALogger.logMessage("Sistema exibe \"Novo Alarme\".");
if (UIATarget.localTarget().frontMostApp().mainWindow().staticTexts()["addAlarmLabel"].value() == "Novo Alarme") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logFail("Fail");
}

UIALogger.logMessage("Usuário preenche Nome com valor \"Alarme criado\"");
UIATarget.localTarget().frontMostApp().mainWindow().textFields()["nameAlarmTextField"].setValue("Alarme criado")
UIATarget.localTarget().tap({x:100, y:100});

UIALogger.logMessage("Usuário ajusta Relógio com valor \"6:00\"");
UIATarget.localTarget().dragFromToForDuration({x:160, y:200},{x:160,y:400},2);

UIALogger.logMessage("Usuário clica no botão Sons.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["soundButton"].tap();

UIATarget.localTarget().delay(2);

UIALogger.logMessage("Usuário clica no botão Tema Praia.");
UIATarget.localTarget().frontMostApp().mainWindow().scrollViews()[0].buttons()["BeachSoundButton"].tap();

UIALogger.logMessage("Usuário seleciona Intensidade com a opção Baixa.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["lowIntensityButton"].tap();

UIALogger.logMessage("Usuário clica no botão Selecionar Tema.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["selectSoundButton"].tap();

UIATarget.localTarget().delay(2);

UIALogger.logMessage("Usuário clica no botão Salvar Alarme.");
UIATarget.localTarget().frontMostApp().mainWindow().buttons()["saveButton"].tap();

UIATarget.localTarget().delay(2);

var cell = UIATarget.localTarget().frontMostApp().mainWindow().tableViews()[0].cells()["Alarme criado"];

if (cell.value() == "Alarme criado") {
    UIALogger.logPass("Pass");
} else {
    UIALogger.logPass("Fail");
}
```

Figura 55: Script de teste do caso de uso “Criar alarme - Parte 2”.

5.2.2. Resultados e considerações

Nesta seção comentamos os resultados, dificuldades e soluções ao utilizarmos a ferramenta *Easy* nos casos de usos selecionados da aplicação *Intensity Alarm*. Semelhante ao realizado na avaliação da tela de autenticação, utilizamos dois processos de geração de *scripts* de testes em paralelo: produção manual e o utilizando a ferramenta *Easy*. O processo manual divide-se em duas partes, onde a primeira é a identificação dos cenários de um caso de uso, e a segunda é a produção dos *scripts* de testes respectivos aos cenários listados. O desenvolvedor recebeu a descrição dos casos de uso no início de ambos processos. A Tabela 5 apresenta o tempo gasto pelo desenvolvedor com o processo manual de geração de *scripts* de testes. A Tabela 6 mostra o tempo consumido pelo desenvolvedor com a ferramenta *Easy*. A Tabela 7 faz um comparativo entre o número de

cenários e *scripts* de teste gerados pelos dois processos. Nesta tabela, os casos de uso que possuem o símbolo *(M)* foram os utilizados no processo manual. Os casos de uso usados na ferramenta *Easy* são identificados pelo símbolo *(E)*.

Casos de uso	Listagem de cenários (min.)	Codificação (min.)	Total (min.)
Criar alarme	21:45	41:52	63:39
Editar alarme	9:56	38:30	48:26
Excluir alarme	-	5:40	5:40
Iniciar alarme	-	4:10	4:10
Cancelar alarme	-	2:01	2:01

Tabela 5: Tempo gasto na geração dos scripts de testes com o processo manual.

Casos de uso	Preenchimento de formulário (min.)	Manipulação de *cenários abstratos* (min)	Manipulação de *cenários concretos* (min.)	Cadastro de novos elementos de interface (min.)	Total (min.)
Criar alarme - Parte 1	1:38	0:11	0:00	-	1:49
Criar alarme - Parte 2	6:07	13:25	1:53	50:29	71:54
Editar alarme - Parte 1	1:59	0:12	0:02	3:12	5:25
Editar alarme - Parte 2	4:56	8:33	1:59	-	15:28
Excluir alarme	2:50	0:27	0:02	1:22	4:41
Iniciar alarme	1:32	0:10	0:01	-	1:43
Cancelar alarme	0:54	0:04	0:01	-	0:59

Tabela 6: Tempo gasto na geração dos scripts de testes com a ferramenta Easy.

Casos de uso	Cenários abstratos	Cenários concretos	Scripts de testes
Criar alarme (M)	-	-	32
Criar alarme – Parte 1 (E)	4	1	1
Criar alarme – Parte 2 (E)	384	32	32
Editar alarme (M)	-	-	32
Editar alarme – Parte 1 (E)	8	1	1
Editar alarme – Parte 2 (E)	384	32	32
Excluir alarme (M)	-	-	1
Excluir alarme (E)	16	1	1
Iniciar alarme (M)	-	-	1
Iniciar alarme (E)	8	1	1
Cancelar alarme (M)	-	-	1
Cancelar alarme (E)	2	1	1

Tabela 7: Número de cenários e scripts de testes gerados em relação aos casos de uso.

Semelhante ao observado no exemplo da tela de autenticação, a ferramenta *Easy* foi capaz de agilizar a geração de *scripts* de testes para um sistema real. O primeiro ponto a ser abordado é o cadastro de novos elementos de interface na ferramenta *Easy*. Ao observarmos a Tabela 5, o cadastro de elementos de interface contribuiu com o aumento do tempo da geração dos testes de alguns casos de uso. O caso de uso com maior tempo dedicado foi o *Criar alarme - Parte 2*, com mais de 50 minutos. Este caso foi o primeiro a precisar cadastrar elementos de interface, onde alguns desses elementos são exclusivos do *Intensity Alarm*, não existindo em outras aplicações no mercado. Ao total, foram cadastrados quatro elementos de interface, sendo três exclusivos da aplicação. Aplicações que possuam elementos de interface exclusivos tendem a ser mais problemáticas na geração de testes – manualmente ou utilizando a *Easy* – devido a seus *scripts*de testes serem mais complexos. No *Intensity Alarm*, a ferramenta de execução de testes de interface *UI Automation* não possui suporte aos seus elementos de interface exclusivos. Sendo assim, o desenvolvedor necessitou buscar formas alternativas para a elaboração dos *scripts* de testes.

Ao compararmos o tempo gasto do caso de uso *Criar alarme* nos dois processos, percebe-se que, somente neste caso, a produção manual dos *scripts* foi mais ágil do que utilizando a ferramenta *Easy*. Porém, ao compararmos o tempo gasto pelos dois processos com todos os casos de uso juntos, observamos que a utilização da ferramenta *Easy* consumiu 19% menos tempo que o processo

manual. Essa diferença é ainda maior em alguns casos de uso, como o *Editar alarme*, onde a ferramenta *Easy* consumiu 43% menos tempo do que o processo manual. É importante comentar que os casos de uso *Criar alarme – Parte 2* e *Editar alarme – Parte 2* utilizam os mesmos elementos de interface, porém como em *Criar alarme – Parte 2* haviam sido realizados os cadastros destes elementos, o caso *Editar alarme – Parte 2* pôde utilizá-los sem perder tempo.

Isso mostra que mesmo com a adição de novos elementos de interface à ferramenta *Easy*, o processo de geração manual é mais custoso ao desenvolvedor. Uma vez tendo os elementos cadastrados, os próximos casos de uso estão isentos deste custo. No processo manual, o desenvolvedor precisa dedicar constantemente tempo aos diferentes cenários possíveis. Além disso, os *scripts* gerados pela ferramenta *Easy* tendem a possuir menos erros de programação do que os gerados manualmente. Ao cadastrar um elementos de interface na *Easy*, o desenvolvedor tende a dedicar mais tempo na elaboração do módulo respectivo ao elemento, tornando-o mais seguro e genérico aos possíveis cenários a serem gerados.

O processo manual apresentou problemas semelhantes aos abordados no exemplo de tela de autenticação. Desta vez, todas as etapas do processo exigiam maior esforço do desenvolvedor para a geração dos testes. A presença de diferentes casos de uso acarretavam um aumento significativo no volume de informações a serem tratadas pelo desenvolvedor. A listagem de cenários exigiu mais atenção do desenvolvedor devido a alguns casos de uso utilizarem diversos elementos de interface. O custo na produção dos *scripts* também aumentou por causa do grande número de elementos de interface, gerando assim *scripts* maiores e mais complexos, principalmente devido aos elementos de interface exclusivos do *Intensity Alarm.* Em suma, o custo do processo manual está diretamente relacionado ao número de casos de uso, quantos elementos de interface estes utilizam e a complexidade desses elementos na aplicação a ser testada. Na ferramenta *Easy*, somente o cadastro de novos elementos implicou um aumento considerável ao custo do desenvolvedor. As demais etapas da ferramenta mantiveram o custo da geração de testes baixo ao desenvolvedor.

A funcionalidade *Segmentação de casos de uso* mostrou bons resultados ao reduzir drasticamente o número de cenários a serem gerados nos casos de uso. Os casos de uso *Criar alarme - Parte 2*, *Editar alarme - Parte 2* e *Cancelar alarme* utilizaram a funcionalidade, porém a vantagem da segmentação ficou evidente nos

dois primeiros casos. Inicialmente, os casos de uso *Criar alarme* e *Editar alarme* foram passados à ferramenta sem serem segmentados e geraram 1.536 e 3.072 cenário, respectivamente. Devido ao grande número de cenários, o desenvolvedor optou por segmentar cada caso em duas partes, gerando assim 4 cenários para caso de uso *Criar alarme - Parte 1*, 384 para *Criar alarme - Parte 2*, 8 para *Editar alarme - Parte 1* e 384 para *Editar alarme - Parte 2.* Ao observarmos os tempos de manipulação de *cenários abstratos* da Tabela 5, percebemos que o desenvolvedor gastaria, no mínimo, quatro vezes mais tempo manipulando os cenários desses casos de uso caso não fosse utilizado a segmentação. A *Segmentação de casos de uso* foi um artifício fundamental para a redução do custo da geração de casos de testes ao desenvolvedor.

Outro ponto a ser abordado foi a dificuldade encontrada devido ao *Intensity Alarm* utilizar diferentes tipos de elementos de interface como oráculos e em janelas diferentes. Essa situação é citada na seção *Caso de uso"Criar alarme"*, onde comentamos que essa prática é comum nas aplicações destinadas a dispositivos móveis. Devido ao protótipo da ferramenta *Easy* exigir um único elemento de interface responsável pelo oráculo dos cenários, somente foi possível gerar cenários derivados do fluxo principal, pois estes possuem um oráculo em comum. Somente os cenários que percorressem todas as telas relacionadas a uma funcionalidade é que poderiam ter seus *scripts* gerados. Ficou clara a importância da ferramenta permitir a utilização de diferentes elementos de interface como oráculos na geração de testes para aplicações de dispositivos móveis. É importante esclarecer que esta situação é gerada devido à falta de uma funcionalidade do protótipo gerado e não um ponto falho do método proposto.

A ferramenta foi capaz de gerar cenários do fluxo sugerido pela simples concatenação dos *valores abstratos* dos elementos de interface que não participavam no fluxo principal. Porém estes cenários também não puderam ser utilizados devido aos elementos de interface responsáveis pelos oráculos. Parte do tempo apresentado na coluna *Manipulação de cenários abstratos* da Tabela 5 foi ocasionado pelo desenvolvedor desabilitar um grande número de cenários gerados que não seriam utilizados na geração de testes. Os cenários desabilitados foram os do fluxo sugerido. Caso a ferramenta *Easy* possuísse o recurso de grupos condicionais, onde o desenvolvedor especifica que determinados elementos de interface precisam satisfazer um condição para serem utilizados em conjunto, o

número de cenários seria menor e mais objetivo. Esse recurso foi implementado no protótipo apresentado em [Caldeira, 2010].

Por fim, não foram encontradas falhas com os casos de testes produzidos. O motivo para tal pode ser devido as aplicações para dispositivos móveis serem mais simples, com poucas funcionalidades, ao compararmos com sistemas destinados a computadores pessoais ou aplicações web. Dessa forma, o processo de desenvolvimento tende a testar massivamente as funcionalidades básicas da aplicação. Outro ponto importante é o histórico de falhas do *Intensity Alarm* no mercado. Em seus quatro meses, não houve a ocorrência de falhas identificadas pelos usuários.

6
Avaliação com usuários

Neste capítulo apresentamos os resultados de um estudo realizado com usuários utilizando a ferramenta *Easy*. O procedimento consiste em submeter os participantes ao processo proposto de geração de casos de testes para aplicações fictícias e a realização de uma entrevista sobre a experiência ao utilizar a ferramenta. O objetivo dessa pesquisa é coletar a opinião dos participantes sobre as vantagens e desvantagens da ferramenta e sugestões que possam melhorar o processo.

Para esse estudo foram selecionadas duas pessoas com perfis semelhantes. Ambos são desenvolvedores experientes, graduados pela universidade PUC-Rio, conhecem as metodologias de desenvolvimento de testes TDD e BDD e o padrão UML. Também possuem experiência na codificação de testes de unidade para sistemas reais. A única divergência pertinente ao experimento é que um dos participantes já utilizou uma ferramenta de geração automática de testes. No caso, esse participante usou a ferramenta produzida em [Caldeira, 2010]. Devido a essa oportunidade, em sua entrevista realizamos perguntas que traçam um comparativo entre as ferramentas.

Esse capítulo é dividido em 4 partes: a primeira é uma breve introdução do experimento e perfil dos participantes; a segunda explica o procedimento realizado no experimento; a terceira mostra a análise de cada participante ao utilizar a ferramenta; por fim, a quarta parte, apresentamos as considerações finais.

6.1
Procedimento

O estudo realizado foi conduzido nas dependências do Departamento de Informática da PUC-Rio, com a presença de um único participante a cada vez que o experimento fosse realizado. O estudo consiste em quatro etapas: treinamento

do participante, utilização da ferramenta *Easy* em uma aplicação web, utilização da ferramenta *Easy* em uma aplicação de dispositivos móveis e, por fim, uma entrevista.

A etapa de treinamento foi realizada submetendo os participantes à utilização da ferramenta *Easy* para gerar casos de testes para uma aplicação de exemplo simples. A aplicação escolhida foi a tela de autenticação utilizada nesta dissertação. Inicialmente, apresentamos a ferramenta ao participante, explicando as etapas do processo, o objetivo de cada campo do formulário e fluxo de telas da ferramenta. Os participantes também receberam a interface gráfica da aplicação de exemplo impressa, a descrição do caso de uso a ser passado à ferramenta e um manual. Esse manual explica o padrão de descrição dos passos do fluxo principal e alternativo no formulário da ferramenta, as palavras e expressões utilizadas pelos elementos de interface e o funcionamento do campo *Regra de negócio*para cada elemento de interface. Durante o treinamento, os participantes puderam tirar suas dúvidas com o avaliador. Após a conclusão da execução do processo e não havendo mais dúvidas, foi iniciada a segunda etapa do estudo.

A segunda e terceira etapa são os testes do experimento, onde os participantes utilizaram a ferramenta *Easy* para gerarem casos de testes para duas aplicações fictícias, sem a ajuda de terceiros. As aplicações são as versões web e para dispositivos móvel de uma tela de cadastro de novos usuários de uma rede social. A versão web é inspirada na a tela de cadastro da rede social *Facebook*. A versão de dispositivos móveis foi desenvolvida exclusivamente para esse estudo, não sendo semelhante à versão oferecida pelo *Facebook*. O motivo para isso foi a necessidade de testarmos algumas peculiaridades de aplicações de *smartphones* e *tablets* com os participantes, como a presença de diferentes telas para a realização de uma função.

A interface gráfica da versão web possuía seis elementos: as caixas de entradas de texto *Nome*, *Email* e *Senha*; o *radio buttonSexo,* responsável pela opção sexual do usuário; o botão *Cadastrar-se* e a legenda *Sistema*, responsável pela resposta da aplicação. A diferença entre as versões é a forma como o usuário preenche o campo *Nome*. Na versão de dispositivos móveis, o usuário clica no botão *Inserir Nome* e é levado a tela com cabeçalho *Nome*, onde possui uma caixa de entrada de texto *Nome* e dois botões *Voltar* e *Salvar Nome*. A presença da tela de *Nome* no experimento é fundamental para o surgimento de cenários

problemáticos, os quais os participantes precisaram identificar durante o processo. É importante comentar que no campo *Sexo* da versão de dispositivos móveis, não utilizamos *radio button*, mas sim o *UISegmentedControl*. Assim como na etapa de treinamento, os participantes receberam uma imagem impressa da interface gráfica da aplicação, a descrição do caso de uso e o manual. Também foi registrada a tela do computador utilizado pelos participantes com a ferramenta *Camtasia* [TechSmith, 2002]. Na Figura 56 apresentamos a interface gráfica da aplicação web passada aos participantes. Na Figura 57 expomos o fluxo de telas da aplicação de dispositivos móveis.

Figura 56: Versão web da tela de cadastro de usuário.

Figura 57: Versão para dispositivos móveis da tela de cadastro de usuário.

A última etapa foi a realização de uma entrevista com os participantes. O objetivo foi identificar as vantagens e desvantagens observadas por eles ao utilizar a ferramenta, as dificuldades encontradas durante o processo e sugestões de como melhorá-lo. Para isso, elaboramos um conjunto de perguntas:

- Qual sua opinião sobre utilizar casos de uso para descrever casos de testes?
- Qual sua opinião sobre a abordagem de cenários abstratos e concretos?
- Quais dificuldades você passou ao utilizar a ferramenta?
- Quais são as vantagens de utilizar a ferramenta *Easy?*
- Quais são as desvantagens de utilizar a ferramenta *Easy?*
- Quais novas funcionalidades você gostaria que existissem na ferramenta?

Todas as entrevistas foram registradas com um gravador de voz. Ao término da entrevista, o estudo com o participante foi concluído. É importante comentar que o tempo gasto pelos participantes nos testes não foi computado, devido a existência de somente dois participantes em todo experimento. Para avaliarmos com segurança o custo gerado pela ferramenta na geração de testes, seria necessário um grupo maior de usuários submetidos ao estudo. Outro ponto importante foi a necessidade de os participantes identificarem cenários problemáticos na etapa do experimento responsável pela aplicação de dispositivos móveis. Ao iniciar a etapa, decidimos avisar aos participantes sobre o problema a ser enfrentado e o que deveria ser feito. O objetivo dessa etapa é submeter o participante a uma situação problemática de uso da ferramenta *Easy* e coletar sua opinião sobre o caso. Novamente, os participantes não tiveram suporte do avaliador durante a etapa.

6.2 Análise

Nesta seção apresentamos o que foi observado ao submeter os participantes aos testes com a ferramenta *Easy* e as respostas dadas durante a entrevista. A seção será dividida em duas partes, onde cada uma é responsável pelo o material

coletado de cada participante. Optamos por chamarmos os participantes de P1 e P2. A seção dedicada a cada participante é dividida em duas partes: a primeira aborda o que foi observado pelo avaliador durante a atividade do participante no experimento e a segunda é a entrevista.

6.2.1. P1

6.2.1.1. Atividade

O primeiro participante nunca havia utilizado uma ferramenta de geração automática de teste e não possuía prática com casos de uso. Mesmo assim, seu treinamento foi rápido e sem muitas dificuldades. Inicialmente, o conceito de *valores abstratos* teve certa resistência, porém após ver sua aplicação na ferramenta, o participante conseguiu dominá-lo e aplicá-lo.

Após o treinamento, iniciamos o teste com aplicação web. Percebeu-se logo de início que a falta de prática do participante com casos de uso não o prejudicou ao receber a descrição do caso de uso da aplicação corrente. A descrição do caso de uso utilizada durante o treinamento foi suficiente para o participante obter certo grau de independência. Isso mostrou claramente o quanto casos de uso são de fácil compreensão e orientam o desenvolvedor na produção de casos de testes. O participante conseguiu passar as informações contidas na descrição do caso de uso ao formulário da ferramenta sem problemas. O manual dado ao participante foi constantemente utilizado durante o preenchimento dos campos *Fluxo principal*, *Fluxo alternativo* e *Regra de negócios*, auxiliando-o a seguir o padrão de descrição dos passos dos fluxos, a utilizar as palavras aceitas pelos elementos de interface e os campos adicionados pelos elementos de interface no campo *Regra de negócio*.

Ao concluir o formulário, a ferramenta apresentou os *cenários abstratos*. O participante não dedicou muita atenção à janela, partindo rapidamente à janela de *cenários concretos*. Mesmo com a geração de 108 cenários, o participante acreditava que todos eram cenários possíveis de ocorrerem na aplicação web, por isso não dedicou muito tempo a verificá-los.

A janela de *cenários concretos* foi a etapa da ferramenta que consumiu maior tempo do participante. Devido aos cenários do fluxo principal e alternativo estarem preenchidos, o participante direcionou-se imediatamente ao fluxo sugerido, preenchendo o oráculo dos cenários incompletos. Durante o preenchimento, o participante percebeu um erro nos cenários. Havia cenários que possuíam mais de um passo relacionado a um mesmo elemento de interface, o que é errado. O avaliador sugeriu que o participante voltasse à janela de formulário e verificasse se havia algum erro nas informações passadas. Ao verificar, o participante percebeu que havia preenchido o fluxo alternativo errado, apontando diversos fluxos a um mesmo passo do fluxo principal. O interessante neste caso foi observar a vantagem da funcionalidade de migrar entre as etapas da ferramenta na presença de um erro. Rapidamente o participante pôde corrigir o erro sem precisar reiniciar o processo. Ao término do processo, o participante ficou surpreso com o grande número de *scripts* gerados.

O teste com a aplicação de dispositivos móveis mostrou uma evolução na forma como o participante operava a ferramenta. O participante era mais objetivo ao preencher o formulário, como na descrição dos passos dos fluxos principal e alternativo, que eram escritos com poucas palavras. É importante comentar que antes de iniciar essa etapa do experimento, o participante foi avisado sobre a possibilidade de cenários problemáticos e que ele deveria retirá-los do processo.

Ao gerar os *cenários abstratos*, o participante começou a buscar os cenários problemáticos. Ao total, foram gerados 432 cenários, dos quais somente 108 eram aceitos pela aplicação testada. Nitidamente a busca pelos cenários não desejados foi cansativa ao participante. A janela de *cenários abstratos* foi o que consumiu maior tempo de todo processo. Em seguida, nos *cenários concretos*, o participante novamente focou a atenção no fluxo sugerido. Também foram encontrados cenários que deveriam ter sido removidos na tela de *cenários abstratos*, porém por falta de atenção, o participante manteve-os. Como a janela de *cenários concretos* também possui a funcionalidade de desmarcar cenário, o participante não precisou voltar a etapa anterior para desconsiderá-lo, não precisando gerar novamente os *cenários concretos* e preenchê-los do início. O participante comentou que, devido à existência de cenários problemáticos, a tela de *cenários abstratos* tornava mais simples de identificá-los do que a de *cenários concretos*.

Ao gerar os *scripts*, concluíram-se os testes com o participante corrente, utilizando a ferramenta.

6.2.1.2. Entrevista

A entrevista começa perguntando ao participante sua opinião sobre a utilização de casos de uso para descrever os casos de testes. A resposta foi positiva, afirmando que a descrição de um caso de uso facilita a compreensão do desenvolvedor sobre os tipos de testes a serem gerados.

> *"Minha opinião sobre utilizar casos de uso é de que é bem melhor. Não sou de usar casos de uso para produzir testes, por questão de tempo e gastos. Mas percebi logo de início que casos de uso ajudam bastante para você entender o funcionamento e descobrir possíveis testes a partir dele. Achei realmente interessante."*

Ao ser perguntado se o preenchimento do formulário da ferramenta foi fácil tendo a descrição do caso de uso em mãos, o participante concordou. Em seguida, perguntamos o que achava sobre a abordagem de *cenários abstratos* e *concretos* utilizada na ferramenta. O participante achou interessante, enfatizando que a numeração dos passos que compõem um cenário, a visualização individual de cada cenário e a utilização de linguagem natural restrita são características benéficas ao processo. Por outro lado, criticou a forma como os cenários são organizados e apresentados pela ferramenta ao desenvolvedor.

> *"Achei interessante a visualização individual de cada cenário, que ficam divididos por "caixas". Também achei interessante a identificação do tipo dos passos que compõem um cenário, como S1, A3, 1, pois mostravam o que cada cenário tinha. O uso de linguagem natural também é muito interessante. A organização dos cenários é que poderia ser melhor. A listagem deles é boa, porém se pudéssemos distribuí-los por toda tela, aproveitando mais a tela ao variar a organização, a visualização seria muito melhor."*

A próxima pergunta foi relacionada às dificuldades enfrentadas por ele utilizando a ferramenta. O participante comentou alguns pontos, como a falta de um identificador para cada cenário, legendas nos campos do formulário que ajudariam o desenvolvedor entender o que cada campo é responsável e a

necessidade de se preencher constantemente o oráculo de cenários incompletos. O participante sugeriu a criação de funcionalidades que reaproveitassem os oráculos já conhecidos.

> *"A falta de um identificador para cada cenário, isso foi algo que senti falta. Outra dificuldade foi a falta de legendas ou títulos para os campos da tela de formulários. Alguns momentos tive dúvida sobre qual era a função de alguns campos na tela. Outra foi ao preencher os cenários concretos, poderia ter um "dropdown" com os retornos do sistema já usados. Assim evitaria de ficar preenchendo constantemente os campos de cada cenário. Também poderia sugerir qual retorno deveria ser para um cenário."*

Ao ser perguntado quais as vantagens de utilizar a ferramenta *Easy*, o participante identificou a facilidade de geração de cenários em relação a um caso de uso e de *script* de testes.

> *"Eu me assustei pela quantidade de casos de testes gerados. Se eu fosse gerá-los todos manualmente, com certeza, iria esquecer algum. Não seria trivial gerar todos, pensando em todas as possibilidades de testes. Não estou nem considerando o fato dele gerar os casos de teste codificados, mas a simples identificação dos cenários possíveis já daria muito trabalho. Listar os cenários e codificá-los seria ainda muito trabalhoso."*

A pergunta seguinte foi em relação às desvantagens observadas por ele na ferramenta. Inicialmente, comentou a preocupação de não saber o custo de gerar casos de testes utilizando a ferramenta para uma outra aplicação (o custo no caso seria de cadastrar os elementos de interface da aplicação desejada). Outra desvantagem seria a necessidade da tela de *cenários abstratos* no processo. Disse que em um dos experimentos ela não foi necessária, porém quando houve a necessidade de verificar os cenários, a tela de *cenários abstratos* foi útil. A última observação foi sobre a presença de cenários problemáticos, os quais deveriam ser evitados pela ferramenta.

"A questão do custo de permitir a ferramenta Easy de gerar scripts de testes para outras aplicações. Não sei qual seria o esforço para conseguir gerar testes aos sistemas que trabalho.

Outro ponto era a tela de cenários abstratos. A cronologia das telas (formulários, cenários abstratos e concretos) não foi uma desvantagem, porém, inicialmente, a tela de cenários abstratos não me trazia vantagens, pois todos os cenários seriam utilizados.

Porém, quando percebi que alguns cenários não eram interessantes e deveriam ser eliminados do processo, a tela de cenários abstratos era melhor de identificar cenários do que a de cenários concretos, pois tinha menos poluição visual, sendo assim mais fácil de fazer a eliminação de cenários.

O último, foi a questão de precisar lidar com cenários que não poderiam ser utilizados. Acho que se deveria ter alguma forma inicial de eliminá-los, não havendo a necessidade do desenvolvedor preocupar-se com eles e ter de buscá-los na lista de cenários..."

Por fim, perguntamos quais funcionalidades seriam interessantes que existissem na ferramenta. O participante comentou a necessidade de filtros que evitassem cenários indesejados, uma funcionalidade que reaproveitasse e sugerisse oráculos existentes aos cenários do fluxo sugerido, um identificador nos cenários e melhorias na interface gráfica, como inclusão de legendas nos campos do formulário e a organização dos cenários na tela.

6.2.2. P2

6.2.2.1. Atividade

O segundo participante possui experiência com ferramentas de geração automática de casos de testes e com casos de uso. As dúvidas durante o treinamento foram semelhantes às do participante P1, como a dificuldade inicial de se compreender a função dos *valores abstratos* no processo. Para ser mais específico, a dúvida do participante estava na forma como são utilizados os *valores abstratos* no campo *Regra de negócio*. O participante P2 apresentou maior independência durante o treinamento do que o P1. Acreditamos que tal comportamento ocorreu devido à sua experiência com casos de uso e ferramentas de geração automática de testes.

Ao iniciar o teste com a aplicação web, o participante apresentou facilidade ao preencher o formulário. Semelhante ao P1, o participante corrente utilizou constantemente o manual durante o teste. Durante o preenchimento do fluxo principal, a ferramenta acusou um erro na descrição do passo de resposta do sistema. O participante havia definido errado o tipo de elemento de interface do elemento *Sistema*. Após a correção, o formulário foi concluído sem problemas. Na

janela de *cenários abstratos*, o participante dedicou mais tempo a observar os cenários do que P1. Porém, sua observação não abrangeu todos os cenários e logo encaminhou-se à janela de *cenário concreto*. Novamente, foram gerados 108 cenários.

A janela de cenários concretos foi a que ocupou mais tempo do participante, devido ao preenchimento dos oráculos dos cenários incompletos. O participante deu prioridade ao fluxo sugerido, por ter notado que os cenários do fluxo principal e alternativo já estavam preenchidos. Durante o preenchimento dos cenários, uma solução inusitada ocorreu. Devido à reutilização constante de oráculos conhecidos nos cenários incompletos, o participante optou por abrir a aplicação de bloco de notas do sistema operacional para facilitar a reutilização dos oráculos. Essa solução reforça a ideia de que a ferramenta precisa fornecer ao desenvolvedor uma forma de reaproveitar os oráculos conhecidos, não precisando escrevê-los a cada cenário incompleto.

O teste com a aplicação de dispositivos móveis apresentou resultados semelhantes aos observados com o participante P1, como maior agilidade ao preencher o formulário. Antes de iniciar esse teste, o participante foi informado da possibilidade de cenários problemáticos. O preenchimento do formulário não apresentou problemas. Na janela de cenários abstratos, o participante teve o mesmo reflexo do teste anterior de ir à tela de *cenários concretos* sem observar os *cenários abstratos*. Chegando aos *cenários concretos*, percebeu que seria mais simples selecionar os cenários pertinentes na janela de *cenários abstratos* e retornou. Assim, como no teste com P1, foram gerados 432 cenários, porém somente 108 deveriam permanecer no processo. O participante achou cansativa a forma como era realizada a busca por cenários indesejados. Devido à sua falta de atenção, alguns cenários problemáticos permaneceram e somente foram notados e desmarcados na janela de cenários concretos. O participante utilizou novamente a aplicação de bloco de notas para auxiliá-lo no preenchimento dos cenários. Ao gerar os *scripts*, concluíram-se os testes com o participante corrente, utilizando a ferramenta.

6.2.2.2. Entrevista

Iniciamos a entrevista perguntando ao participante sua opinião sobre a utilização de casos de uso para descrever casos de testes. O participante comentou que é vantajoso utilizá-los devido a serem um método organizado de descrever os testes ao desenvolvedor. Além disso, pertencem ao padrão UML, o que facilita a redução da curva de aprendizagem do desenvolvedor ao utilizar a ferramenta. Também há a vantagem dos casos de uso servirem como documentação da aplicação testada.

> *"É uma boa forma de começar um método de teste, de ser um princípio de método. O grande problema de testar é não seguir um método e esquecer muitas possibilidades [cenários]. Além disso, o caso de uso acaba sendo vantajoso como um meio de documentar a aplicação. Outro ponto é a curva de aprendizado ao utilizar a ferramenta. Pelos casos de uso serem uma abordagem de descrever testes com linguagem natural e um padrão conhecido [UML], torna-se mais fácil operar a ferramenta."*

Em seguida, perguntamos sua opinião sobre a abordagem de *cenários abstratos* e *concretos* na ferramenta. Segundo o participante, são interessantes os dois tipos de cenários, pois os *cenários abstratos* são mais simples de informar o que será realizado no sistema do que os *cenários concretos*. Assim, o desenvolvedor não precisa analisar cenários com valores de entrada que não serão utilizados.

> *"É interessante a distinção [entre os cenários abstratos e concretos], porque muitas vezes você consegue definir logo a forma como o sistema irá se comportar através do cenário abstrato, não precisa valorar o cenário para dizer o que irá acontecer. Assim você consegue "matar" inúmeros casos de teste que não seriam pertinentes. Pode-se dizer que seria inviável se o desenvolvedor tivesse que analisar os scripts gerados para, nesse momento, desconsiderar os testes não pertinentes."*

Ao ser perguntado sobre as dificuldades passadas durante a utilização da ferramenta, o participante comentou que alguns detalhes da interface gráfica o prejudicaram, e a forma como os *valores abstratos* são abordados no campo

Regras de negócio não é clara, principalmente para usuários sem conhecimento de programação.

> *"Inicialmente algumas dificuldades básicas de uso da interface gráfica. Também tive dificuldade com as regras de negócio no contexto da ferramenta, pois a explicação dada foi vaga. Não ficou claro como são tratados os valores abstratos em relação aos fluxos [principal, alternativo e sugerido]. O uso de valores como True e False na Regra de negócio para os valores abstratos é problemático, principalmente à pessoas sem conhecimento de programação, pois não fica claro o que significa um valor abstrato receber True ou False."*

A pergunta seguinte foi relacionada às vantagens geradas ao utilizar a ferramenta *Easy*. O participante apontou inicialmente a agilidade e a confiabilidade dos *scripsts* gerados pela ferramenta, comparadas com o processo manual. Porém, comenta que funcionalidades que eliminam cenários problemáticos ajudariam a agilizar o processo proposto pela *Easy*.

> *"Em relação a escrever manualmente os testes, a Easy é muito mais ágil e confiável. Digo confiável, pois os testes manuais não possuem garantia nenhuma de que estarão corretos [sem erros de programação], pois depende da atenção do desenvolvedor. A utilização de "copy and paste" pode gerar problemas, entre outros coisas. Agora comparando com outra ferramenta de geração automática de teste que eu já usei [Caldeira, 2010], como os grupos condicionais, que você poderia dizer qual estado dos elementos de interface eram pertinentes. Isso reduziu absurdamente o número de testes gerados. Mas eu acredito que se fossem implementados esses filtros, ajudaria muito na operabilidade da Easy."*

Sobre as desvantagens identificadas na utilização da ferramenta *Easy*, o participante apontou a deficiência da organização que os cenários são apresentados ao desenvolvedor. Acredita que o fluxo de telas da ferramenta é um ponto positivo, mas seriam interessantes mecanismos que dividissem os cenários em grupos que possuíssem alguma característica em comum.

> *"A desvantagem está na exposição dos cenários, a forma como é realizada. A organização da interface gráfica [formulário, cenários abstratos e concretos] é interessante. É interessante saber que existe um fluxo de telas. Porém, acho que falta ao usuário conseguir quebrar essas informações em setores. Por exemplo, quando você gera combinações de casos de testes, você identifica grupos pelo o que os testes fazem, como em um grupo tal botão é pressionado e no outro não.*

Essa formação de grupos ou setores seria interessante como uma forma de organizar e filtrar visualmente os cenários na ferramenta. Essa necessidade ficou clara no exemplo da aplicação mobile, pois foi necessário identificar cenários que não poderiam ser gerados."

Ao ser perguntada sobre quais funcionalidades seriam interessantespara ferramenta *Easy* possuir, o participante apontou diversas novas ideias. A primeira é a utilização de *autocomplete* na descrição dos fluxos no formulário da ferramenta. Essa funcionalidade auxiliaria o desenvolvedor a não precisar de um manual ou lembrar do padrão de escrita dos passos e as palavras utilizadas pelos elementos de interface. Outra funcionalidade seria deixar em negrito as principais palavras dos passos de um cenário na tela de *cenários abstratos*. Dessa forma, agilizaria a identificação pelo desenvolvedor do conteúdo dos cenários. Outra funcionalidade para a tela de *cenários abstratos* seria permitir que o desenvolvedor escolhesse os oráculos dos cenários nesta etapa da ferramenta. Segundo o participante, seria uma forma de reduzir o número de valores de entrada passados na janela de *cenários concretos*. Por fim, o participante enfatizou a necessidade de filtros que facilitariam a manipulação dos cenários. Além dos *Grupos condicionais* implementados na ferramenta gerada em [Caldeira, 2010], as telas de cenários poderiam possuir filtros que mostrassem cenários que possuem característicasem comum. Dessa forma, o desenvolvedor facilmente conseguiria selecionar cenários pertinentes aos seus objetivos. Também foram citadas melhorias na interface gráfica da ferramenta.

Por fim, realizamos uma pergunta adicional ao participante P2 devido à sua experiência ao ter utilizado a ferramenta criada em [Caldeira, 2010]. Na pergunta, pedimos que o participante fizesse um comparativo entre as ferramentas, apontando suas principais vantagens. O participante comentou que o principal problema enfrentado por ambas é a explosão combinatória de cenários possíveis. A tabela de decisão tende a ser mais difícil de manusear com grande número de cenários, porém a forma como a ferramenta *Easy* está implementada atualmente também não elimina essa dificuldade. Caso a ferramenta *Easy* tivesse funcionalidades que evitassem cenários problemáticos, está seria a opção mais fácil de utilizar. A forma como cada cenário é apresentado ao desenvolvedor

também é outra vantagem da ferramenta *Easy*, pois facilita a compreensão do desenvolvedor sobre as ações a serem realizadas no sistema.

> *"O problema está na explosão combinatória, de como as duas abordam todas as possibilidades. A tabela de decisão desenvolvida em [Caldeira, 2010] é muito mais fácil de se perder, principalmente se houver um grande número de colunas. A Easy também gera a possibilidade do usuário deixar passar cenários não pertinentes, porém se for implementado as funcionalidades de filtros, as comentadas anteriormente, a ferramenta Easy seria a opção nitidamente mais fácil de ser utilizada.*
>
> *Outra vantagem da Easy é que a visualização de um cenário é mais organizada do que na tabela de decisão, pois os cenários são descritos individualmente, não precisando dos demais para compreender o que um faz. Se por algum motivo eu sair do computador e voltar depois de um tempo, conseguirei facilmente saber onde parei e do que se trata cada cenário. Você consegue visualizar o que cada cenário representa.*
>
> *Acredito que quando a tabela de decisão estiver quase toda preenchida, ela seriamais efetiva, porém há esse problema de se perder quando ela for muito grande."*

6.3 Considerações finais

Podemos afirmar que a realização desse estudo foi fundamental para o amadurecimento da ferramenta *Easy*. Durante os testes, foram observados os pontos positivos e negativos de utilizar a ferramenta, a aparição de dificuldades conhecidas e novos problemas. Também pudemos contar com as sugestões dos participantes para melhoria do processo.

De todas as vantagens identificadas, podemos mencionar três que ambos os participantes comentaram. A primeira delas é a utilização de casos de uso. Ficou clara em todos os testes a facilidade gerada ao desenvolvedor por utilizarmos os casos de uso na descrição dos casos de teste. A segunda vantagem em comum é a forma como os cenários são apresentados na ferramenta. A utilização de linguagem natural restrita e a organização dos cenários de forma independente dos demais ajudaram o desenvolvedor a manipular todos as possibilidades de cenários. A última vantagem a ser comentada é o fluxo de telas da ferramenta.

Ambos os participantes afirmaram que as telas de *cenários abstratos* e *concretos* facilitaram a geração dos casos de testes. Outro ponto positivo do fluxo de telas é a possibilidade de transitar entre as janelas durante o processo, como por exemplo para realizar correção de erros.

As desvantagens apontadas por ambos resumem-se na geração de cenários não desejados pelo desenvolvedor, o reaproveitamento de oráculos conhecidos e melhorias na interface gráfica. Os participantes comentaram diversas vezes a necessidade de mecanismos para evitar a geração de cenários que fossem problemáticos para a aplicação testada, sugerindo a criação de filtros para reduzir e organizar as possibilidades de casos de testes. O reaproveitamento de oráculos conhecidos mostrou-se necessário devido à constante necessidade dos participantes preencherem cenários incompletos com respostas do sistema já cadastradas.

Por fim, acreditamos que o resultado desse estudo foi positivo, pois mostrou que a ferramenta *Easy* é de fácil manuseio pelos desenvolvedores e é capaz de gerar um grande número de casos de testes de forma ágil e segura.

7
Conclusão

O principal objetivo deste trabalho foi avaliar como a geração automática de testes, utilizando máquina de estados e casos de uso, pode reduzir o custo do desenvolvedor na criação e execução de testes, e na identificação de falhas nas aplicações submetidas ao processo proposto.

Para essa pesquisa, foi desenvolvida uma ferramenta, apelidada de *Easy,* responsável pela geração de testes de interface gráfica para aplicações de dispositivos móveis da marca *Apple*, como *iPhone*, *iPad* e *iPod Touch*. A descrição dos testes é realizada pelo desenvolvedor na interface da ferramenta, preenchendo um formulário de casos de uso. Após o preenchimento, a ferramenta apresenta em sua interface gráfica os cenários descritos em linguagem naturalrestrita. O desenvolvedor tem a oportunidade de passar valores de entrada e selecionar os cenários que devem transformar-se em *script* de testes. Os *scripts* gerados são destinados à ferramenta de execução de testes de interface gráfica *UI Automation*. A eficácia do processo proposto e da ferramenta *Easy* foram avaliadas através da geração de casos de testes para uma sistema real e ao comparar com técnicas manuais de geração de testespara o mesmo sistema. Segundo os resultados obtidos, os testes gerados automaticamente utilizando máquina de estados e casos de uso apresentaram as seguintes vantagens:

- Redução do custo do desenvolvedor ao descrever casos de testes utilizando casos de uso.
- Formulação abrangente de suíte de testes.
- Otimização do número de cenários pelo recurso de segmentação de casos de uso.
- Redução do tempo da geração de testes comparado ao processo manual.

Em relação às desvantagens, podemos citar a realização de testes de interfaces gráficas que possuem diversos elementos de interface. Neste caso, o número de cenários pode ser muito grande e dificultar a manipulação destes pelo

desenvolvedor. A interface gráfica das ferramentas de geração de testes, assim como a *Easy*, pode amenizar esse problema, porém não resolvê-lo para grandes quantidades de cenários.

Além das vantagens do processo, podemos expor algumas soluções geradas durante o desenvolvimento da ferramenta *Easy* neste estudo. Ainterface gráfica da ferramenta é um dos motivos responsáveis pela redução do custo do desenvolvedor na geração de testes. Ao apresentar os cenários descritos com linguagem natural restrita, o desenvolvedor pode identificar e manipular estes com mais facilidade. Outro ponto importante é a capacidade de registrar elementos de interface na ferramenta, permitindo ao desenvolvedor gerar testes para diferentes aplicações. Acreditamos que as soluções e dificuldades enfrentadas na implementação da ferramenta *Easy* poderão contribuir para o desenvolvimento de novas ferramentas de geração de testes.

Assim, baseado nos resultados obtidos, acreditamos que o processo proposto neste estudo é uma nova e eficiente alternativa de geração automática de testes para diferentes tipos de sistemas no mercado, principalmente para aplicações destinadas a dispositivos móveis.

8
Trabalhos futuros

Podemos iniciar comentando como trabalho futuro a possibilidade da ferramenta *Easy* gerar *scripts* de testes para outras linguagem de programação. Inicialmente optamos por implementar o módulo responsável por gerar testes para a ferramenta de execução de testes de interface gráfica *UI Automation*. Há o interesse de produzir um módulo para as ferramentas *UI Unit* [Galdino, 2010] que gera testes de interface gráfica para aplicações desenvolvidas com a biblioteca *Qt* [Nokia, 2007]. A inclusão de novas linguagens à ferramenta *Easy* é interessante do ponto de vista de avaliar se a arquitetura da ferramenta é capaz de lidar com diferentes linguagens, elementos de interface e aplicações. Importante lembrar que a ferramenta *Easy* não foi desenvolvida exclusivamente para aplicações de dispositivos móveis. Acredita-se que ela seja capaz de gerar casos de testes para outros tipos de sistemas, como aplicações web e de computadores pessoais.

Um estudo interessante a ser realizado é a comparação do processo proposto neste documento com outros existentes de geração automática de testes, como tabela de decisão e *capture and replay*, avaliando quais os pontos fortes e fracos de cada processo e qual oferece menor custo ao desenvolvedor na geração de casos de testes.

Diversas funcionalidades aumentariam a eficiência da ferramenta na geração de testes. Uma delas é a implementação de grupos condicionais na janela de formulários de casos de uso. Esse mecanismo evitaria de cenários não aceitos por uma aplicação possam ser gerados. Somente quando elementos de interface pertencentes a um grupo condicional satisfazem condições definidas pelo desenvolvedor, os cenários seriamcriados. Outra funcionalidade interessante é a utilização do operador lógico *SE* e *SE NÃO* no fluxo principal do caso de uso. Dessa forma, o usuário poderá criar casos de testes mais complexos ao permitir diferentes caminhos a serem tomados.

Outra funcionalidade seria permitir a utilização de diferentes elementos de interface responsáveis pelos oráculos dos testes. No capítulo *Prova de conceito*,

comentamos que as aplicações de dispositivos móveis tendem a utilizar diversas telas para uma única funcionalidade. Assim, não há um único elemento de interface encarregado de informar a situação atual da aplicação ao usuário. Seria interessante permitir ao desenvolvedor selecionar quais elementos de interface serão responsáveis pelos oráculosdos cenários, semelhante ao feito na tela de *cenários concretos* da ferramenta, onde o desenvolvedor pode passar valores de entrada aos cenários.

Por fim, durante a geração dos *scripts* de testes, percebeu-se a necessidade de gerar cenários semelhantes com valores de entrada diferentes. Em outras palavras, permitir que um *cenário abstrato* gere mais de um *cenário concreto*. Esses *cenários concretos* testariam uma situação semelhante, porém possuiriam valores de entradas diferentes. A versão atual da ferramenta *Easy* permite somente transformar um *cenário abstrato* em um único *cenário concreto.* Essa funcionalidade tornará a ferramenta mais flexível às vontades do desenvolvedor.

9
Referências bibliográficas

Apfelbaum, L. (1995) “Automated functional test generation.” in Autotestcon ’95 Conference. IEEE, 1995.

Apple (2007). “UIKit Framework Reference”. Disponível em: http://developer.apple.com/library/ios/#documentation/uikit/reference/UIKit_Framework/_index.html

Apple (2007). “Developer for iOS”. Disponível em: https://developer.apple.com/technologies/ios/

Araujo, T.; Staa, A. (2009) “Um Método Baseado em Comportamento com Foco no Desenvolvimento de Aplicações Baseadas em Interfaces Gráficas”. Monografias em Ciência da Computação, Departamento de Informática, PUC-RIO.

Astels, D. (2006) “rSpec Quick Reference”. Disponível em: http://blog.daveastels.com/files/QuickRef.pdf

Beck, K. (2002) “Test Driven Development: By Example” Addison-Wesley Professional, 2002

Belli, F. (2001) “Finite-state testing and analysis of graphical user interfaces,” ISSRE. IEEE Computer Society, 2001, pp. 34–43.

Barnett, M.; Grieskamp, W.; Nachmanson, L.; Schulte, W.; Tillmann, N.; Veanes, M. (2003) “Towards a tool environment for model-based testing with AsmL.” in FATES, 2003, pp. 252–266.

Berris, D. (2009) “A Behavior Driven Development (BDD) inspired library for specifications/testing in C++”. Disponível em: https://github.com/mikhailberis/spec-c--

Brady, P.; Swicegood, T. (2008) “PHPSpec PEAR Channel”. Disponível em: http:// pear.phpspec.org/index.php

Boehm, B. W.; Basili, V. R. (2001) “Software Defect Reduction Top 10 List” IEEEComputer, Los Alamitos, v. 34, n. 1, p. 135-137, jan. 2001

Caldeira, L.R.N. (2010) “Geração de Massas de Teste para Aplicações WEB a Partir da Composição de Casos de Uso e Tabelas de Decisão” Dissertação de Mestrado, DI/PUCRio; 2010.

de Souza, C. S.; Leite, J. C.; Prates, R.O. & Barbosa, S.D.J. (1999) "Projeto de Interfaces de Usuário: Perspectivas Cognitiva e Semiótica". Anais da Jornada de Atualização em Informática, XIX Congresso da Sociedade Brasileira de Computação, Rio de Janeiro.

Cockburn, A. (2000) “Writing Effective Use Cases”. Addison-Wesley, vol. 1.

Ding, A. “Kiwi”. (2011) Disponível em: http://www.kiwi-lib.info/docs.html

Farchi, E.; Hartman, A.; Pinter, S. S. “Using a model-based test generator to test for standard conformance.” (2002) IBM Systems Journal, vol. 41, no. 1, pp. 89–110, 2002.

Ferreira, E.; Albuquerque, J. “IntensityAlarm – A New Waking Mood” (2013) Disponível em:https://itunes.apple.com/sa/app/intensity-alarm/id623694364?mt=8.

Fischer, G. (1998) “Beyond ‘Couch Potatoes’: From Consumers to Designers”. In Proceedings of the 5thAsia Pacific Computer-Human Interaction Conference. IEEE Computer Society. pp.2-9.

Galdino, C. (2010) “Ferramenta de auxílio à automação de testes de interfaces gráficas de- senvolvidas com C++” Relatório de Iniciação Cientifica, DI/PUC-Rio

Hong, H. S.; Kwon, Y. R.; Cha, S. D. (1995) “Testing of object-oriented programs based on finite state machines.” APSEC. IEEE Computer Society, 1995, pp. 234.

Janzen, D. S.; Saiedian H. (2008) “Does Test-Driven Development Really Improve Software Design Quality?”IEEE Software; March/April 2008; pp 77-84

Lachtermacher, L. (2009) “Automação dos testes a partir de tabelas de decisão” Rio de Janeiro, RJ. 2009. Dissertação de Mestrado - Departamento de Informática, PontificiaUniversidade Católica (PUC-Rio).

Labs, P. (2011) “Cedar”. Disponível em: https://github.com/pivotal/cedar

Lindgaard. G. (1994) “Usability Testing and System Evaluation”. London, UK: Chapman & Hall.

Lucio, L.; Pedro, L.; Buchs, D. (2004) “A methodology and a framework for model-based testing.” in RISE, ser. Lecture Notes in Computer Science, N. Guelfi, Ed., vol. 3475. Springer, 2004, pp. 57–70.

Memon, A. (2002) “GUI Testing: Pitfalls and Process” Computer, v.35, n.8, p. 87-88, ago. 2002

Moran, T. (1981) “The Command Language Grammars: a representation for the user interface of interactive computer systems”. International Journal of Man-Machine Studies,15, 3-50.

Nist (2009) “The Economic Impacts of Inadequate Infrastructure for Software Testing” National Institute of Standards & Tecnology, Maio de 2009.

Nokia. (2010). “Qt - Cross-platform application and UI framework”. Disponível em: http://qt.nokia.com/

Norman, D. (1986) “Cognitive Engineering”. In D. Norman & S. Draper (eds.) User Centered System Design. Hillsdale, NJ. Lawrence Erlbaum. pp.31-61.

Norman, D. (1988) “Psychology of Everyday Things”, Basic Books, Harper Collins Publishers.

Norman, D. (1991) “Cognitive Artifacts”. In Carroll (ed.) Designing Interaction: Psychology ate the Human-Computer Interface. pp.17-38.

Norman, D. (1993) “Things that make us smart: defending human attributes in the age of the machine”. Reading, MA: Addison-Wesley

North, D. (2003) Introducing BDD. Disponível em: http://dannorth.net/introducing-bdd

Perry D. E.; Evangelist W. M. (1987) “An Empirical Study of Software Interface Faults” Pro- ceedings of the International Symposium on New Directions in Computing, IEEE CS, Trondheim Norway, v. 2, p. 32-38, jan. 1987.

Preece, J.; Rogers, Y.; Sharp, E.; Benyon, D.; Holland, S.; Carey, T. (1994) “Human-Computer Interaction”. Addison-Wesley

Staa, A. (2010) Automação dos testes. Disponível em: http://www.inf.puc-rio.br/~inf2134/docs/ INF2134_Modulo14_AutomacaoDosTestes.pdf

Staa, A. (2013) Qualidade de Software Conceitos. Disponível em: http://www.inf.puc-rio.br/~inf1413/docs/INF1413_Aula02_QualidadeConceitos.pdf

Staa, A. (2013) Máquina de estado. Disponível em: http://www.inf.puc-rio.br/~inf1413/docs/INF1413_Aula13_MaquinaEstados.pdf

Staa, A. (2013) Especificações, Resumo. Disponível em: http://www.inf.puc-rio.br/~inf1413/docs/INF1413_Aula04_Especificacao.pdf

TechSmith (2002) “TechSmith – Camtasia” Disponível em: http://www.techsmith.com/

Whittaker, J. A. (1997) “Stochastic software testing.” Ann. Software Eng., vol. 4, pp. 115–131, 1997.

White, L.; Almezen, H. (2000) “Generating test cases for GUI responsibilities using complete interaction sequences,” in ISSRE ’00: Proceedings of the 11th International Symposium on Soft ware Reliability Engineering. Washington, DC, USA: IEEE Computer Society, 2000, p. 110.