UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Karina Karimova

Enhancement of iOS Application Accessibility: Automation of Testing and Guidelines

Master's Thesis (30 ECTS)

Supervisor(s):
Ishaya Peni Gambo, PhD

Tartu 2023

Enhancement of iOS Application Accessibility: Automation of Testing and Guidelines

Abstract:
Mobile application accessibility is an essential part of user interface and experience, that ensures all individuals, regardless of their disabilities such as blindness, low vision, or color blindness, can access and use the functionalities of the digital product successfully. Considering the widespread use of smartphones and the fact that they have become an indispensable part of daily routines, it is crucial to enhance the accessibility level of mobile applications.

The primary objective of this study was to contribute to the development of more inclusive mobile apps, specifically those based on the iOS platform offered by Apple. There is a lack of cost-free automated tools that can help identify accessibility issues of mobile applications. Thus, an automated testing tool, named Open Accessibility Tool (OAT), with accessibility faults detection and descriptive error reports has been developed.

The OAT tool is based on a preliminary in-depth exploration of the accessibility guidelines required by European Union regulations, generally accepted industry standards, and Apple's UI design principles. Guidelines and their Success Criteria are assessed regarding their capability for automated testing on iOS. Based on these prerequisites, the developed tool can help applications to ensure compliance with accessibility criteria.

Considering the significance of this subject and its dynamic nature, OAT was developed as an open-source library, offering convenient options for expanding its functionalities by using modulated architecture and a modern technical stack. Tool quality is ensured by an extensive code coverage encompassing all prevailing accessibility checks.

The OAT tool was checked on sample projects. Additionally, a comparative analysis has been conducted against the solution available in the market. The results showcase OAT's superiority in terms of the quantity of identified issues over the competing free-of-cost alternative.

Keywords:
Accessibility, automated accessibility testing, iOS, mobile applications, accessibility guidelines

CERCS: P170 Computer science, numerical analysis, systems, control

iOS rakenduste ligipääsetavuse täiustamine: testimise ja juhiste automatiseerimine

Lühikokkuvõte:

Mobiilirakenduste ligipääsetavus on kasutajaliidese ja kogemuse oluline osa, mis tagab, et kõik inimesed, olenemata nende puuetest, nagu pimedus, vaegnägemine või värvipimedus, pääsevad edukalt digitaalse toote funktsioonidele ligi ja saavad neid kasutada. Arvestades nutitelefonide laialdast kasutust ja seda, et need on muutunud igapäevase rutiini lahutamatuks osaks, on oluline tõsta mobiilirakenduste ligipääsetavuse taset.

Selle uuringu esmane eesmärk on aidata kaasa kaasavamate mobiilirakenduste, eelkõige Apple'i pakutava iOS platvormil põhinevate, arendamisele. Hetkel on puudus tasuta automatiseerimise tööriiste, mis aitaksid tuvastada mobiilirakenduste ligipääsetavuse probleeme. Seega on käesoleva lõputöö raames välja töötatud automaatne testimistööriist nimega Open Accessibility Tool (OAT), millega on võimalik tuvastada ligipääsetavuse vigu ja genereerida veaaruandeid.

OAT-tööriist põhineb Euroopa valitsuse määrustes, üldtunnustatud standardites ja Apple'i kasutajaliidese kujundamise põhimõtetes nõutavate ligipääsetavuse juhiste esialgsel põhjalikul uurimisel. Juhiseid ja nende edukriteeriume hinnatakse vastavalt nende võimele toetada iOS rakenduste automaattestimist. Nendest eeldustest lähtuvalt arendati välja tööriist, mis aitab rakendustel tagada ligipääsetavust vastavalt kriteeriumitele.

Arvestades selle teema olulisust ja selle dünaamilist olemust, arendati OAT välja kui avatud lähtekoodiga teek, mis pakub mugavaid võimalusi oma funktsionaalsuse laiendamiseks moduleeritud arhitektuuri ja kaasaegse tehnilise pinu abil. Tööriista kvaliteedi tagab ulatuslik koodikatvus, mis hõlmab kõiki kehtivaid ligipääsetavuse kontrolle.

OAT-tööriista kontrolliti näidisprojektides. Lisaks on turul pakutavate lahenduste kohta tehtud võrdlev analüüs. Tulemused näitavad OAT-tööriista paremust tuvastatud probleemide arvu osas võrreldes konkureeriva tasuta alternatiiviga.

Võtmesõnad:
Ligipääsetavus, automaatne ligipääsetavuse testimine, iOS, mobiilirakendused, ligipääsetavuse juhised

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

List of Figures

List of Tables

1 Introduction

The increasing popularity of smartphones has been leading to a surge in the development of mobile applications. However, despite the importance of designing for accessibility in digital products and services, accessibility remains an area that is often overlooked in software engineering. This is particularly concerning given the large number of individuals who rely on mobile devices to access information and perform daily tasks [18]. Today an approximated 1.3 billion people experience disability, which is 16% of the world's population, or 1 in 6 of us [12].

In Europe, the European Accessibility Act (Directive 2019/882) is a landmark EU law that requires everyday products and services to be accessible for persons with disabilities [16]. The directive will apply from 28 June 2025. As of that date, companies must ensure that the newly marketed products and services covered by the Act are accessible. The standard EN 301 549 for Information and Communications Technology (ICT) products and services supports the legislation mentioned above [4]. The European digital accessibility standard adopts the Web Content Accessibility Guidelines (WCAG) from the World Wide Web Consortium (W3C) as it provides comprehensive accessibility recommendations for web and non-web technologies [17]. It also follows WCAG four principles: Perceivable, Operable, Understandable, Robust, and three Levels of conformance: A, AA and AAA. Mobile applications accessibility requirements are stated in clause 11 and directly reference 44 WCAG Success Criteria. Within those Success Criteria, only A and AA Levels are required, because AAA Level is not recommended by W3C to be mandatory as a general policy, since it is not possible to satisfy fully for some content [1].

The most significant role in an accessibility topic on the mobile application market is played by Apple and its iOS operating system. It sets the industry standards and trends subsequently adopted by competitors. However, iOS remains a relatively understudied subject within the area of scientific research, receiving less attention compared to its Android counterpart.

Besides the law, in iOS development it is necessary to follow Human Interface Guidelines provided by Apple [6]. This guideline includes the chapter on accessibility as well. By following The Accessibility Law and Apple Inc guidelines, this study aims to ensure that required accessibility features are incorporated into the development process of iOS mobile applications with the help of an automated accessibility testing tool, named Open Accessibility Tool (OAT).

[1]https://bit.ly/3qsGpif

1.1 Problem statement

Accessibility testing is an expensive and challenging procedure. Manual testing requires dedicated employees with expertise in the guidelines, standards set by the law, and knowledge of assistive technologies and disabilities. Another approach is user testing, which by involving people with disabilities adds to the expenses because of recruiting and compensating participants. Automated tests can be written during development process by using platform-specific testing tools; however, it is time-consuming and requires developers with expertise in the accessibility topic. Even when developers are aware of these accessibility needs, the lack of tool support makes the development and assessment of accessible apps challenging [3]. Existing automated accessibility testing tools are either paid or require a lot of additional manual checks. Although free testing tools are also available, they are usually unsupported, have limited accessibility checks, low ratings and quality, or use outdated technologies. Overall, all testing approaches require time and money investments.

This research aims to create an automated accessibility testing tool, which is open-source, written with the latest updates for iOS platforms, and free-of-costs. In order to accomplish the goal, the following Research Questions are formulated to investigate the accessibility requirements, existing testing tools, and potential advantages of the OAT tool.

RQ1: What are the required accessibility Success Criteria for mobile applications? How each criterion can be tested on iOS?

RQ2: What are the existing accessibility testing tools and what are their capabilities?

RQ3: What are the advantages of the new testing tool offered by this study over existing tools? What is the quality assurance of the tool?

1.2 Research goal

The goal of the research is to enhance the accessibility level of iOS applications by providing an automated testing tool, which can indicate all in-depth research-based important accessibility issues in the application and highlight proposed solutions to improve ones. This approach allows to make accessibility checks built-in in the source code almost seamless and therefore makes testing and supporting accessibility effortless, which leads to ensuring that iOS mobile applications are accessible to all individuals, regardless of their disabilities, and enables them to fully participate in today's digital world.

1.3 Thesis outline

Chapter 2 presents the research background and shows other existing approaches. Chapter 3 describes the methodology and approach used in answering the research questions. Chapter 4 presents the study results based on the methodologies used. Chapter 5 presents a discussion of the results. Lastly, the threats to validity, conclusions, and future work are presented in Chapter 6 and Chapter 7 accordingly.

2 Background

2.1 Related work

One of the very first accessibility guidelines for digital products provided by the World Wide Web Consortium (W3C) initially were written for web only. Background to web accessibility guidelines with their regulations were clearly explained by Juha-Pekka Jokinen in his research towards implementing accessibility to the web application [9]. Guidelines were explained with the concrete requirements formulation. The research gave the initial understanding of where to search for official laws and policies that were subsequently referenced in this study.

There are several related works done by the researchers that investigate accessibility of mobile applications based on Android OS and offer automated accessibility testing tools. One such example is a tool named MATE [3], which provides checks for several types of visual impairment and motor skill issues and implements efficiency optimizations tailored towards the use for accessibility testing. Authors of MATE tool conducted a survey on 73 open-source Android projects to indicate MATE tool's advantages over existing accessibility testing tools [3].

Another example of related work done by other researchers [11], where they offered a plugin for extending Figma to analyze how automated accessibility evaluation is accomplished in the design stage. They systematized WCAG 2.1 techniques in order to reduce the time and budget required to detect accessibility issues in the early application stages of the application development process.

There is also research where an automatic evaluation tool of mobile accessibility for Android applications [13] was developed. Eunju Park et al. designed an accessibility evaluation tool that checks mobile apps for conformance with accessibility standards and developed a prototype of the evaluation tool. The proposed tool verifies the existence (or nonexistence) of alternative text for non-text content, which is an important accessibility factor for visually impaired users. It discovers missing alternative text by checking the `contentDescription` attribute of the `ImageView` element in Android XML files.

2.2 Gap analysis

Much theoretical and practical work was done in accessibility research area. However, most of them are for mobile applications based on Android OS and analyze only few guidelines and mostly for visually impaired people. By analyzing the required accessibility guidelines Success Criteria and their possible automation on iOS, we will be able to broaden the scope of the research area of digital product accessibility specifically for devices that use iOS as their operating system.

3 Methodology

This section covers the methodology and approach used in answering the research questions.

3.1 Approach to Answering Research Questions

3.1.1 Approach for RQ1

First, to answer the **RQ1**, we will conduct research to identify regulations related to accessibility requirements for mobile applications. This review will include an examination of required WCAG guidelines [17] that are referenced by EN 301 549 document [4] that supports the European Accessibility Act.

Next, the identified list of required Success Criteria outlined in the guidelines will be assessed to establish the potential for automating the evaluation of each Criterion by investigating its possible implementation on iOS. This process will involve an examination of official iOS developer documentation and best practices from Human Interface guidelines for accessibility [6] provided by Apple. Any potential challenges or limitations for automated testing will be identified.

Overall, this methodology will provide a comprehensive analysis of WCAG guidelines, and a comparison with Apple Inc developer documentation which will be synthesized and applied to the practical part of this study. The result will be as a basis for accessibility checks that will be implemented by OAT in **RQ3**.

3.1.2 Approach for RQ2

The preliminary research about existing accessibility tools showed that they are limited in number and can be grouped into three types [10]: automated testing tools, auditing/inspection tools and linting tools Table 1 [1].

Linting tools conduct static analysis which limits it to inspection of only one or three rules, such as `accessibiltyLabel`. That is because their main purpose of use is formatting source code for programmatic and stylistic errors [15], while accessibility checks are side effects and have a different level of complexity [1].

Auditing and Inspection tools are specifically dedicated for accessibility testing and discover a significantly larger number of accessibility issues. For example, Apple's integrated development environment tool Xcode has Accessibility Inspector that scans the view by going through elements as Voice Over would do and allows to operate specific activities of iOS applications. However, it requires manual work by means of managing navigation and processes of the testing [8].

Automated testing tools are mostly coming as extensions to UI tests. UI tests are usually fragile and overlooked by projects. Specifically, this research will investigate

Table 1. Accessibility testing tools

Type	Tools
Linting Tools	XibLint (free) Swiftlint (free)
Auditing & Inspection Tools	Accessibility Inspector (free) Accessibility Menu (free) Evinced Flow Analyzer (free) Reveal ($)
Automated Testing Tools	Evinced SDK ($) Deque SDK ($) A11yUITests (free) XCUITest (free) GTXiLib (free)

testing tools that use Unit Testing approach and execute the accessibility checks along with them. Since such tools are mostly paid, to answer **RQ2** the study investigates the practical implementation of GTXiLib. This is primarily due to the fact that GTXiLib [5] is the sole freely available option.

Next, we will evaluate the selected testing tool in order to understand the level of maintainability and flexibility, as well as the possibility to extend or contribute to it. It will be implemented by reading available documentation and evaluating the technical stack used to build it.

To test the GTXiLib in practice, we will integrate it into a sample project. The integration process will be assessed as well. Checks provided by the tool will be launched on example views or their elements.

3.1.3 Approach for RQ3

The outcomes derived from the analysis of required Success Criteria and examination of the existing tool (GTXiLib), from answering **RQ1** and **RQ2** respectively, will serve as a base and preconditions to offer a new optimized solution - the automated accessibility testing tool OAT [2]. This tool will be implemented with a modern technical stack, a self-testable approach, and a modulate architecture with a possibility to be easily extended and adapted to any possible accessibility changes. Additionally, by taking into consideration recommendations from Dias et al. [2] in research on improving future automated accessibility tools, the tool will provide informative reports with suggestions for improvements.

OAT quality assurance approach is to write tests for itself. Each type of accessibility

[2]https://github.com/KarimovaKarina/OAT

fault that the tool is going to detect will be validated by writing tests and checking if the fault is detected correctly on special mock UI elements. Testing coverage will check both cases, positive and negative scenarios.

For example, we will set up a UI element that has intentionally violated the accessibility requirement which is currently tested. By running OAT checks on the element we assert that the checks have returned specific issues and the result about the element is not accessible. Along with negative tests, positive tests will be written as well. The UI element with required accessibility features will be set and we will assert that the element is accessible.

```
XCTAssertFalse(notAccessibleElement.isAccessible())
XCTAssertTrue(accessibleElement.isAccessible())
```

Next, the OAT tool will be validated on the same sample project which was used in **RQ2** in order to show how it works in practice. Following this, the comparison of OAT with GTXiLib will be implemented and outcomes will be depicted visually based on the number of detected issues.

3.1.4 Sample projects overview

FinanceFuel [3] is a sample project that we will use in this study for the purpose of conducting evaluations using automated accessibility testing tools. The source code was forked from the origin repository of the project for the convenience of testing processes. FinanceFuel contains different types of UI elements which can be evaluated for compliance with accessibility criteria. The project is currently undergoing development, and it is considered a best practice to integrate accessibility assessments in the early stages. FinanceFuel adopts the Moonlight architecture, inspired by The Elm Architecture's concepts [4]. This architecture has the following key advantages which are useful during testing process:

- business and presentation logic are strongly separated;
- most of logic is pure functions;
- provides simplified testing of each layer.

Another sample project is Expense-Tracker. It is an open-source iOS application for managing expenses from daily income [5]. The app was chosen because it has a programmatic UI setup and no storyboards, which makes it possible to test views with automated testing tools.

[3]https://github.com/KarimovaKarina/Thesis
[4]https://github.com/mooncascade/MoonKit
[5]https://github.com/abdorizak/Expense-Tracker-App

4 Results

This chapter presents the study results based on the methodologies used.

4.1 Answering RQ1

There are four principles of WCAG 2.2: Perceivable, Operable, Understandable and Robust. Each principle has Success Criteria with a description written on the official website [17].

4.1.1 Perceivable Principle

The first principle is Perceivable, which means that information and UI components must be presented in a way that users can perceive them. The content of the website or application must be presented in different formats such as text, images, videos, and sounds to accommodate different user needs. All four guidelines are presented with Success Criteria and their Level on Table 3.

Success Criterion 1.1.1 Non-text content. The Criterion ensures that non-text content, such as images, have the alternative presented to users via speech output unless the non-text content is pure decoration or is used only for visual formatting [6]. On iOS, `UIAccessibility` protocol provides accessibility features for UI elements. Basic `UIKit` controls and views implement this protocol by default. But assistive technologies get information about UI element if it has `isAccessibilityElement` property set to `true`. Only controls in UIKit have `isAccessibilityElement` default value as true. In this case it is possible to check elements that subclass `UIView`, except `UIControl`, only if they set as accessible and show failure on automated testing, with possibility to silent the error case for non-essential elements. Possible error cases for `isAccessibilityElement == true` condition are presented in further analysis by checking basic accessibility properties: `accessibilityLabel, accessibilityValue, accessibilityHint` and `accessibilityTraits`. (SC 1.3.3 Sensory characteristics and SC 4.1.2 Name, role, value)

Success Criterion 1.2.1 Audio-only and video-only (pre-recorded). The intent of Criterion is to ensure that if a UI element provides pre-recorded auditory or video information, it has a text-based alternative, for example, a transcript. Except for the cases if audio or video is a media alternative for text. This can be validated only manually by a quality assurance representative.

Success Criterion 1.2.2 Captions (pre-recorded). Captions or subtitles (as it is called in some countries), shall provide synchronized visual and text alternative for both speech and non-speech audio information needed to understand the media content where

[6]https://bit.ly/non-text-content

non-speech information includes sound effects, music, laughter, speaker identification and location [4]. On iOS, `AVFoundation` framework among a wide range of tasks allows capturing and processing audiovisual media. Testing a mobile application content for conforming to this criterion requires a human being to check it visually [7][7].

Success Criterion 1.2.3 Audio description or media alternative (pre-recorded). The Criterion ensures that video content, which has parts that are not audible, also has an audio description or text alternative. An alternative for time-based media should provide a running description of all that is going on in the synchronized media content. For instance, in a movie when actors stop their dialogue, audio or text description explains the actions and expressions of actors, and any other visual material during that pause. The testing process of the video content for following this Criterion is possible only by a real person and cannot be automated.

Success Criterion 1.2.4 Captions (live). This Success Criterion ensures that if the video has sound, real-time captions are presented. On iOS, AVFoundation framework supports the required functionality starting from iOS 16 and in Beta **??**. The testing method is the same as in SC 1.2.2.

Success Criterion 1.2.5 Audio description (pre-recorded). The Criterion requires that an audio description is provided for all prerecorded video content in synchronized media. This requirement can be met as part of Success Criteria 1.2.3

Success Criterion 1.3.1 Info and relationships. The Criterion requires related UI elements to be grouped to help assistive technology users to understand the relationship between them. For instance, the visual formatting of list items is that they are preceded by a bullet and perhaps indented, or items that share a common characteristic are organized into tabular rows and columns. On iOS, such formatting can be achieved by gathering related elements in one view, which would have a descriptive label or header and subviews would be grouped by setting `shouldGroupAccessibilityChildren` property on that view to `true`. However, the testing process for this Criterion can be implemented only manually, because the correct relationship as it was intended by an author (designers, clients etc) requires an understanding of the mental model and perception of the screen that is possible to test only by a human being.

Success Criterion 1.3.2 Meaningful Sequence. The Criterion ensures that in a case when the sequence in which content is presented affects its meaning, a correct reading sequence is determined programmatically. On iOS, `UIAccessibilityContainer` protocol provides necessary methods for UIView subclasses that serve mainly as container to make subcomponents accessible as separate elements. The difference between `UIAccessibilityContainer` and the approach offered in previous SC 1.3.1 is that `UIAccessibilityContainer` cares about subviews order. The elements can be gathered in one property of container `accessibilityElements` as an array and `isAccessibilityElement` property should be set to `false`, otherwise assistive tech-

[7]apple.co/4515nV4

nologies will ignore subviews. A real person is required to check if VoiceOver reads all UI elements in the correct order and none of the elements were missed to be added to the container.

Success Criterion 1.3.3 Sensory characteristics. This Success Criterion requires that additional instructions are provided to understand and operate content that relies on sensory characteristics, for example, shape, size, location, orientation or sound [8] [17].

On iOS, instructions can be conveyed via `accessibilityHint` property, which should contain a brief description of the result of performing an action on the accessibility element. Since `accessibilityHint` should be set only if the results of an action are not obvious from the element's label, checks for its string can be implemented when it exists. By following Apple Inc guidelines for creating hints [9] here are `accessibilityHint` possible errors under the condition `accessibilityHint != nil`:

1. `accessibilityHint` should not be empty string
2. `accessibilityHint` should begin with a capitalized word
3. `accessibilityHint` should end with a period
4. `accessibilityHint` should not repeat label
5. `accessibilityHint` should not contain stop words (the type of the control or view)
6. `accessibilityHint` should be localized

Success Criterion 1.3.4 Orientation. The Criterion requires content view and operation be presented by user-preferred display orientation, such as portrait or landscape, unless a specific display orientation is essential [10] [17]. This criterion can be tested only manually, as the automation tool cannot know if the view is needed to be forced to a certain orientation. Furthermore, it is not able to detect if the layout has become broken after orientation changes or should be specific for this case.

Success Criterion 1.3.5 Identify input purpose. All input fields should have a clear purpose [11] [17]. On iOS, input forms are represented by classes that conform to `UITextInput` protocol, for example, `UITextField` or `UITextView`. In order to satisfy criterion text input element should be set up with appropriate content type and its keyboard type by using `textContentType` [12] and `keyboardType` [13] properties respectively.

[8]https://www.w3.org/WAI/WCAG21/Understanding/sensory-characteristics
[9]https://apple.co/3QC2ghQ
[10]https://www.w3.org/WAI/WCAG21/Understanding/orientation
[11]https://www.w3.org/WAI/WCAG21/Understanding/identify-input-purpose
[12]https://developer.apple.com/documentation/uikit/uitextcontenttype
[13]https://developer.apple.com/documentation/uikit/uikeyboardtype

It is possible partially automate the testing for this criterion. As `textContentType` is an optional property and provides a limited number of types, while `keyboardType` has a default value, it is possible to test the obvious cases. For instance, `emailAddress` content type requires `emailAddress` type of keyboard.

The following `UITextInput` error case can be shown during automated testing:

- `keyboardType` should be set to `UIKeyboardType.self` for defined `textContentType` `UITextContentType.self`

Success Criterion 1.4.1 Use of colour. The Criterion ensures that color is not used as the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element [14] [17]. For example, in an input form, the required text field is in red color or the error is shown in red. Manual testing is required for checking the user interface for such accessibility faults if color has a meaning assigned to elements.

Success Criterion 1.4.2 Audio control. If any audio content plays automatically for more than 3 seconds, it should have the possibility to be paused or stopped. This criterion can be tested with UI Testing, where actions on video content can be set with a timer for 3 seconds and check if the required buttons exist or not.

Success Criterion 1.4.3 Contrast (minimum). This Success Criterion requires the visual presentation of text and images of text to have a contrast ratio between the text colour and background colour of at least 4.5:1, with an exception for the following [15] [17]:

- Large Text: Large-scale text and images of large-scale text have a contrast ratio of at least 3:1;
- Incidental: Text or images of text that are part of an inactive user interface component, that are pure decoration, that are not visible to anyone, or that are part of a picture that contains significant other visual content, have no contrast requirement.
- Logotypes: Text that is part of a logo or brand name has no contrast requirement.

Conformance to this Criterion can be partially tested, mainly on text content. Contrast ratio calculation [16] is:

$$(L1 + 0.05)/(L2 + 0.05)$$

,

where $L1$ is the relative luminance of the lighter of the colors, and $L2$ is the relative luminance of the darker of the colors. Label background color and its text color can be

[14]https://www.w3.org/WAI/WCAG21/Understanding/use-of-color
[15]https://www.w3.org/WAI/WCAG21/Understanding/contrast-minimum
[16]https://www.w3.org/TR/2016/NOTE-WCAG20-TECHS-20161007/G18.html

compared by that formula. The following errors can be shown during automated testing if contrast is not conforming to Criterion:

- Contrast ratio should be at least 4.5:1 for small text (below 18 point regular or 14 point bold)
- Contrast ratio should be at least 3.0:1 for large text (18 point and above regular or 14 point and above bold)

Success Criterion 1.4.4 Resize text. This Criterion requires text to be resized without assistive technology up to 200 percent without loss of content or functionality. The requirement is not applied to captions and images of text. However, not every content part should be resizable, otherwise main parts can be lost, unless a special design is provided for the case when all UI elements are in the largest or smallest scales. Although visual and semantic testing is required by a real person, part of preventing the loss of content can be automated. For example, it is possible to prevent a label's text cutting by setting the property of label `numberOfLines = 0`. For the case when truncating is intended it would be possible to silence the error in the testing process.

Here is `UILabel` possible errors:

- numberOfLines property should be set to 0

and separately for `UIButton`:

- `titleLabel?.numberOfLines` property should be set to 0
- `titleLabel?.textAlignment` should be defined when `titleLabel?.numberOfLines = 0`, to avoid text going out of bounds

Success Criterion 1.4.5 Images of text. This Success Criterion ensures that information is conveyed through text instead of images containing textual content except for the following images of text [17] [17]:

- Customizable: The image of text can be visually customized to the user's requirements;
- Essential: A particular presentation of text is essential to the information being conveyed. For example, logotypes (text that is part of a logo or brand name) are considered essential.

[17]https://www.w3.org/WAI/WCAG21/Understanding/images-of-text

The testing process can be implemented only manually.

Success Criterion 1.4.10 Reflow. The Success Criterion requires the content to be easily readable without requiring horizontal scrolling. In case of large text font size, when other elements might be pushed out of view, make sure that users can still access the content, for instance, by vertical scrolling. Except for the cases when two-dimensional scrolling is required, for example, maps, diagrams, videos, games, presentations, data tables, and so on. Usually, content should be put on a scrollable layout and have constraints instead of fixed sizes, but sometimes there can be a case when a fixed size is needed for UI elements that cannot calculate their own size due to initialization in the early stages of the view lifecycle. The conformance to this Criterion can be tested manually with the help of assistive technologies by increasing the font size and checking if the layout is broken. By automated testing, it is possible only to ensure that labels were set correctly to adapt for resized text (SC 1.4.4 Resize text).

Success Criterion 1.4.11 Non-text contrast. The Criterion ensures that meaningful visual cues achieve 3:1 against the background [18] [17]. Designers should provide images with the required contrast. The conformance for this Criterion can be tested with the help of assistive technologies. However, it is possible to test contrast ratio of controls, for example, by comparing background color and borders color (the calculation can be taken from SC 1.4.3 Contrast (minimum))

Success Criterion 1.4.12 Text spacing. This Success Criterion ensures that content implemented using markup languages adapts to user-defined text settings. On iOS, Swift is not a markup language. However, text style properties can be changed on `UILabel` by setting `attributedString` property with modified style `NSMutableParagraphStyle`. According to Criterion, no loss of content or functionality should occur by setting all of the following and by changing no other style property:

- Line height (line spacing) to at least 1.5 times the font size;
- Spacing following paragraphs to at least 2 times the font size;
- Letter spacing (tracking) to at least 0.12 times the font size;
- Word spacing to at least 0.16 times the font size.

During testing in case if `attributedString` of `UILabel` or `UITextView` has `paragraphStyle` [19], then the following should be correct:

```
lineHeightMultiple >= 1.5
```

or

```
lineSpacing >= (font * 1.5)
```

[18]https://www.w3.org/WAI/WCAG21/Understanding/non-text-contrast

[19]https://developer.apple.com/documentation/uikit/nsmutableparagraphstyle

otherwise show error. The same audit can be implemented on other properties. Here is the possible error of paragraph style:

- lineSpace (lineHeightMultiple) should be at least 1.5 times the font size
- paragraphSpacing (paragraphSpacingBefore) should be at least 2 times the font size
- kern (tracking) should be at least 0.12 times the font size

Success Criterion 1.4.13 Content on hover or focus. This Success Criterion ensures that when pointer hover or keyboard focus causes extra content to appear and disappear, the following conditions apply:

- Dismissible: Users can dismiss the additional content without having to move the pointer hover or keyboard focus, except when the content conveys an input error or doesn't obstruct or replace other content.
- Hoverable: If pointer hover can reveal the extra content, users can move the pointer over the additional content without it disappearing.
- Persistent: The additional content remains visible until the hover or focus trigger is removed, the user dismisses it, or the information it provides becomes invalid.

On iOS, the additional content can be presented by `UIAlertController`, where according to this requirement should always be a cancel action to close the view. Another way is to use `UIPopoverPresentationController` where its behavior is managed by `UIPopoverPresentationControllerDelegate`. However, the testing requires a manual method. From automated testing, it is possible to ensure only if `UIAlertViewController` has more than one action then check for the following error:

- `UIAlertViewController` should have action with cancel style

The reason to check only if more than one action is because the alert view with one action can be as confirmable with OK button. And usually it is set as `default` style.

4.1.2 Operable Principle

The second principle is Operable: UI components and navigation must be operable by in the users' preferred way. This means that users must be able to navigate and interact with the application using a variety of input methods such as a screen reader, voice control, keyboard, mouse, touch screen and etc. All guidelines with Success Criteria for this principle are listed in Table 4.

Success Criterion 2.1.1 Keyboard. The Criterion requires all app functionalities to be accessible with a keyboard. On iOS, the property `accessibilityRespondsToUserInteraction` can be set to false for UI elements that only display information. But the testing process requires manual checks with the help of assistive technologies.

Success Criterion 2.1.2 No keyboard trap. A mobile application should allow users to navigate through all interactive elements using the keyboard without getting stuck in a keyboard trap. A keyboard trap is a situation where a user cannot move focus away from a particular element or section of the app using only the keyboard. Meeting SC 1.4.13 and SC 2.1.1 can help to satisfy this criterion as well.

Success Criterion 2.1.4 Character key shortcuts. The criteria guarantee that accidental activation of shortcuts is prevented while using assistive technologies [20] [17]. Some assistive technologies imitate keystrokes to execute actions, which can lead to unintended shortcuts being triggered. If a keyboard shortcut is implemented in content using only letters (including upper- and lower-case letters), punctuation, number, or symbol characters, then at least one of the following is true:

- Turn off: it is possible to turn the shortcut off;
- Remap: it is possible to remap the shortcut to include one or more non-printable keyboard keys (Command, Alt);
- Active only on focus

On iOS, turn off option can be implemented via `UIViewController` method:

```
func removeKeyCommand(_ keyCommand: UIKeyCommand)
```

Remapping should be done by initialising `UIKeyCommand` with `modifierFlags` property [21]:

```
let deleteCommand = UIKeyCommand(
                        title: "Delete",
                        action: #selector(delete),
                        input: "d",
                        modifierFlags: .command
                    )
```

And the availability is managed by `state` property of `UIKeyCommand`. However, the testing requires manual audit with the help of keyboard-type assistive technology to indicate if shortcut invoke wrong action.

Success Criterion 2.2.1 Timing adjustable. The criterion ensures that each time limit is enough to implement the task. There are 6 requirements provided by WCAG21

[20] https://www.w3.org/WAI/WCAG21/Understanding/character-key-shortcuts
[21] https://developer.apple.com/documentation/uikit/uikeymodifierflags

where at least one is required [22] [17]. However, the use of a timer in presenting content or implementing a task may vary from showing a short message named Toast or pop-ups (for example, on Google mail when a message send the pop-up appears to cancel the sending) to updating API call. It is not possible to test the purpose of timer via an automated tool, which is why it requires only manual testing.

Success Criterion 2.2.2 Pause, stop, hide. The criterion ensures that users have the ability to pause, stop, or hide any moving, blinking or scrolling elements displayed on the screen, unless those are essential. The example of essential moving is a loading indicator. It is usually animating to show a pre-load phase or similar situation in order to not cause users to think that content was frozen or broken. On iOS, if `Boolean` property `UIAccessibility.isReduceMotionEnabled` it is possible to disable non-essential animations. However, it is difficult to test to via automated test, as the purpose of moving, blinking or scrolling element can be indicated by a real person.

Success Criterion 2.3.1 Three flashes or below threshold. Success Criterion allows users to access the full content of a site without inducing seizures due to photosensitivity [23]. The purpose of this criterion is to ensure that no more than three flashes per second are being presented, because it can cause an epileptic seizure. Flashing can occur from the rendering process of an image, and developers have no authority over these aspects, which can be addressed through device/connection performance. However, such features as animation, device torch or video content should be used carefully to not exceed the flash thresholds, and testing requires a human to understand the semantic purpose.

Success Criterion 2.4.3 Focus order. The Criterion requires that navigation focus order over elements preserves both meaning and operational significance of the view [24] [17]. As the testing of meaning and operations can be tested only by humans, no automation is possible for this case.

Success Criteria 2.4.4 Link purpose (in context). The purpose of each link should be clear without needing additional context. It can be tested manually or by using assistive technologies, because automated test cannot identify if the purpose is clear.

Success Criterion 2.4.6 Headings and labels. The intent of this Success Criterion is to help users understand what information is contained on a screen. This criterion does not check the presence or identification of content serving as headings or labels, because it is addressed separately by SC 1.3.1: Info and Relationships. It rather requires headings or labels be descriptive if presence. Such aspects as descriptiveness or clearness can be tested by human being [25] [17]. However, the conformance to Apple Inc provided guidelines for labels and headings text can be tested, and it is shown on SC 4.1.2 Name, role, value.

[22]https://www.w3.org/WAI/WCAG21/Understanding/timing-adjustable

[23]https://www.w3.org/WAI/WCAG21/Understanding/three-flashes-or-below-threshold

[24]https://www.w3.org/WAI/WCAG21/Understanding/focus-order

[25]https://www.w3.org/WAI/WCAG21/Understanding/headings-and-labels

Success Criterion 2.4.7 Focus visible. Any operable UI element should have a visible focus indicator [26] [17]. On iOS, VoiceOver and Switch Control already have a focus frame. The testing process will require manual using this technologies to indicate if the placement is correct and that the colour is clearly visible.

Success Criterion 2.5.1 Pointer gestures. The Success Criterion requires to provide one finger (single-point) and reduced gestures operation for all functions [27] [17]. For instance, the scrolling or zooming can be additionally provided by buttons with arrows up and down or plus and minus respectively. The presence of alternative operation can be tested only manually by human.

Success Criterion 2.5.2 Pointer cancellation. The Criterion requires touches to be cancellable, with exception for cases when the down-event triggered by touch is essential or the up-event reverses any outcome of the preceding down-event. The purpose of touch can be tested manually or with the help assistive technologies. UI tests can also test the results of touch events.

Success Criterion 2.5.3 Label in name. The Criterion requires visual label for controls to be a trigger for speech activation. This can be tested manually with the help of Voice Control [28] or by automated tool. The second option should check if `accessibilityLabel` of controls is equal or contained in the text that is presented visually. The following error case can be added to cases mentioned in SC 4.1.2 Name, role, value:

- `accessibilityLabel` should match or be contained in title of element, to trigger Voice Contol.

Success Criterion 2.5.4 Motion actuation. Following this Criterion, functionality should not rely solely on device or user motion. Alternative UI component should be provided for motion-triggered actions and responding to the motion can be disabled to prevent accidental actuation, except when supported interface or essential [29] [17]. The conformance to this Criterion can be tested manually or by UI tests.

4.1.3 Understandable Principle

The third principle is Understandable: information and the operation of the user interface must be understandable. This means that the language and presentation of the content must be clear and easy to understand, and that the user interface should be consistent and predictable. All guidelines with Success Criteria for this principle are gathered in Table 5.

[26]https://www.w3.org/WAI/WCAG21/Understanding/focus-visible
[27]https://www.w3.org/WAI/WCAG21/Understanding/pointer-gestures
[28]https://support.apple.com/en-us/HT210417
[29]https://www.w3.org/WAI/WCAG21/Understanding/motion-actuation

Success Criterion 3.1.1 Language of software. The default human language of all content should be programmatically determined. Each mobile application source code has its own specific implementation of setting default language which cannot be tested by using common approach.

Success Criterion 3.2.1 On focus. The functionality of UI element that receives focus should be predictable and it does not initiate a change of context [30] [17]. On iOS, assistive technologies that uses focus frame are not triggering an action on element while just listing or describing elements on a view. Actions are taken, for example, after double tap on screen for VoiceOver. However, there is possibility to override function named `accessibilityElementDidBecomeFocused`, where no actions on changing context should be implemented. Changes of context difficult to indicate by automation testing, unless using UI or Screenshot testing and comparing view state before and after actions. Manual testing with the help of assistive technologies is the most effective one.

Success Criterion 3.2.2 On input. The intent of this Criterion is to assure that entering data or selecting a form control has predictable effects. Context changes are acceptable only when it is predictable that they will occur as a direct result of the user's action. For example, checking a checkbox button or inserting text into a text field changes setting of the control, but tapping on links or tabs in a tab control should activate the control and not change the setting of that control. It is also considered a failure of the criterion if after changing setting a new window is appearing without advanced warning or the data from text fields is submitted automatically [31] [17]. The testing process can be implemented by UI tests to check if only intended changes has appeared on a screen, and manually.

Success Criterion 3.3.1 Error identification. The Criterion requires to notify users that an error has occurred while entering data and to determine what is wrong by showing and announcing error message [32] [17]. Error message visual presentation can be implemented with `UIAlertViewController` or `UILabel` near to input field. The second option should be followed with the accessibility announcement by `UIAccessibility.post` static method [33]. Testing is possible with UI tests and manually.

Success Criterion 3.3.2 Labels or instructions. The intent of this Success Criterion is to have instructions or labels that explain what data is needed for input control. This criterion is only about clear instructions that provided for all users. It does not require that labels or instructions be correctly marked up or identified, that is required in SC 1.3.1: Info and Relationships. Furthermore, this Criterion does not take into consideration whether or not accessibility label or hint were provided for text fields as this aspect is covered separately by next SC 4.1.2: Name, Role and Value. Therefore, the testing can be implemented manually for checking if the instruction has clear, comprehensive

[30]https://www.w3.org/WAI/WCAG21/Understanding/on-focus
[31]https://www.w3.org/WAI/WCAG21/Understanding/on-input
[32]https://www.w3.org/WAI/WCAG21/Understanding/error-identification
[33]https://apple.co/3rZ1NvQ

explanation for all users by Accessibility Quality Assurance person.

Success Criteria 3.3.3 Error suggestion. This Success Criterion ensures that users receive appropriate suggestions for correction of an input error if it is possible. Albeit Success Criterion 3.3.1 ensures notification of errors, persons with cognitive limitations may find it difficult to understand how to correct the errors [34] [17]. That is important to test the suggestion text for correctness and descriptiveness manually.

Success Criteria 3.3.4 Error prevention (legal, financial, data). Actions that cause legal commitments or financial transactions, that modify or delete user-controllable data in data storage systems, or that submit user test responses, at least one of the following is true [35]:

- Reversible: submissions are reversible.
- Checked: SC 3.3.2 Labels or instructions, SC 3.3.3 Error suggestion.
- Confirmed: it is possible to review, confirm and correct data before the submission.

Functionalities such as undo, correct and confirm can be tested manually depending on a context and required business logic.

4.1.4 Robust Principle

The last principle is Robust: content must be robust enough that it can be interpreted reliably by a wide variety of user agents, including assistive technologies. This means that the content should be compatible with current and future technologies and should be able to adapt to changes in user agent technology. All guidelines with Success Criteria for this principle are listed in Table 6.

Success Criteria 4.1.1 Parsing. This Success Criterion requires code be error-free and without deprecated functions. Failure to update the code to modern standards may lead to unexpected behavior with assistive technologies. Application functionality should work correctly on all supported iOS versions. Automated Unit, UI and manual tests as well as linting tools can ensure the quality of a code base.

Success Criteria 4.1.2 Name, role, value. The Criterion ensures that all UI elements have name and role programmatically determined; states, properties, and values that set by the user are programmatically set; and notification of changes to these items is available to user, including assistive technologies [36] [17].

On iOS, setting `isAccessibilityElement` to `true` makes UI element visible for assistive technologies. The property for instances of `UIControl` class is `true` by default,

[34]https://www.w3.org/WAI/WCAG21/Understanding/error-suggestion
[35]https://www.w3.org/WAI/WCAG21/Understanding/error-prevention-legal-financial-data
[36]https://www.w3.org/WAI/WCAG21/Understanding/name-role-value

while other elements requires manual settings if needed. For accessibility-visible elements accurate and helpful information should be set for next properties: label, trait, hint, frame and value. By following Apple Inc guideline it is possible to test if there is accessibility faults.

The property `accessibilityLabel` gives succinct name for element, which takes element title or text as its default value [37] [7]. Here are accessibility label possible error cases [38]:

1. `accessibilityLabel` or its default value by means of title/text of element should be defined.
2. `accessibilityLabel` should not be empty string.
3. `accessibilityLabel` should not contain Stop Words (the type of the control or view)
4. `accessibilityLabel` should not be a white space
5. `accessibilityLabel` should start with a capitalized letter
6. `accessibilityLabel` should not end with a period.
7. there should be no repeated strings for `accessibilityLabel`

The property `accessibilityTraits` gives combination of accessibility traits that best characterizes the accessibility element. There are traits that are mutually exclusive and conflicting traits can be indicated during automated testing [39]. Here are accessibility traits possible error case:

- `accessibilityTraits` for element are conflicting.

Success Criteria 4.1.3 Status messages. The intent of Success Criterion is to ensure users are notified by assistive technologies about status changes that don't take focus. Such changes can be announced by the same method as in SC 3.3.1 Error identification.

4.1.5 Summary

The analysis of required accessibility Success Criteria for mobile applications showed that automating accessibility testing is challenging task, because accessibility testing checks semantic complexity of the UI that computers cannot independently comprehend. Therefore, it is not possible to cover all 44 criteria. However, automated testing tools can cover important basic requirements and possible accessibility issues that can be detected by automated testing tool were offered.

[37] https://developer.apple.com/documentation/uikit/uiaccessibilityelement/1619577-accessibilitylabel
[38] https://apple.co/3OQ8TvJ
[39] https://apple.co/47pJgJu

4.2 Answering RQ2

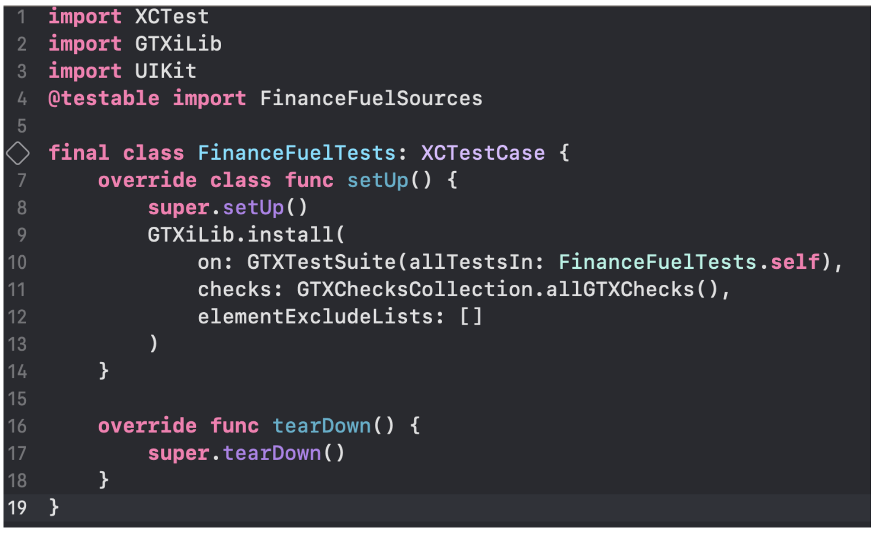
4.2.1 GTXiLib analysis

Description. GTXiLib is Google tool for Accessibility Testing for the iOS applications. As stated in documentation, it enhances unit tests by conducting accessibility checks alongside. GTXiLib is able to accomplish this by hooking into the test tear-down process and invoking the registered accessibility checks (such as check for presence of accessibility label) on all elements on the screen [5].

Initial analysis. Based on technical stack and requirements a few important disadvantages could be highlighted:

- it is written in Objective-C which is old, non-popular programming language for iOS platforms. As a consequence, providing support becomes challenging;
- there is no comprehensive documentation for the tool;
- integration into projects with CocoaPods only, disadvantages of which are shown on Table 7;
- it has low popularity and unresolved opened issues for last few years.

Testing. Testing process by GTXiLib starts with integration of tool into sample project FinanceFuel via installing CocoaPods and initializing required files. Next, in test target of project, to run all tests of specific test class the following snippet of code Figure 1 was added.

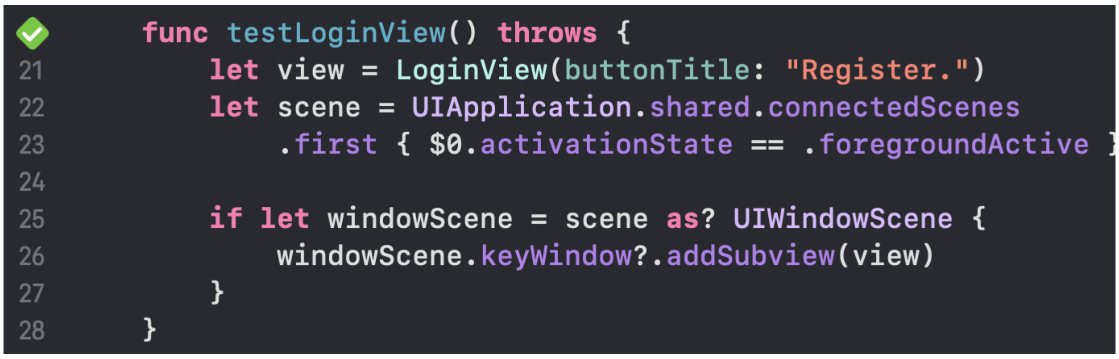


```swift
import XCTest
import GTXiLib
import UIKit
@testable import FinanceFuelSources

final class FinanceFuelTests: XCTestCase {
    override class func setUp() {
        super.setUp()
        GTXiLib.install(
            on: GTXTestSuite(allTestsIn: FinanceFuelTests.self),
            checks: GTXChecksCollection.allGTXChecks(),
            elementExcludeLists: []
        )
    }

    override func tearDown() {
        super.tearDown()
    }
}
```

Figure 1. GTXiLib setup.

GTXiLib needs to be setup and installed once per test suite, so `setup` function of the class was overridden. To test the view it needs to be added to the keyWindow, as the library uses it's root object and then runs the checks on its subviews. However, the most important part is to set the frame for the view, which was not mentioned in the documentation for the tool. The frame of view is a rectangle, which describes the view's location and size in its superview's coordinate system. It means that test will not run on view's which has Auto Layout. Auto Layout makes UI layout adaptive and flexible for different screen sizes, which is important for accessible application. It dynamically calculates the size and position of all the views in your view hierarchy, based on constraints placed on those views [40].

As an example, `LoginView` which is setup with Auto Layout was tested with GTXiLib on all checks available there. In order to fail test, the button was set intentionally with punctuation error, but tests showed false positive results Figure 2

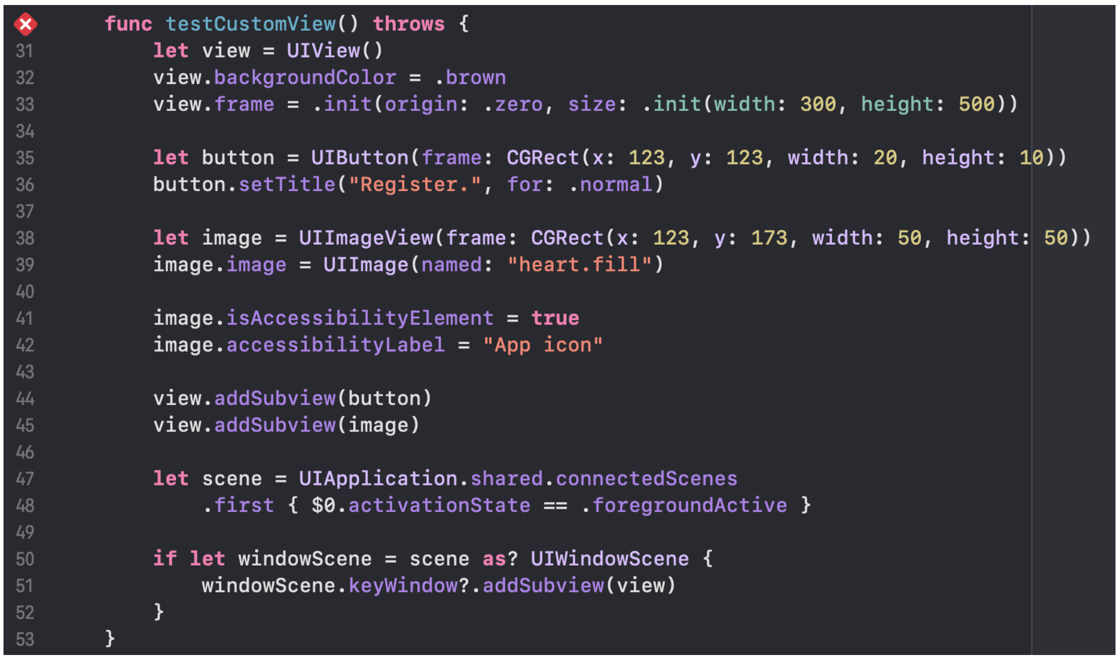


```
    func testLoginView() throws {
        let view = LoginView(buttonTitle: "Register.")
        let scene = UIApplication.shared.connectedScenes
            .first { $0.activationState == .foregroundActive }

        if let windowScene = scene as? UIWindowScene {
            windowScene.keyWindow?.addSubview(view)
        }
    }
```

Figure 2. GTXiLib `testLoginView` false positive result.

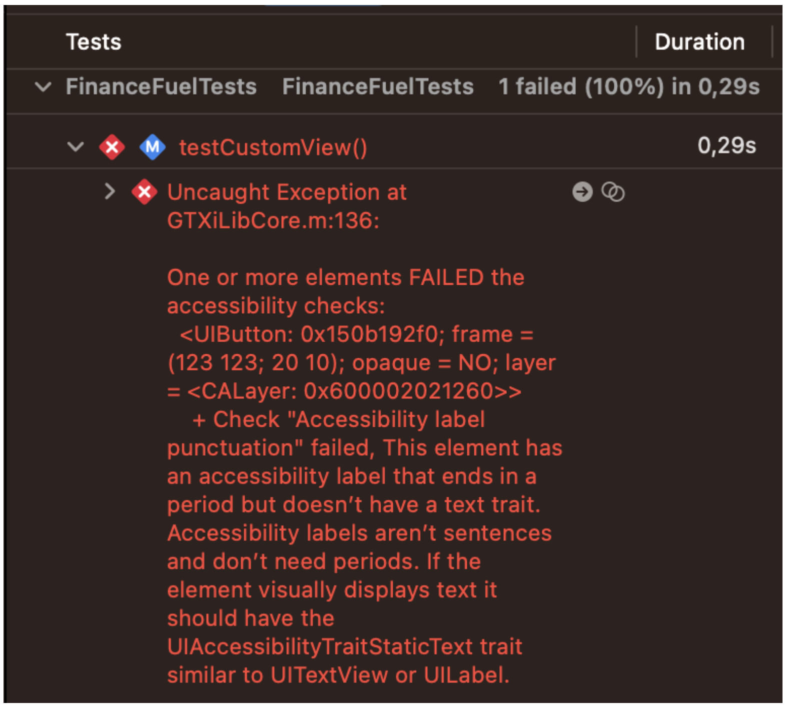
For comparison, another view was created with frame setups instead of constraint-based approach with the same intentional error. Tests failed (Figure 3) and showed description in project navigator sections shown in Figure 4.

[40]https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html



```
    func testCustomView() throws {
        let view = UIView()
        view.backgroundColor = .brown
        view.frame = .init(origin: .zero, size: .init(width: 300, height: 500))

        let button = UIButton(frame: CGRect(x: 123, y: 123, width: 20, height: 10))
        button.setTitle("Register.", for: .normal)

        let image = UIImageView(frame: CGRect(x: 123, y: 173, width: 50, height: 50))
        image.image = UIImage(named: "heart.fill")

        image.isAccessibilityElement = true
        image.accessibilityLabel = "App icon"

        view.addSubview(button)
        view.addSubview(image)

        let scene = UIApplication.shared.connectedScenes
            .first { $0.activationState == .foregroundActive }

        if let windowScene = scene as? UIWindowScene {
            windowScene.keyWindow?.addSubview(view)
        }
    }
```

Figure 3. GTXiLib `testCustomView` accessibility faults detection.



```
Tests                                              Duration
⌄ FinanceFuelTests   FinanceFuelTests   1 failed (100%) in 0,29s
   ⌄ testCustomView()                                 0,29s
      > Uncaught Exception at
        GTXiLibCore.m:136:

        One or more elements FAILED the
        accessibility checks:
          <UIButton: 0x150b192f0; frame =
        (123 123; 20 10); opaque = NO; layer
        = <CALayer: 0x600002021260>>
            + Check "Accessibility label
        punctuation" failed, This element has
        an accessibility label that ends in a
        period but doesn't have a text trait.
        Accessibility labels aren't sentences
        and don't need periods. If the
        element visually displays text it
        should have the
        UIAccessibilityTraitStaticText trait
        similar to UITextView or UILabel.
```

Figure 4. GTXiLib `testCustomView` accessibility faults report.

Another issue with the tool was indicated. Running both tests `testLoginView` from Figure 2 and `testCustomView` from Figure 3 in one class gives misleading report by showing errors of the latter in the test report of the former. Although some manipulations with `UIWindowScene` might help, it requires additional setups and workarounds. Also, when running tests on Xcode(14.3.1 version) simulator, for making next launches of tests work correctly, simulator should be quit. Otherwise, no checks will be applied and all views will pass the tests successfully, even though they were failed on a preceding launch.

The results of examining GTXiLib code base on what the checks are implemented there are shown in Table 2 by means of "yes" in the row of the covered accessibility issue. Furthermore, another accessibility issue was added to the table, as the tool had check on target size of touch for the controls, such as button. The issue is marked with asterisk because it non-required Criterion by the EN 301 549 Standard[4].

The code presented in this section is available in commit with the message "Analyse GTXiLib testing tool" [41].

[41] https://github.com/KarimovaKarina/Thesis/commit/ffbaf03735e2022f4d0aaf54eced72b3caf21316

4.3 Answering RQ3

4.3.1 Overview of OAT

OAT[42] is open-source automated accessibility testing tool, written in Swift. The integration of the tool into the project is managed by native package manager Swift Package Manager described in Table 7 [14]. To start the testing package needs to be imported in the swift file with test class and the view can be tested by sending it as a parameter to the function named `checkAccessibility`. Overall, the function has few parameters:

```
public func checkAccessibility(
    _ view: UIView,
    with settings: AccessibilitySettings = .default,
    file: StaticString = #file,
    testName: String = #function,
    line: UInt = #line
) {
    let errors = collectErrors(for: view, with: settings)
    errors.forEach { error in
        XCTFail("\n -> " + error.errorMessage, file: file, line: line
           )
    }
}
```

where, `view` is the view to be tested. `settings` is the public structure which contains two properties: `excluding` and `recursiveChecking`. The former is the list of excluded checks. The latter is of `Bool` value and stands for whether checks should be applied recursively for subviews or not. `file` is the file where the failure occurs. The default is the filename of the test case where this function is called. `testName` is the name of test function where the failure occurs. The default is the function name where `checkAccessibility` function is called. `line` is the line number where the failure occurs. The default is the line number where this function is called. However, last three parameters are service parameters of the function which are not supposed to be filled in.

```
public struct AccessibilitySettings {
    let excluding: [ExcludedChecks]
    let recursiveChecking: Bool

    public  init(
        excluding: [ExcludedChecks],
        recursiveChecking: Bool
    ) {
        self.excluding = excluding
        self.recursiveChecking = recursiveChecking
    }
}
```

[42] https://github.com/KarimovaKarina/OAT

```
public extension AccessibilitySettings {
    static var `default`: Self {
        .init(excluding: [], recursiveChecking: true)
    }
}
```

The parameter `excluding` has type `[ExcludedChecks]`, which is list of enumeration cases. It represents the list of checks, that can be excluded from test, for example `images`, as images can be just a decoration part of UI.

In case when accessibility testing of some project is planned to be iterated, it is possible to check specific UI element without passing its subviews by setting `recursiveChecking` to `true`

Accessibility faults that were offered to be checked in section 4.1 are used as values that have a common type, for instance, `AccessibilityHintError`:

```
enum AccessibilityHintError: Error {
    case hintIsEmpty
    case containsType([String])
    case firstWordIsNotCapitalized
    case doesNotEndWithPeriod
    case containsLabel(String)
}
```

Each accessibility error enumeration is conforming to common `AccessibilityError` and its own setup of value `errorMessage`. The value represents a textual explanation of detected accessibility faults.

```
protocol AccessibilityError {
    var errorMessage: String { get }
}
```

When tests has failed the description of accessibility issues are presented in the file sent as parameter. Also they can be checked on project's Issue Navigator and Report Navigator.

4.3.2 Practical use of OAT

To show how the tool works in practice it is integrated into sample project FinanceFuel. We will test the first two views that users see when launching the app depicted on Figure 5: `WelcomeView` and `LoginView` classes.

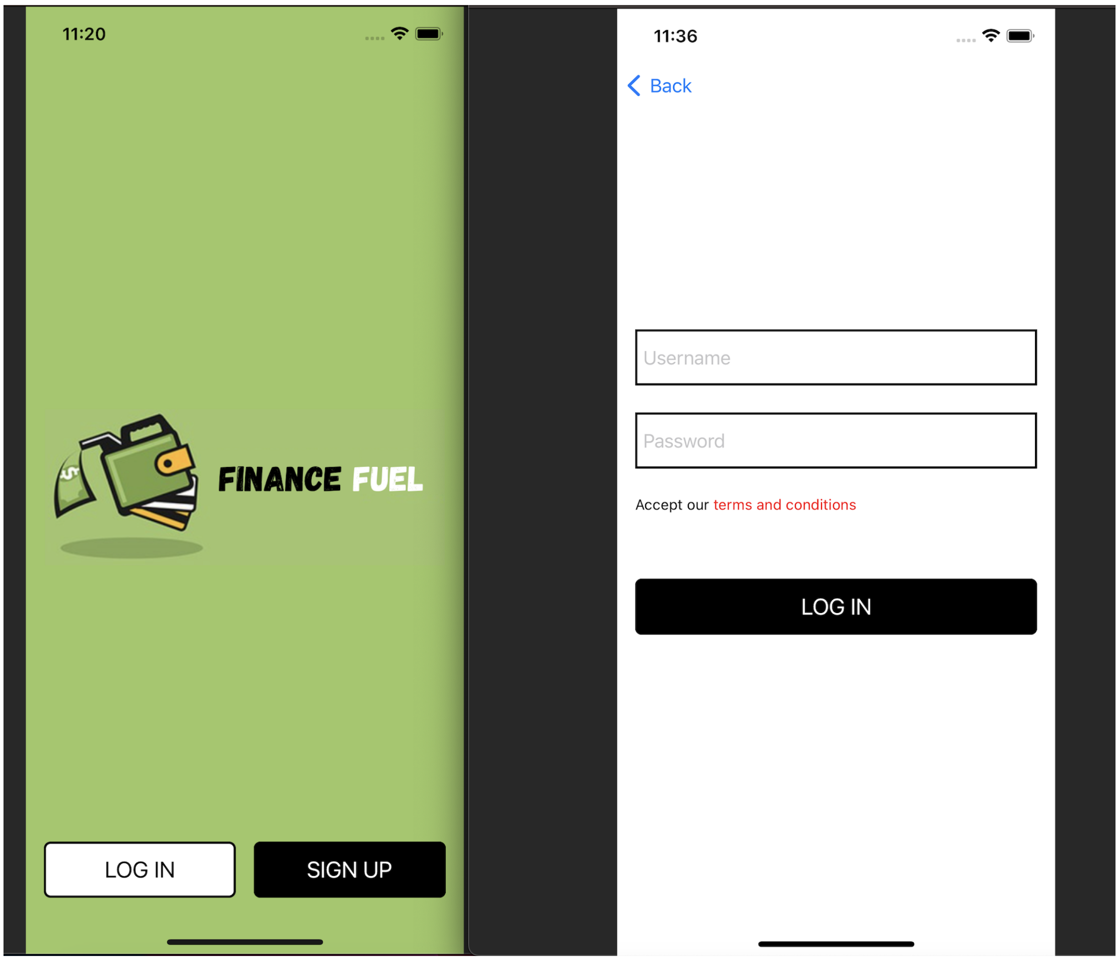
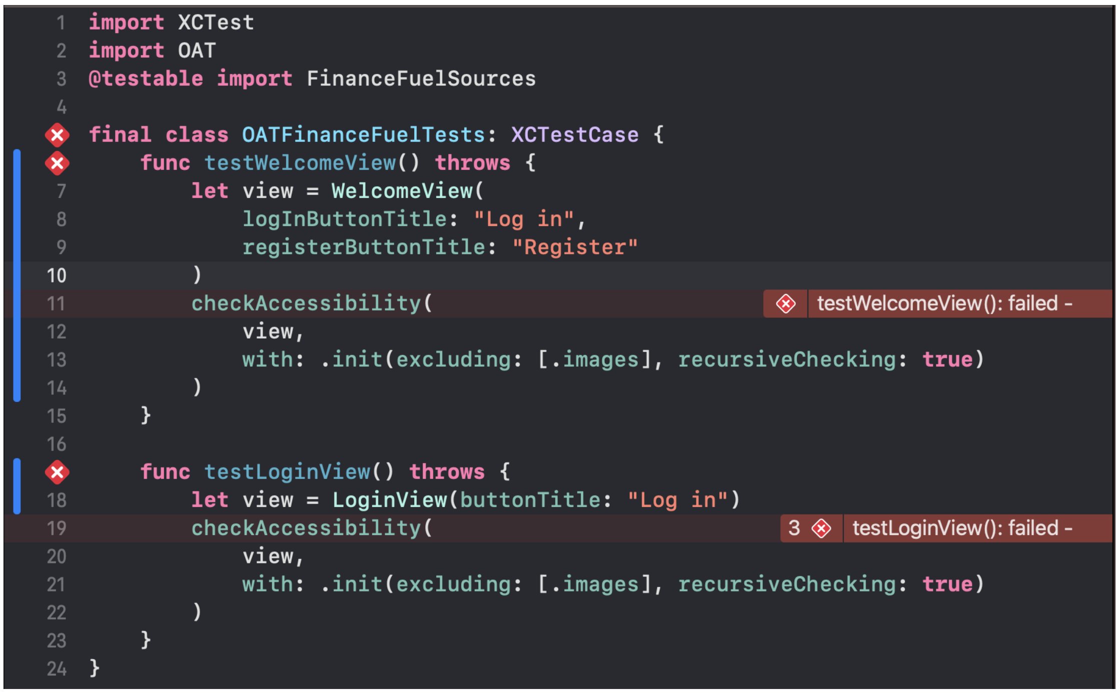


Figure 5. FinanceFuel Welcome and Login views design

On Figure 6 presented the code where two selected views are initialized and sent to `checkAccessibility`. For checking `WelcomeView` additional parameters are sent to ignore image as it meant only for decoration. After launching tests, OAT indicated the following accessibility issues:



```swift
import XCTest
import OAT
@testable import FinanceFuelSources

final class OATFinanceFuelTests: XCTestCase {
    func testWelcomeView() throws {
        let view = WelcomeView(
            logInButtonTitle: "Log in",
            registerButtonTitle: "Register"
        )
        checkAccessibility(                          testWelcomeView(): failed -
            view,
            with: .init(excluding: [.images], recursiveChecking: true)
        )
    }

    func testLoginView() throws {
        let view = LoginView(buttonTitle: "Log in")
        checkAccessibility(                          3   testLoginView(): failed -
            view,
            with: .init(excluding: [.images], recursiveChecking: true)
        )
    }
}
```

Figure 6. OAT FinanceFuel accessibility faults detection.

1. `LoginView`

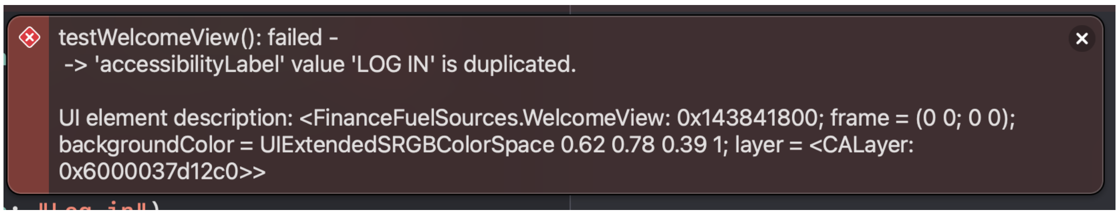
   - "'accessibilityLabel' value 'LOG IN' is duplicated."

     Note: The view was initialize with the same titles for button. (FYI: titles are uppercased inside that view class, that is why it reports uppercased value)

2. `WelcomeView`

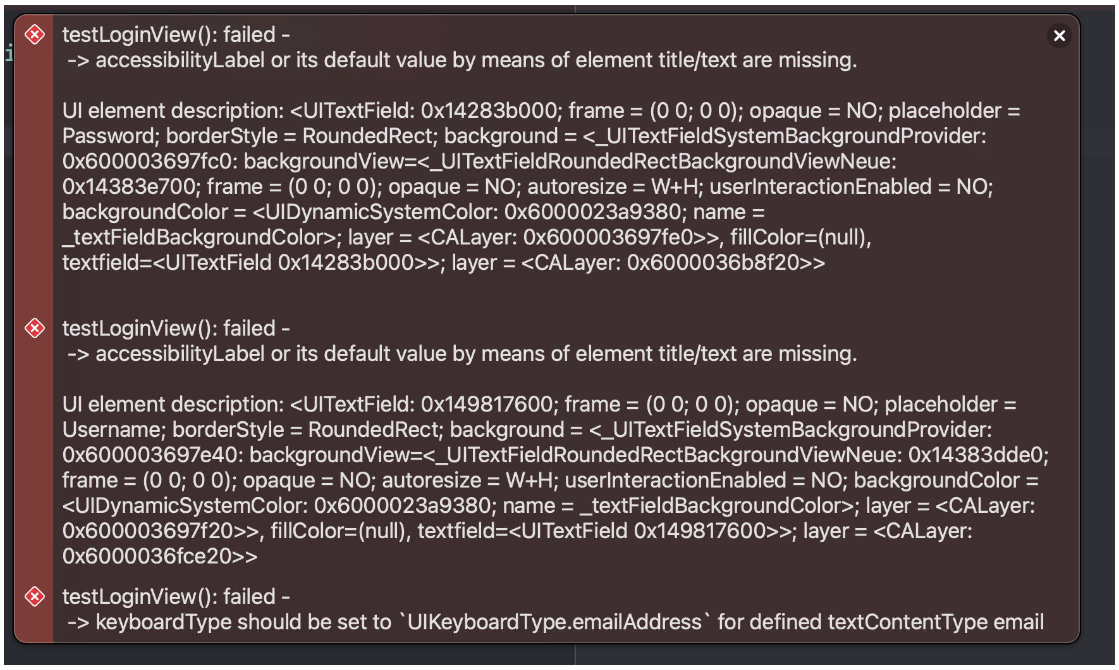
   - "accessibilityLabel or its default value by means of element title/text are missing."
   - "accessibilityLabel or its default value by means of element title/text are missing. "
   - "keyboardType should be set to ‘UIKeyboardType.emailAddress‘ for defined textContentType email"

The full description of accessibility faults with related element information are shown on Figure 7 and Figure 8.



```
testWelcomeView(): failed -
 -> 'accessibilityLabel' value 'LOG IN' is duplicated.

UI element description: <FinanceFuelSources.WelcomeView: 0x143841800; frame = (0 0; 0 0);
backgroundColor = UIExtendedSRGBColorSpace 0.62 0.78 0.39 1; layer = <CALayer:
0x6000037d12c0>>
```

Figure 7. OAT `testCustomView` accessibility faults report.

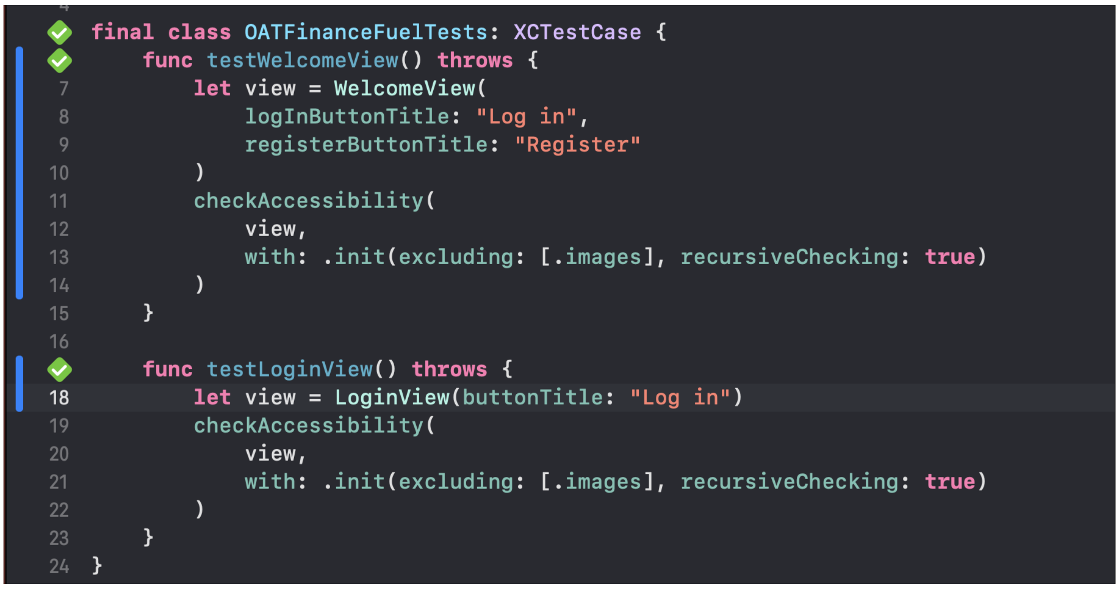


```
testLoginView(): failed -
 -> accessibilityLabel or its default value by means of element title/text are missing.

UI element description: <UITextField: 0x14283b000; frame = (0 0; 0 0); opaque = NO; placeholder =
Password; borderStyle = RoundedRect; background = <_UITextFieldSystemBackgroundProvider:
0x600003697fc0: backgroundView=<_UITextFieldRoundedRectBackgroundViewNeue:
0x14383e700; frame = (0 0; 0 0); opaque = NO; autoresize = W+H; userInteractionEnabled = NO;
backgroundColor = <UIDynamicSystemColor: 0x6000023a9380; name =
_textFieldBackgroundColor>; layer = <CALayer: 0x600003697fe0>>, fillColor=(null),
textfield=<UITextField 0x14283b000>>; layer = <CALayer: 0x6000036b8f20>>


testLoginView(): failed -
 -> accessibilityLabel or its default value by means of element title/text are missing.

UI element description: <UITextField: 0x149817600; frame = (0 0; 0 0); opaque = NO; placeholder =
Username; borderStyle = RoundedRect; background = <_UITextFieldSystemBackgroundProvider:
0x600003697e40: backgroundView=<_UITextFieldRoundedRectBackgroundViewNeue: 0x14383dde0;
frame = (0 0; 0 0); opaque = NO; autoresize = W+H; userInteractionEnabled = NO; backgroundColor =
<UIDynamicSystemColor: 0x6000023a9380; name = _textFieldBackgroundColor>; layer = <CALayer:
0x600003697f20>>, fillColor=(null), textfield=<UITextField 0x149817600>>; layer = <CALayer:
0x6000036fce20>>

testLoginView(): failed -
 -> keyboardType should be set to `UIKeyboardType.emailAddress` for defined textContentType email
```

Figure 8. OAT `testLoginView` accessibility faults report.

As we know from description which elements are not passing the checks, we can fix those issues. Additionally, error message for input filed gives us suggestion which keyboard type should be set. After adding improvements for elements in `LoginView` and setting non-duplicated strings for `WelcomView` initialization, tests passed successfully as shown in Figure 9.

```
private func setup() {
        usernameTextField.accessibilityLabel = "Email address"
        usernameTextField.keyboardType = .emailAddress
        passwordTextField.accessibilityLabel = "Password"

        ...
}
```



```
final class OATFinanceFuelTests: XCTestCase {
    func testWelcomeView() throws {
        let view = WelcomeView(
            logInButtonTitle: "Log in",
            registerButtonTitle: "Register"
        )
        checkAccessibility(
            view,
            with: .init(excluding: [.images], recursiveChecking: true)
        )
    }

    func testLoginView() throws {
        let view = LoginView(buttonTitle: "Log in")
        checkAccessibility(
            view,
            with: .init(excluding: [.images], recursiveChecking: true)
        )
    }
}
```

Figure 9. OAT `OATFinanceFuelTests` succeed.

4.3.3 Quality Assurance of OAT

To cover all accessibility issues that the tool promises to indicate, the extension to each objects that conforms to AccessiblityError is added in the test target of the OAT. The following code snippet shows the extension example on enum AccessibilityHintError:

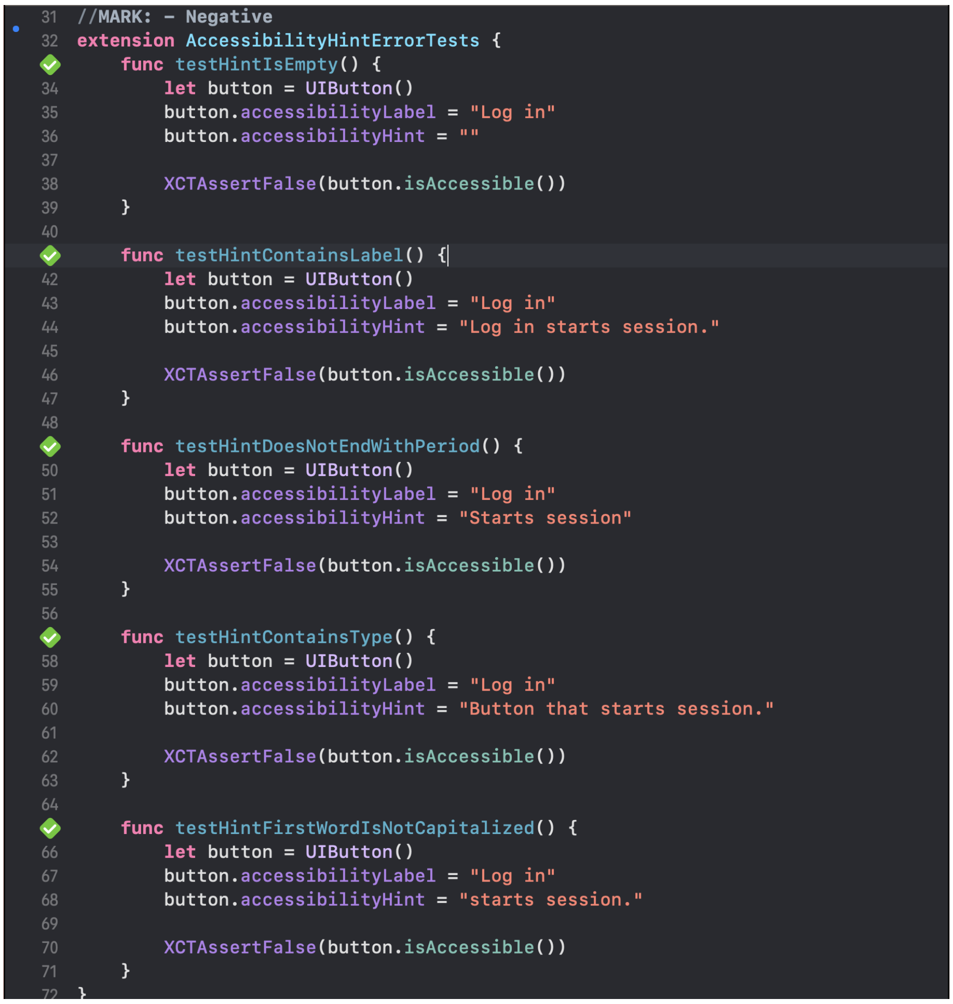
```
extension AccessibilityHintError: CaseIterable {
    public static var allCases: [AccessibilityHintError] {
        [
            .hintIsEmpty,
            .containsLabel("Log in"),
            .containsType(["button"]),
            .doesNotEndWithPeriod,
            .firstWordIsNotCapitalized
        ]
    }
}
```

In order to get a collection of all of values of type AccessibilityHintError it needs to be conformed to CaseIterable. For cases which has associated type mocked string was added.

Also UIView is extended with function isAccessible which indicates whether the element has accessibility fault or not. When errors are indicated, the function returns false which means element is not accessible.

```
extension UIView {
    func isAccessible() -> Bool {
        collectErrors(for: self, with: .default).isEmpty
    }
}
```

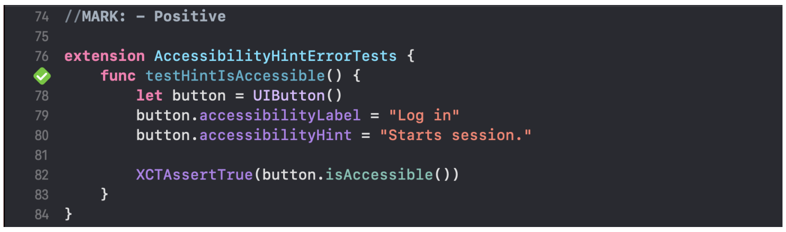
After necessary extension, in a test class AccessibilityHintErrorTests we can iterate through all cases of enum AccessibilityHintError and run tests for each, with the confidence that no functionality will be missed out. Tests functions and their result are shown on Figure 10, Figure 11 and Figure 12.



```
//MARK: - Negative
extension AccessibilityHintErrorTests {
    func testHintIsEmpty() {
        let button = UIButton()
        button.accessibilityLabel = "Log in"
        button.accessibilityHint = ""

        XCTAssertFalse(button.isAccessible())
    }

    func testHintContainsLabel() {
        let button = UIButton()
        button.accessibilityLabel = "Log in"
        button.accessibilityHint = "Log in starts session."

        XCTAssertFalse(button.isAccessible())
    }

    func testHintDoesNotEndWithPeriod() {
        let button = UIButton()
        button.accessibilityLabel = "Log in"
        button.accessibilityHint = "Starts session"

        XCTAssertFalse(button.isAccessible())
    }

    func testHintContainsType() {
        let button = UIButton()
        button.accessibilityLabel = "Log in"
        button.accessibilityHint = "Button that starts session."

        XCTAssertFalse(button.isAccessible())
    }

    func testHintFirstWordIsNotCapitalized() {
        let button = UIButton()
        button.accessibilityLabel = "Log in"
        button.accessibilityHint = "starts session."

        XCTAssertFalse(button.isAccessible())
    }
}
```

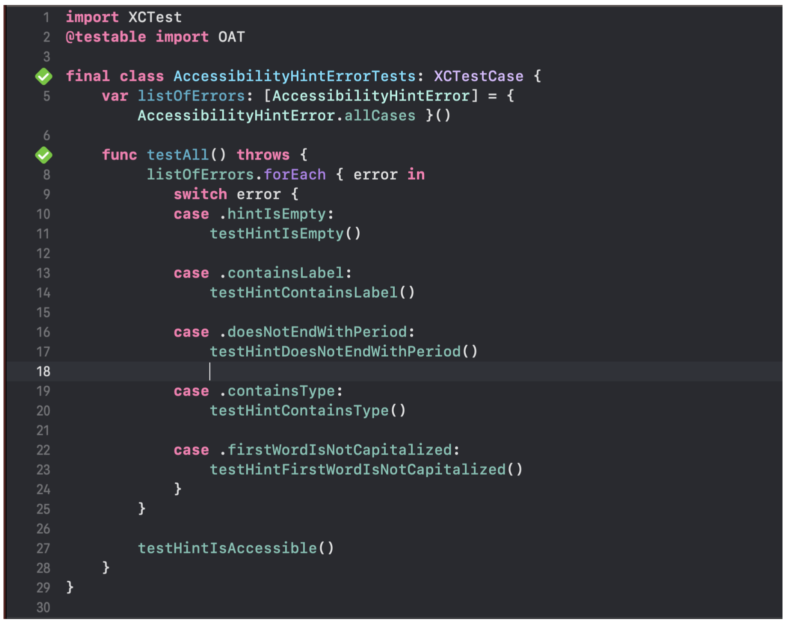
Figure 10. Tests for `AccessibilityHintError` negative cases

Each test function has its own UI element to test, where `accessibilityLabel` is set correctly, in order to check only `acceesibilityHint`. If tests are passing successfully, then checks work as expected. For example, in `testHintDoesNotEndWithPeriod` function we check if OAT detects that hint does not end with the period and we expect that element will not be accessible `XCTAssertFalse(button.isAccessible())`.



```
//MARK: - Positive

extension AccessibilityHintErrorTests {
    func testHintIsAccessible() {
        let button = UIButton()
        button.accessibilityLabel = "Log in"
        button.accessibilityHint = "Starts session."

        XCTAssertTrue(button.isAccessible())
    }
}
```

Figure 11. Tests for `AccessibilityHintError` positive case



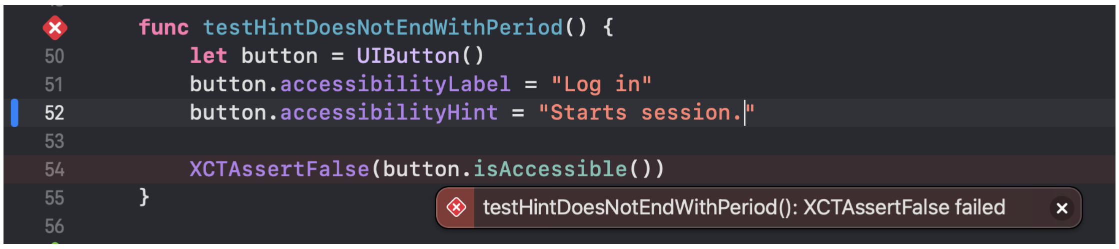
```
import XCTest
@testable import OAT

final class AccessibilityHintErrorTests: XCTestCase {
    var listOfErrors: [AccessibilityHintError] = {
        AccessibilityHintError.allCases }()

    func testAll() throws {
        listOfErrors.forEach { error in
            switch error {
            case .hintIsEmpty:
                testHintIsEmpty()

            case .containsLabel:
                testHintContainsLabel()

            case .doesNotEndWithPeriod:
                testHintDoesNotEndWithPeriod()

            case .containsType:
                testHintContainsType()

            case .firstWordIsNotCapitalized:
                testHintFirstWordIsNotCapitalized()
            }
        }

        testHintIsAccessible()
    }
}
```

Figure 12. Iteration through all `AccessibilityHintError` cases for testing

For testing purpose, we set value for accessibility hint which complies to accessibility requirements that it should end with a period and we receive failed case false negative result Figure 13:

```
func testHintDoesNotEndWithPeriod() {
        ...
        button.accessibilityHint = "Starts session."
        ...
}
```

The presented test approach makes sure that all accessibility faults are indicated by OAT checks.



```
func testHintDoesNotEndWithPeriod() {
    let button = UIButton()
    button.accessibilityLabel = "Log in"
    button.accessibilityHint = "Starts session."

    XCTAssertFalse(button.isAccessible())
}
```

testHintDoesNotEndWithPeriod(): XCTAssertFalse failed

Figure 13. Failed `AccessibilityHintError` test

4.3.4 Comparing test results of GTXiLib and OAT

Taking into account previously detected issues with GTXiLib tool in 4.2.1, GTXiLib checks works only if the view and its subviews has the frame setup with coordinates and size. As FinanceFuel project uses Auto Layout, we will use separate UI element without subviews with identified frame. As a test subject the instance of `AccessibleButton` was chosen, which is subclassing `UIButton`.

```
@testable import FinanceFuelSources
import Foundation

extension AccessibleButton {
    convenience init() {
        self.init(
            title: "button .",
            hint: "button opens link",
            color: .black,
            titleColor: .blue
        )
    }
}

var accessibleButton: AccessibleButton = {
    let button = AccessibleButton()
    button.accessibilityTraits.insert(.link)
    button.frame = CGRect(x: 0, y: 0, width: 30, height: 40)
    return button
}()
```

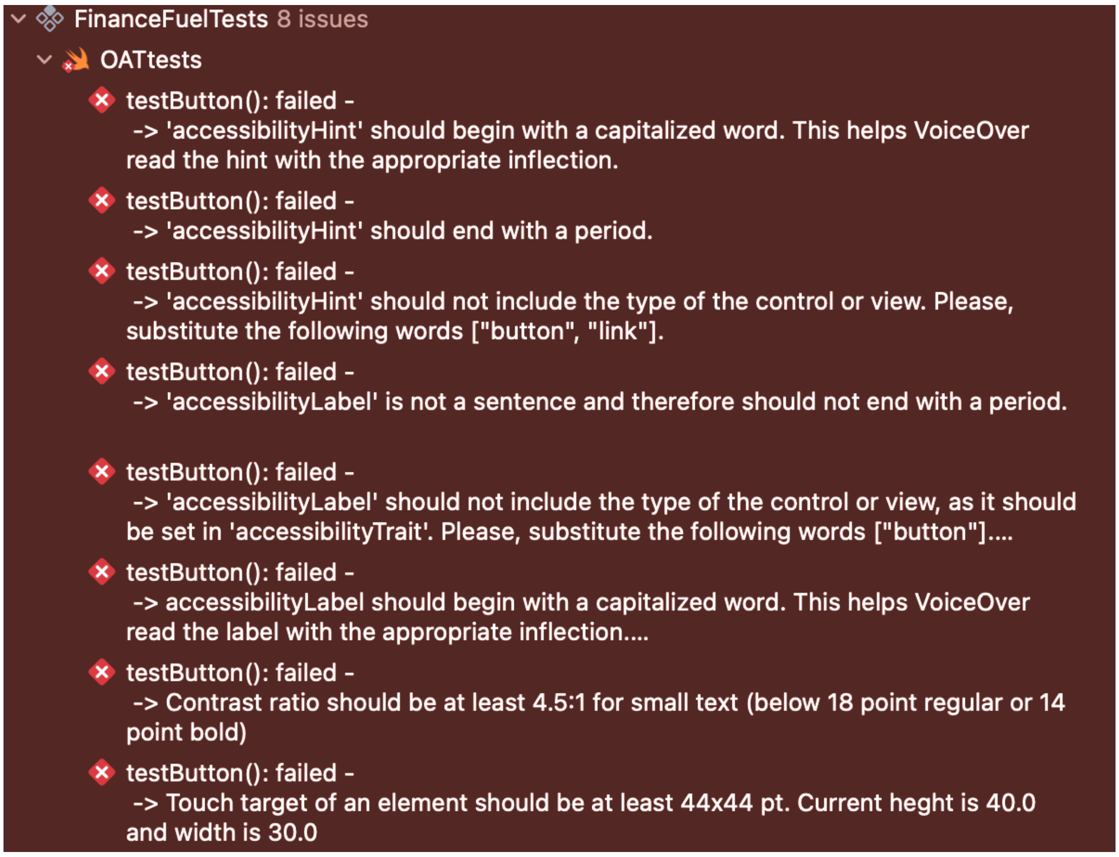
The button is specifically initialized with violated accessibility rules from WCAG and Apple Inc guidelines. Next, tested with both tools GTXiLib and OAT.

```
final class FinanceFuelTests: XCTestCase {
    ....

    func testButton() {
        let scene = UIApplication.shared.connectedScenes
```

```
            .first { $0.activationState == .foregroundActive }

        if let windowScene = scene as? UIWindowScene {
            windowScene.keyWindow?.addSubview(accessibleButton)
        }
    }
}
```

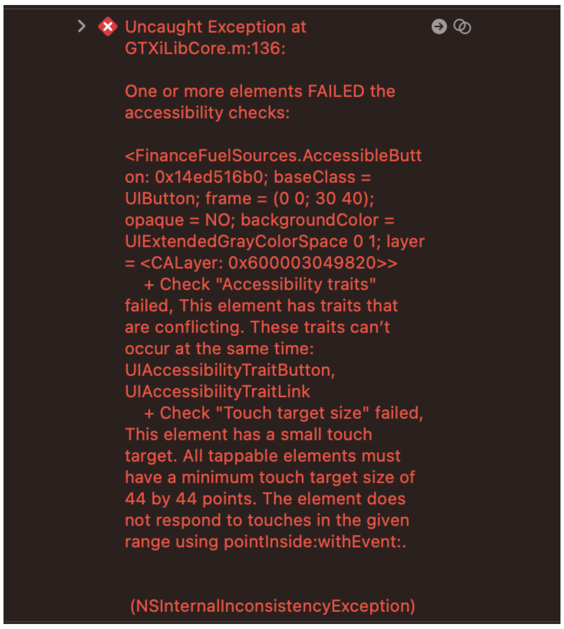
```
final class OATFinanceFuelTests: XCTestCase {
    ....

    func testButton() {
        checkAccessibility(accessibleButton)
    }
}
```

Overall, 8 issues are indicated by OAT Figure 14 and only 2 by GTXiLib Figure 15.



```
FinanceFuelTests 8 issues
  OATtests
    testButton(): failed -
     -> 'accessibilityHint' should begin with a capitalized word. This helps VoiceOver
    read the hint with the appropriate inflection.
    testButton(): failed -
     -> 'accessibilityHint' should end with a period.
    testButton(): failed -
     -> 'accessibilityHint' should not include the type of the control or view. Please,
    substitute the following words ["button", "link"].
    testButton(): failed -
     -> 'accessibilityLabel' is not a sentence and therefore should not end with a period.

    testButton(): failed -
     -> 'accessibilityLabel' should not include the type of the control or view, as it should
    be set in 'accessibilityTrait'. Please, substitute the following words ["button"]....
    testButton(): failed -
     -> accessibilityLabel should begin with a capitalized word. This helps VoiceOver
    read the label with the appropriate inflection....
    testButton(): failed -
     -> Contrast ratio should be at least 4.5:1 for small text (below 18 point regular or 14
    point bold)
    testButton(): failed -
     -> Touch target of an element should be at least 44x44 pt. Current heght is 40.0
    and width is 30.0
```

Figure 14. OAT testButton() report

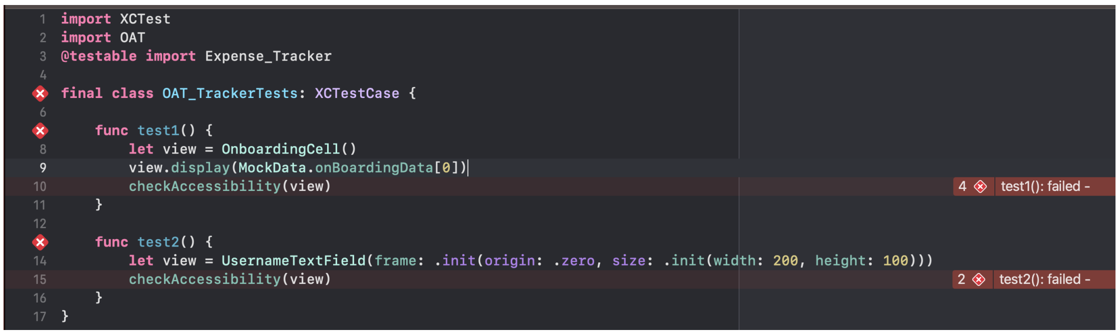


```
Uncaught Exception at
GTXiLibCore.m:136:

One or more elements FAILED the
accessibility checks:

<FinanceFuelSources.AccessibleButt
on: 0x14ed516b0; baseClass =
UIButton; frame = (0 0; 30 40);
opaque = NO; backgroundColor =
UIExtendedGrayColorSpace 0 1; layer
= <CALayer: 0x600003049820>>
   + Check "Accessibility traits"
failed, This element has traits that
are conflicting. These traits can't
occur at the same time:
UIAccessibilityTraitButton,
UIAccessibilityTraitLink
   + Check "Touch target size" failed,
This element has a small touch
target. All tappable elements must
have a minimum touch target size of
44 by 44 points. The element does
not respond to touches in the given
range using pointInside:withEvent:.


 (NSInternalInconsistencyException)
```

Figure 15. GTXiLib `testButton()` report

Additional comparison on another sample project Expense Tracker was conducted as well. The results showed that only OAT has detected accessibility faults of two randomly chosen views from the project. Overall, there are 6 issues detected by OAT and 0 by GTXiLib, on Figure 16 and Figure 17) respectively.

The accessibility issues indicated by OAT are for `UILabel` that are used in the view where one of them might be ignored by assistive technologies as it is not visible for them and has violated required line spacing, `UIImageView` which is not described (Figure 18) and `TextField` with missing keyboard type for `username` content type (Figure 19). Although there is possibility to ignore first two checks by excluding them in case if those label and image are for decorative purpose only.

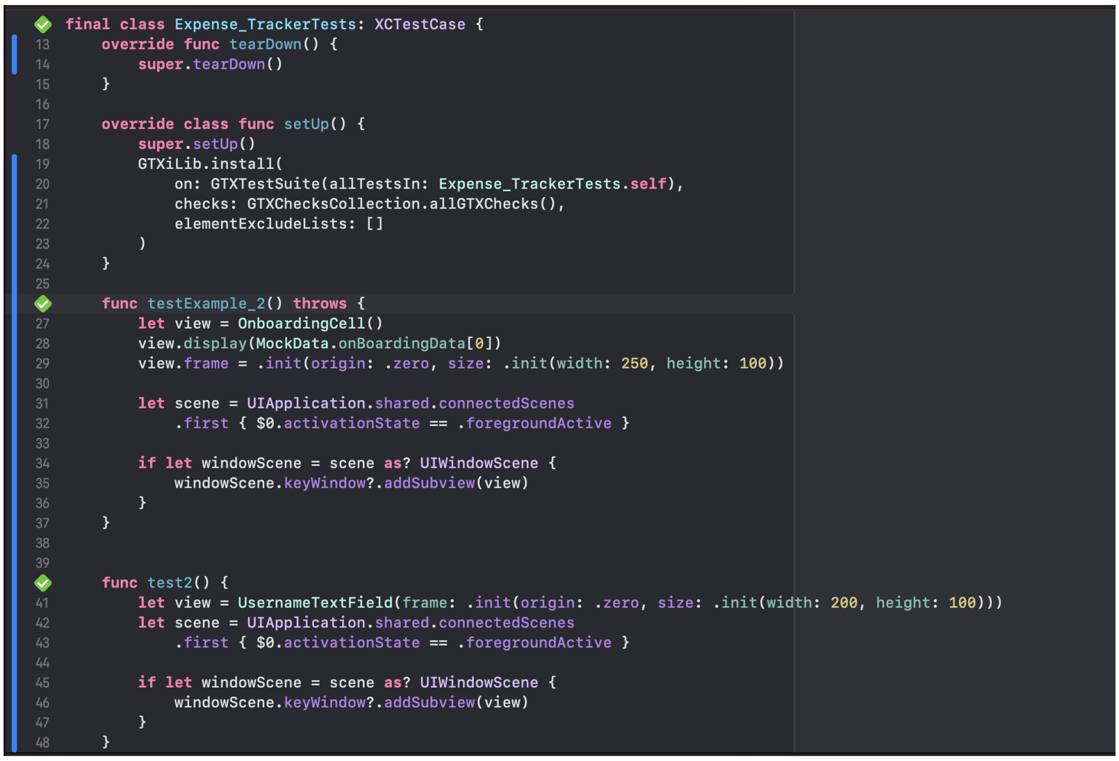


```swift
import XCTest
import OAT
@testable import Expense_Tracker

final class OAT_TrackerTests: XCTestCase {

    func test1() {
        let view = OnboardingCell()
        view.display(MockData.onBoardingData[0])
        checkAccessibility(view)                                                    4 ⊗ test1(): failed -
    }

    func test2() {
        let view = UsernameTextField(frame: .init(origin: .zero, size: .init(width: 200, height: 100)))
        checkAccessibility(view)                                                    2 ⊗ test2(): failed -
    }
}
```

Figure 16. OAT Expense Tracker tests result

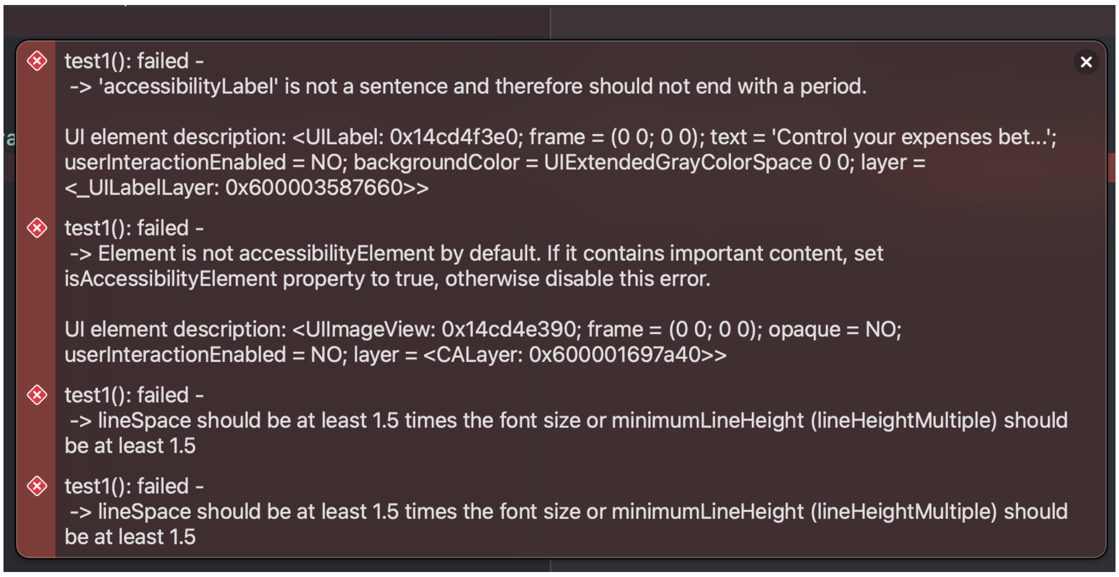
To conclude the comparison of tools in practice, the suggested accessibility checks from section 4.1, are listed in Table 2 in "Accessibility Checks" column with the addition of non-required Criterion of AAA level for target size, as it was provided in GTXiLib tool. To the right side of that column for each check the values "yes" and "no" indicates whether the tool (name in column title) includes that check. The Table 2 shows that OAT covers more accessibility issues than GTXiLib.

However, it is worth to consider that GTXiLib has the ability to check only certain rules, which is convenient when gradually introducing accessibility into a project. For example, it is possible to check only the presence of accessibility label. Although OAT also offers to exclude checks, it is implemented only for specific type of UI elements. Considering extending this feature to excluding checks by type of the rule would be useful as well.



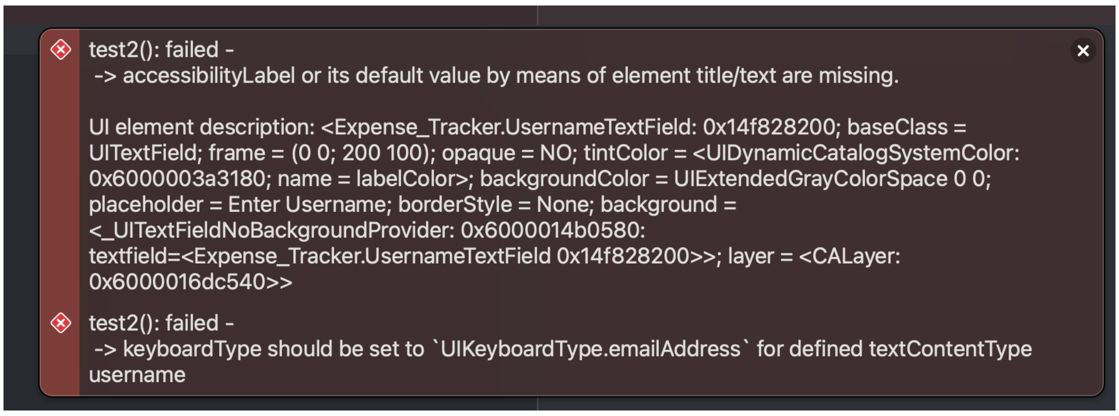
```swift
final class Expense_TrackerTests: XCTestCase {
    override func tearDown() {
        super.tearDown()
    }

    override class func setUp() {
        super.setUp()
        GTXiLib.install(
            on: GTXTestSuite(allTestsIn: Expense_TrackerTests.self),
            checks: GTXChecksCollection.allGTXChecks(),
            elementExcludeLists: []
        )
    }

    func testExample_2() throws {
        let view = OnboardingCell()
        view.display(MockData.onBoardingData[0])
        view.frame = .init(origin: .zero, size: .init(width: 250, height: 100))

        let scene = UIApplication.shared.connectedScenes
            .first { $0.activationState == .foregroundActive }

        if let windowScene = scene as? UIWindowScene {
            windowScene.keyWindow?.addSubview(view)
        }
    }


    func test2() {
        let view = UsernameTextField(frame: .init(origin: .zero, size: .init(width: 200, height: 100)))
        let scene = UIApplication.shared.connectedScenes
            .first { $0.activationState == .foregroundActive }

        if let windowScene = scene as? UIWindowScene {
            windowScene.keyWindow?.addSubview(view)
        }
    }
```

Figure 17. GTXiLib Expense Tracker tests result



```
test1(): failed -
 -> 'accessibilityLabel' is not a sentence and therefore should not end with a period.

UI element description: <UILabel: 0x14cd4f3e0; frame = (0 0; 0 0); text = 'Control your expenses bet...';
userInteractionEnabled = NO; backgroundColor = UIExtendedGrayColorSpace 0 0; layer =
<_UILabelLayer: 0x600003587660>>

test1(): failed -
 -> Element is not accessibilityElement by default. If it contains important content, set
isAccessibilityElement property to true, otherwise disable this error.

UI element description: <UIImageView: 0x14cd4e390; frame = (0 0; 0 0); opaque = NO;
userInteractionEnabled = NO; layer = <CALayer: 0x600001697a40>>

test1(): failed -
 -> lineSpace should be at least 1.5 times the font size or minimumLineHeight (lineHeightMultiple) should
be at least 1.5

test1(): failed -
 -> lineSpace should be at least 1.5 times the font size or minimumLineHeight (lineHeightMultiple) should
be at least 1.5
```

Figure 18. OAT Expense Tracker accessibility issues report 1



```
test2(): failed -
 -> accessibilityLabel or its default value by means of element title/text are missing.

UI element description: <Expense_Tracker.UsernameTextField: 0x14f828200; baseClass =
UITextField; frame = (0 0; 200 100); opaque = NO; tintColor = <UIDynamicCatalogSystemColor:
0x6000003a3180; name = labelColor>; backgroundColor = UIExtendedGrayColorSpace 0 0;
placeholder = Enter Username; borderStyle = None; background =
<_UITextFieldNoBackgroundProvider: 0x6000014b0580:
textfield=<Expense_Tracker.UsernameTextField 0x14f828200>>; layer = <CALayer:
0x6000016dc540>>

test2(): failed -
 -> keyboardType should be set to `UIKeyboardType.emailAddress` for defined textContentType
username
```

Figure 19. OAT Expense Tracker accessibility issues report 2

Table 2. Accessibility issues covered by OAT and GTXiLib testing tools

Accessibility Checks	**OAT**	**GTXiLib**
Element is accessible	yes (checks images with possibility to execlude this check)	no
Accessibility Label is not missing	yes	yes (does not check the default)
Accessibility Label is not empty	yes	yes
Accessibility Label does not include trait/type	yes	yes (only if one trait or type)
Accessibility Label first word is capitalized	yes	no
Accessibility Label does not end with period	yes	yes
Accessibility Label is not duplicated	yes	no
Accessibility Hint is not empty (when defined)	yes	no
Accessibility Hint does not include trait/type	yes	no
Accessibility Hint first word is capitalized	yes	no
Accessibility Hint ends with period	yes	no
Accessibility Hint does not contain Accessibility Label	yes	no
Accessibility Traits does not contain mutually exclusive traits	yes	yes
Contrast Ratio	yes (only button)	yes (only label)
`NSAttributedString` line(spacing) height at least 1.5 times the font size	yes	no
`NSAttributedString` paragraph spacing (spacing before) at least 2 times the font size	yes	no
`NSAttributedString` kern(tracking) at least 0.12 times the font size	yes	no
Text Input content type and keyboard type are matching	yes (for content types: email address, username, telephone number, credit card number)	no
* Target Size SC 2.5.5 (AAA) Table 4	yes (except for Auto Layout, but applies other checks)	yes (requires frame setup, for Auto Layout doesn't apply any checks)

5 Discussion

5.1 Answers to the research questions

The analysis implemented for answering **RQ1** (*What are the required accessibility Success Criteria for mobile applications? How each criterion can be tested on iOS?*) has shown that there are 44 Success Criteria for accessibility requirements of mobile applications which are mandatory by official legislation. Each of them was comprehensively explored and validated on how it can be automated in testing to detect the compliance of the app with accessibility requirements along with conformance to Human Interface Guideline provided by Apple. The results from two combined guidelines formed the fundamental accessibility checks that can be used in existing testing tools to extend their functionality or in new testing tools to create valuable products. Additionally, the outcomes taken from answering **RQ1** can be used as a base for future research in accessibility mobile applications fields.

Overall, most of the accessibility checks that can be automated are related to the essential accessibility features such as `accessibilityLabel`, `accessibilityHint`, and `accessibilityTrait`. By following Apple guidelines we formulated basic rules of which values can be set to the listed properties, as it is important for assistive technologies that use those values. For example, one of the rules is that `accessibilityLabel` should not contain UI element's `accessibilityTrait`, otherwise a screen reader will read the description for the element twice ("Submit button", "Button"), as the type of the element is set in `accessibilityTrait`. Besides basic checks, we offered contrast ratio, line spacing, touch size of controls, and other checks, which are also required Success Criteria for accessibility requirements.

For **RQ2** (*What are the existing accessibility testing tools and what are their capabilities?*) we classified all existing accessibility testing tools into three main groups based on their testing approach: linting, auditing & inspection, and automated. As it was described in the methodology 3.1.2, linting tools are not initially meant to test for accessibility, while inspection and auditing tools, dedicated specifically for this purpose, behave the same way as assistive technologies would do. However, such tools require a lot of manual work during the whole process of testing: setup, navigation, applying actions, preparing the report and etc.

We focused on automated tools which can ease the testing process and introduced into a testing target of a project. There are two types of automated testing tools: UITests and Unit tests. As UI tests are usually fragile, overlooked by projects and challenging to write and maintain, we have chosen unit testing style. Unit testing is a common approach to make fast regression testing, and usually they are already included in a project and a project's CI/CD pipelines. Thus, automated accessibility testing tools can be easily integrated into the existing test target to ensure the accessible UI/UX for users.

Further the study focused on the investigation of one tool named GTXiLib because it is the only tool that is free-of-cost, while others require paid subscriptions. During the analysis of the tool, we gained insights into its functionality and capability to detect important accessibility issues as well as its limitations. To validate it in practice, we tested it on a sample project. The results have shown that the tool has significant drawbacks. First, it uses old technical stack, which is hard to maintain and extend. Second, the results of the tests depend on the initial setup of the UI elements to frames. Third, it requires to restart the simulator after each test launch, otherwise the false positive results will be shown. Another case with incorrect results was revealed when running parallel tests in one test class and one of the test functions is checking the view with Auto Layout. Overall, the tool is very limited in terms of accessibility testing and covers only few accessibility issues.

To summarize, there is a significant need for free automated testing tools in the field of mobile applications accessibility. This gap should be filled through new development, guided by the results obtained from answering **RQ1**.

For **RQ3** (*What are the advantages of the new testing tool offered by this study over existing tools? What is the quality assurance of the tool?*), the new tool OAT with modulated architecture and the modern technical stack was presented and showed its initial advantages over the existing tools. The quality assurance of the OAT is covering all possible accessibility checks provided by the tool for both positive and negative cases. The tool showed useful results when applied to a sample project. By detecting important accessibility faults and providing descriptive reports it helped to enhance its accessibility level. Furthermore, OAT showed better results in terms of the number of detected issues during comparative analysis with GTXiLib on two sample projects. For the first project, OAT showed 4 times more relevant issues than GTXiLib (8 versus 2). Furthermore, the accessibility issues of the two views on the second project were indicated only by OAT. In conclusion, the overview of accessibility checks available on both tools showed that OAT covers a broader range of important issues.

Considering that this testing tool is an open-source library, the open-source community can contribute to this project and extend its functionalities if necessary. OAT is a great hands-on example of an automated testing tool and this experience can be used and adopted by others.

5.2 Limitations of the study

The following limitations in this research should be considered:

1. Some of the Success Criteria was not automated by this research, as most of them require context and mental model understanding that was presented during the analysis of Criteria.

2. The next limitation is that only one existing automated accessibility testing tool was analysed as others are requiring buying subscriptions.

3. Overall, there are two frameworks for creating UI in iOS: `UIKit` and `SwiftUI`. Sample projects and the offered testing tool OAT are written with `UIKit` framework consideration only.

6 Threats to validity

It would be relevant to acknowledge the threats to validity while acquiring the survey results from the analysis of GTXiLib and its comparison with OAT. Since both projects are open-source, the former might be updated with extended functionality or improvements. In that case, the results shown in this study might get outdated.

7 Conclusion and future work

This research aimed to enhance the accessibility level of iOS mobile applications by developing an automated testing tool based on the analysed accessibility guidelines.

To reach the goal, first, we conducted a comprehensive analysis of guidelines and concluded which Success Criteria should be considered in the automated solution. Second, we investigated already existing solutions on the market and assessed the one of them which is the only available for free with regards to the Success Criteria and technical aspects. The results have shown disadvantages of the existing solution, which should be avoided in a new tool. Finally, we developed a new open-source automated accessibility tool called OAT (Open Accessibility Tool). It uses the modulated architecture and the latest technical stack, which guarantee necessary flexibility and possibility for extending existing functionalities. The quality of the tool is assured by exhaustive test code coverage for all its accessibility checks. By detecting important accessibility faults and providing descriptive report, it helped to enhance the sample projects accessibility level. Furthermore, it showed better results in terms of number of detected issues compared to previously analyzed existing tool.

For further work, the following suggestions could be implemented. One possible improvement would be to extend the tool to another UI framework from Apple Inc `SwiftUI` since the current research considered only one framework `UIKit` for building UI in both new tool and sample projects. Another suggestion is to test the tool on more projects to determine some corner cases and possibly expand functionality. In terms of theoretical work, the same analysis as for answering **RQ1** could be implemented for the rest of accessibility requirements that are non-mandatory by the law. Having compliance with non-required criteria would only increase the quality of user experience.

8 Acknowledgements

I would like to express gratitude to my supervisor Ishaya Peni Gambo for all his help and guidance throughout the whole process.

References

[1] Sen Chen et al. "Accessible or Not? An Empirical Investigation of Android App Accessibility". In: *IEEE Transactions on Software Engineering* 48.10 (2022), pp. 3954–3968. DOI: 10.1109/TSE.2021.3108162.

[2] João Dias et al. "Identifying Concerns On Automatic Tools For Accessibility Assessment: A Comparative Case Study On Two Portuguese Universities' Websites". In: *2022 5th International Conference on Information and Computer Technologies (ICICT)*. 2022, pp. 1–6. DOI: 10.1109/ICICT55905.2022.00008.

[3] Marcelo Medeiros Eler et al. "Automated Accessibility Testing of Mobile Apps". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 116–126. DOI: 10.1109/ICST.2018.00021.

[4] ETSI. *EN 301 549*. URL: https://www.etsi.org/deliver/etsi_en/301500_301599/301549/03.02.01_60/en_301549v030201p.pdf.

[5] Google. *GTXiLib: Google Toolbox for Accessibility for iOS*. URL: https://github.com/google/GTXiLib.

[6] Apple Inc. *Human Interface Guidelines | Apple Developer Documentation*. URL: https://developer.apple.com/design/human-interface-guidelines.

[7] Apple Inc. *Technologies | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/technologies.

[8] Apple Inc. *Testing for Accessibility on OS X*. URL: https://apple.co/3QC6KoS.

[9] J Jokinen. "Implementing web accessibility to an existing web application". PhD thesis. Master's Thesis in Technology. Software Engineering. Department of Future . . ., 2020.

[10] Robin Kanatzar. *Testing for Accessibility*. 2022. URL: https://speakerdeck.com/robinkanatzar/testing-for-accessibility-frenchkit-2022.

[11] Sidrah Kashif et al. "Development of An Automated Accessibility Evaluation Plugin Tool for Mobile Applications". In: *2022 3rd International Conference on Innovations in Computer Science Software Engineering (ICONICS)*. 2022, pp. 1–10. DOI: 10.1109/ICONICS56716.2022.10100578.

[12] World Health Organization. *Disability*. URL: https://www.who.int/health-topics/disability.

[13] Eunju Park et al. "Development of Automatic Evaluation Tool for Mobile Accessibility for Android Application". In: *2019 International Conference on Systems of Collaboration Big Data, Internet of Things Security (SysCoBIoTS)*. 2019, pp. 1–6. DOI: 10.1109/SysCoBIoTS48768.2019.9028034.

[14] Kristiina Rahkema, Dietmar Pfahl, and Rudolf Ramler. “Analysis of Library Dependency Networks of Package Managers Used in iOS Development”. In: *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2023, pp. 23–27. DOI: `10.1109/MOBILSoft59058.2023.00010`.

[15] Realm. *SwiftLint*. URL: `https://github.com/realm/SwiftLint`.

[16] European Union. *Directive (EU) 2019/882 of the European Parliament and of the Council of 17 April 2019 on the accessibility requirements for products and services (Text with EEA relevance)*. URL: `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32019L0882`.

[17] W3C. *World Wide Web Consortium (W3C)*. URL: `https://www.w3.org/`.

[18] Shunguo Yan and PG Ramachandran. “The current status of accessibility in mobile apps”. In: *ACM Transactions on Accessible Computing (TACCESS)* 12.1 (2019), pp. 1–31.

Table 3. Perceivable Principle

Guideline	**Success criterion**
1.1 Text Alternatives	1.1.1 Non-text Content (A)
1.2 Time-based Media	1.2.1 Audio-only and Video-only (Prerecorded) (A) 1.2.2 Captions (Prerecorded) (A) 1.2.3 Audio Description or Media Alternative (Prerecorded) (A) 1.2.4 Captions (Live) (AA) 1.2.5 Audio Description (Prerecorded) (AA) 1.2.6 Sign Language (Prerecorded) (AAA) 1.2.7 Extended Audio Description (Prerecorded) (AAA) 1.2.8 Media Alternative (Prerecorded) (AAA) 1.2.9 Audio-only (Live) (AAA)
1.3 Adaptable	1.3.1 Info and Relationships (A) 1.3.2 Meaningful Sequence (A) 1.3.3 Sensory Characteristics (A) 1.3.4 Orientation (AA) 1.3.5 Identify Input Purpose (AA) 1.3.6 Identify Purpose (AAA)
1.4 Distinguishable	1.4.1 Use of Color (A) 1.4.2 Audio Control (A) 1.4.3 Contrast (Minimum) (AA) 1.4.4 Resize text (AA) 1.4.5 Images of Text (AA) 1.4.6 Contrast (Enhanced) (AAA) 1.4.7 Low or No Background Audio (AAA) 1.4.8 Visual Presentation (AAA) 1.4.9 Images of Text (No Exception) (AAA) 1.4.10 Reflow (AAA) 1.4.11 Non-text Contrast (AA) 1.4.12 Text Spacing (AA) 1.4.13 Content on Hover or Focus (AA)

Table 4. Operable Principle

Guideline	**Success criterion**
2.1 Keyboard Accessible	2.1.1 Keyboard (A) 2.1.2 No Keyboard Trap (A) 2.1.3 Keyboard (No Exception) (AAA) 2.1.4 Character Key Shortcuts (A)
2.2 Enough Time	2.2.1 Timing Adjustable (A) 2.2.2 Pause, Stop, Hide (A) 2.2.3 No Timing (AAA) 2.2.4 Interruptions (AAA) 2.2.5 Re-authenticating (AAA) 2.2.6 Timeouts (AAA)
2.3 Seizures and Physical Reactions	2.3.1 Three Flashes or Below Threshold (A) 2.3.2 Three Flashes (AAA) 2.3.3 Animation from Interactions (AAA)
2.4 Navigable	2.4.1 Bypass Blocks (A) 2.4.2 Page Titled (A) 2.4.3 Focus Order (A) 2.4.4 Link Purpose (In Context) (A) 2.4.5 Multiple Ways (AA) 2.4.6 Headings and Labels (AA) 2.4.7 Focus Visible (AA) 2.4.8 Location (AAA) 2.4.9 Link Purpose (Link Only) (AAA) 2.4.10 Section Headings (AAA)
2.5 Input Modalities	2.5.1 Pointer Gestures (A) 2.5.2 Pointer Cancellation (A) 2.5.3 Label in Name (A) 2.5.4 Motion Actuation (A) 2.5.5 Target Size (AAA) 2.5.6 Concurrent Input Mechanisms (AAA)

Table 5. Understandable Principle

Guideline	**Success criterion**
3.1 Readable	3.1.1 Language of Page (A) 3.1.2 Language of Parts (AA) 3.1.3 Unusual Words (AAA) 3.1.4 Abbreviations (AAA) 3.1.5 Reading Level (AAA) 3.1.6 Pronunciation (AAA)
3.2 Predictable	3.2.1 On Focus (A) 3.2.2 On Input (A) 3.2.3 Consistent Navigation (AA) 3.2.4 Consistent Identification (AA) 3.2.5 Change on Request (AAA)
3.3 Input Assistance	3.3.1 Error Identification (A) 3.3.2 Labels or Instructions (A) 3.3.3 Error Suggestion (AA) 3.3.4 Error Prevention (Legal, Financial, Data) (AA) 3.3.5 Help (AAA) 3.3.6 Error Prevention (All) (AAA)

Table 6. Robust Principle

Guideline	**Success criterion**
4.1 Compatible	4.1.1 Parsing (A) 4.1.2 Name, Role, Value (A) 4.1.3 Status Messages (AA)

Table 7. Dependency managers for iOS projects (brief overview)

	CocoaPods	**Carthage**	**Swift Package Manager (SPM)**
Info	Centralized dependency manager, it has a single repository that stores all pods	Decentralized manager, there is no single repository for finding dependencies.	Native decentralized manager from Apple, there is no single repository for finding dependencies.
Setup	Have to be set up and updated manually on a target system		No setup required, already integrated in Xcode and ready for using.
Language	Written in Ruby and uses Ruby to describe all dependencies.	Written in Swift and uses Swift to describe all dependencies	
Compiling	All dependencies have to be compiled separately into frameworks, then they can be reused by Xcode in the application.		Compiling automatically when it is necessary, then it's kept in a cache.
Advantages	Easy to integrate, some dependencies support only CocoaPods.	Very flexible, and can be used in very different configurations. Does not affect '.xcodeproj' and '.xcworkspace' files.	Integration or any setup isn't needed at all, already integrated into Xcode and always ready for isung.
Disadvantages	Quite inflexible, '.xcworkspace' file is generated automatically and cannot be changed. Does some magic under the hood to link dependencies in the project which potentially can lead to unexpected behavior in some corner cases.	Requires an extra mile to integrate to a project (one more script in build settings).	Some dependencies may have not yet built-in SPM support.

Appendix

I. Glossary

- UI - user interface
- RQ - research question
- app - application
- UI - user interface
- UX - user experience
- SC - Success Criteria(-on)
- SPM - Swift Package Manager

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public