Санкт-Петербургский государственный университет

Направление «Математическое обеспечение и администрирование информационных систем»
Кафедра системного программирования

Евсеев Олег Александрович

Распознавание элементов пользовательского интерфейса на изображениях

Курсовая работа

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Санкт-Петербург
2018

Содержание

1. Введение **3**

2. Обзор **5**

2.1. Постановка задачи 5

2.2. Существующие подходы 6

2.3. Особенности реализации 7

3. Реализация **8**

3.1. Архитектура 8

3.2. Поиск шаблона 10

3.3. Поиск кнопок на изображении 11

3.4. Поиск надписей на изображении 14

3.5. Алгоритм валидации 14

3.6. Случайные тесты 15

3.7. Ручные тесты 16

4. Заключение **20**

Список литературы **21**

1 Введение

В разработке качественного программного продукта тестирование играет одну из самых важных ролей. Во-первых, с внедрением в процесс автоматического тестирования возрастает продуктивность разработки. Во-вторых, увеличивается качество продукта. В-третьих, сокращается время, затрачиваемое на верификацию. Также стоит заметить, что средства автоматизации тестирования позволяют повторно использовать тесты при корректировке ПО.

Изменения в продукты вносятся постоянно, и из-за этого становится сложно отследить всё, не имея перед собой специальных инструментов. Наибольшие успехи были достигнуты в сфере веб-разработки, а фреймворк Selenium WebDriver [7] стал де-факто стандартом для автоматического тестирования веб-приложений.

Однако, для нативных приложений автоматическое тестирование не менее важно. В JetBrains для этих целей используется GUI Automation Framework, позволяющий выполнять автоматизированные действия над компонентами пользовательского интерфейса, наподобие тех, что позволяет делать Selenium WebDriver и другие подобные ему технологии.

Но ни одна из этих технологий не позволяет полностью охватить весь спектр взаимодействий пользователя с интерфейсом, поэтому часть работ все равно ложится на плечи живых тестировщиков. Одной из таких задач является проверка «корректности» отрисовки интерфейса в зависимости от целевой платформы, выбранной локали, размеров окна и прочих факторов: элементы могут выходить за границы окна, перекрываться другими элементами или быть тем или иным образом продеформированы (может быть изменен их размер, цвет, сокращен текст внутри и прочее).

В таких случаях кажется разумным снять скриншот формы (или, если это позволяет UI-фреймворк, отрисовать ее средствами самого фреймворка), для которой нужно проверить правильность отрисовки, и научиться находить на нем элементы UI, для которых точно известно, что они должны быть видны на экране, а также проверить найденные элементы на то, отображаются ли они согласно указанным свойствам.

Если такой функционал будет реализован в виде фреймворка, легко расширяемого путем введения новых подходов, позволяющих искать элементы во все более общих и общих случаях, он легко найдет применение в сфере автоматического тестирования. В данной курсовой рассматривается задача реализации такого функционала для стандартного для Java-приложений фреймворка Swing.

2 Обзор

2.1 Постановка задачи

Цель проекта — реализация фреймворка, позволяющего избавить тестировщиков JetBrains от рутинных задач по проверке соответствия отрисовке формы тому, как она описана в коде.

Общими требованиями к тому, чтобы подобный функционал был интегрирован в GUI Automation Framework, были следующие требования:

- Чтение свойств GUI-элемента приложения, позволяющих идентифицировать его на скриншоте (таких, как, например: тип элемента, наличие текста, взаимное расположение относительно соседних элементов и прочее);
- Захват скриншота приложения;
- Проверка согласованности свойств элемента на скриншоте свойствам, прописанным в коде (в частности, проверить, не выходит ли элемент за границу окна, не скрыт ли он за другим элементом, соответствует ли ему заданный текст и т.д.);
- Сравнение взятого скриншота с эталонным (опционально).

В рамках данной курсовой работы была поставлена задача реализовать минимальный рабочий прототип такой системы (работающий для кнопок и надписей для форм в цветовой схеме Darcula, используемой в продуктах JetBrains).

То есть, для заданной формы, о которой мы точно знаем, что при правильном рендеринге на ней должны присутствовать определенные элементы, нужно найти эти элементы на скриншоте формы с помощью средств компьютерного зрения.

Качество распознавания элементов на форме подлежит оценке. Для этого в процессе работы над данной курсовой работой было решено построить генератор соответствующих задаче форм. При генерации формы увеличивались до предпочтительных размеров для того, чтобы все элементы гарантированно отображались правильно. Основным критерием качества было

значение recall (полнота). Пусть tp (true positives) — количество элементов, которые мы нашли на изображении, а fn (false negatives) — количество элементов, которые мы найти не смогли.

$$\text{recall} = \frac{tp}{tp + fn} \tag{1}$$

Полученный результат было необходимо оформить в виде архитектуры классов, удобной как для использования извне в тестовых сценариях, так и для расширения путем добавления новых подходов к детектированию элементов UI.

2.2 Существующие подходы

Применение скриншотов в автоматическом тестировании в основном ограничено сценарием «проверить, насколько сильно изменился эталонный скриншот». [10]

Поставленная задача похожа на задачу object detection компьютерного зрения. Вариаций этой задачи существует огромное множество (в зависимости от того, какого типа объекты мы ищем; наиболее популярные применения — это распознавание лиц и распознавание пешеходов). Объединяет эти подходы то, что в качестве оценки обычно применяется *mean average precision* (mAP). [6]

Однако, так как мы точно знаем, что элемент содержится на скриншоте, и существуют способы находить элементы с пренебрежимо низким количеством ложноположительных результатов, для таких форм нам достаточно считать только recall, и эту метрику мы и должны улучшать.

Более адекватную оценку качества мы могли бы получить, если бы у нас имелись элементы, которые мы бы гарантированно *не* находили бы на изображении (например, кнопки, гарантированно вылезающие за границы окна). Однако тесты для таких случаев в силу их относительной сложности пришлось бы генерировать вручную. Поэтому наша задача состоит в том, чтобы улучшать результат на «хороших» формах.

2.3 Особенности реализации

Работа велась с фреймворком Swing. Его особенность в том, что все отрисовывается платформонезависимо и непосредственно во время исполнения приложения, то есть скриншот можно получить двумя путями:

- Скриншот можно снять с помощью метода `createScreenCapture` класса `java.awt.Robot`;
- Можно вручную отрисовать форму с помощью метода `paint(java.awt.Graphics)` класса `java.awt.Component`.

Метод `paint` отрисовывает форму на объекте типа `java.awt.Graphics`, то есть мы имеем возможность получать скриншоты в любом нужном нам разрешении, так как имеем дело с векторной графикой. Высокое разрешение изображений требуется для корректной работы с Tesseract — фреймворком для распознавания текста на изображениях.

Зная о том, что все компоненты формы должны в текущий момент находиться на экране, гарантию правильного рендеринга получаем от метода `pack()` класса `java.awt.Window`, который выставляет размеры компонентов на основе предпочтительных размеров самих компонентов и потомков, соответствующих им.

Заметим, что мы можем найти несколько потенциальных кандидатов на роль нужного нам объекта (такая ситуация возникает, например, когда у нас имеются несколько кнопок с одинаковым текстом). В таком случае нам придется использовать дополнительные отсечения, позволяющие однозначно идентифицировать нужный нам компонент. Метрика, используемая для оценки качества, также должна быть немного более сложной, чем простой подсчет recall. В рамках данной курсовой работы мы пока не разделяем эти случаи.

3 Реализация

Код проекта: `https://gitlab.com/oevseev/ui-recognition`.

3.1 Архитектура

Основным классом, отвечающим непосредственно за сам механизм валидации, является класс `WindowValidator` с конструктором, принимающим на вход `javax.swing.RootPaneContainer` (интерфейс Swing для контейнеров верхнего уровня: `JWindow`, `JDialog` и `JPanel`) и имеющий методы `validate(java.awt.BufferedImage)` и `getBestConfiguration()`, позволяющие найти набор ограничивающих прямоугольников, наилучшим образом описывающий набор компонентов UI.

Метод `validate` реализует простой алгоритм перебора с возвратом, сохраняющий информацию о прямоугольниках, назначенных каждому из компонентов.

Поиск кандидатов осуществляется конкретными реализациями класса `ComponentMatcher`. Были реализованы подходы, позволяющие искать `JButton` и `JLabel` при установленном оформлении Darcula, стандартном для продуктов JetBrains, для случая простых деформаций (изменение размера, начертания, положения, сокращение текста внутри).

Так как внешний вид Swing-приложения определяется тем, какой объект класса `javax.swing.LookAndFeel` был установлен в статическом классе `javax.swing.UIManager`, отвечающим за глобальные настройки UI, было решено поручить создание конкретных экземпляров `ComponentMatcher` иерархии абстрактных фабрик `ComponentMatcherFactory`. Был также реализован прямой потомок этого класса `SwingComponentMatcherFactory`, содержащий фабричные методы, соответствующие терминальным неабстрактным компонентам Swing. Для удобства тестирования был введен вспомогательный интерфейс `ValidatableComponent`: реализующий его компонент вправе объявить, при помощи какого `ComponentMatcher` его стоит искать на скриншоте. В `WindowValidator` помимо возможности установить конкретную фабрику для конкретного экземпляра валидатора был добавлен статический метод `setFactoryPrototype(ComponentMatcherFactory)`, поз-

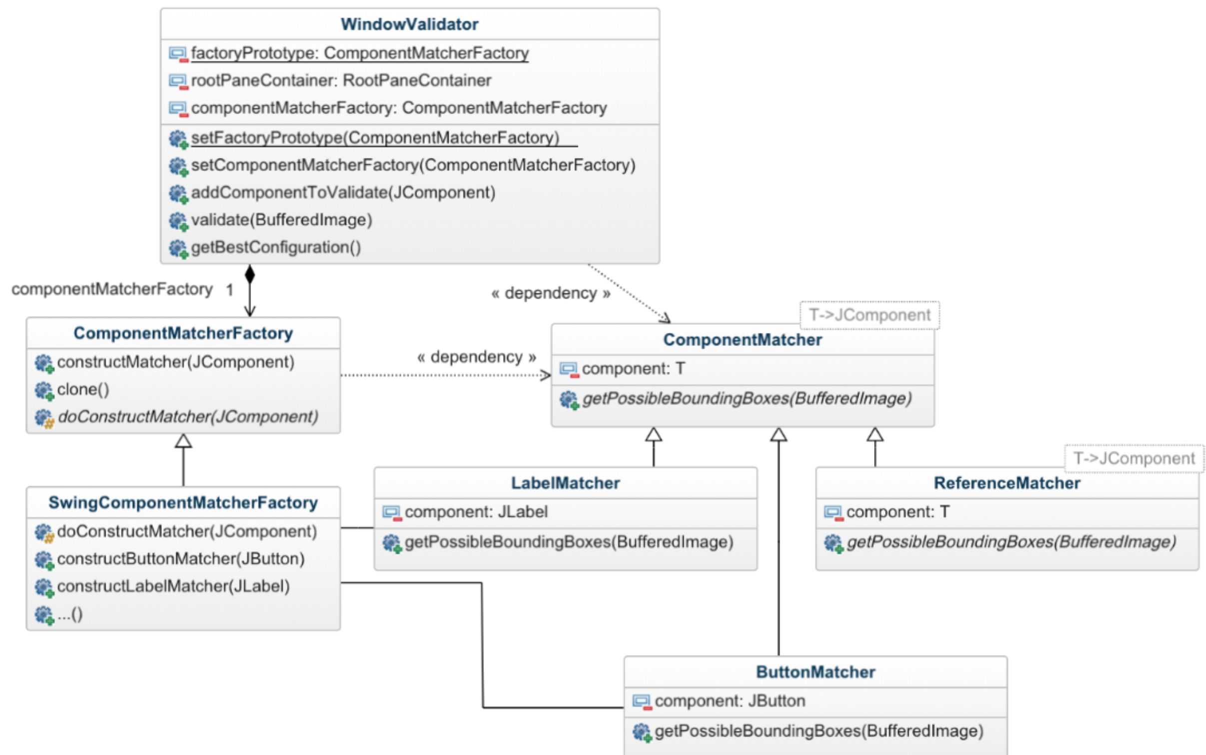


Рис. 1: Иерархия классов

воляющий установить фабрику по умолчанию (сама же фабрика реализует шаблон «прототип»).

Для тестирования на случайно сгенерированных данных применяются классы `RandomForm`, генерирующий случайную Swing-форму, и исполняемый класс `RandomTestPipeline`, позволяющий оценить качество валидации на некотором наборе из случайно сгенерированных форм.

Основными внешними библиотеками, применяемые в данной задаче, были OpenCV (алгоритмы компьютерного зрения и обработки изображений) [5] и Tesseract (библиотека для распознавания текста) [9]. Так как обе библиотеки изначально написаны преимущественно на C++, использовались их Java-байндинги на основе JavaCPP [4].

Так как для запуска Tesseract нужно инстанцировать объект класса `TessBaseAPI` и проинициализировать его параметрами, которые не будут изменяться в процессе выполнения программы (в частности, местоположением директории `tessdata`, содержащей модели), было решено создать класс `LocalTesseract`, реализующий паттерн «одиночка» и предоставля-

ющий доступ к вызовам Tesseract.

Паттерны «абстрактная фабрика», «фабричный метод», «одиночка» и «прототип» изначально описаны в книге «Паттерны проектирования» [3].

3.2 Поиск шаблона

Наиболее простым способом найти элемент на изображении является проход по нему скользящим окном и подсчет значения взаимнокорреляционной (cross-correlation) функции для шаблона и текущего окна [8]. Более конкретно, пусть T — матрица шаблона, а I — матрица изображения. Тогда *нормированная взаимнокорреляционная функция* для одноканального изображения определяется следующим образом:

$$R(x,y) = \frac{\sum_{x',y'} T(x',y') \cdot I(x+x',y+y')}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}} \qquad (2)$$

В случае, если изображение цветное, $R(x,y)$ вычисляется для каждого из каналов, а их значения усредняются.

Если в качестве шаблона мы будем использовать изображение компонента, отрисованного в условиях, когда он находится вне иерархии компонентов, куда он помещен, этот подход будет работать только в том случае, если внешний вид компонента не сильно отличается от его идеализированного представления. Также это будет требовать создания копии конкретного компонента. Так как `JComponent` не реализует функционал прототипа, копирование всех свойств объекта придется производить вручную.

Если же не «вытаскивать» компонент из иерархии, а отрисовать его «как есть», то, в силу того, что метод `paint(java.awt.Graphics)` вызывается рекурсивно для всех потомков данного компонента в дереве компонентов, **не** найти компонент на скриншоте будет возможно лишь в том случае, если он выходит за границы родительского компонента или перекрывается другим компонентом. Все некорректности отрисовки, виной которых — сами свойства компонента, будут считаться штатной ситуацией, что не всегда то, что нам требуется.

Соответствующий функционал оформлен в классе `ReferenceMatcher`.

3.3 Поиск кнопок на изображении

Для того, чтобы найти кнопки на изображении, воспользуемся тем фактом, что в общем виде они представляют собой прямоугольники с текстом внутри.

Рис. 2: Типичная кнопка

Для поиска прямоугольников в компьютерном зрении обычно используется обобщенное преобразование Хафа [1]. Однако, использование его для компьютерно-сгенерированных изображений несколько неоправдано на практике: преобразование Хафа обычно используется для поиска объектов нужной формы на фотографиях, а в нашем случае, например, мы точно знаем, что стороны прямоугольника всегда параллельны краям окна.

Перед тем, как прийти к подходу, имеющемуся сейчас, было протестировано два вида преобразования Хафа: для поиска отрезков на изображении и для поиска прямоугольников на изображении. Оба метода дали неудовлетворительный результат, так как иногда не находили прямоугольники.

Было решено прибегнуть к операциям `erosion` и `dilation`, изменяющим бинарное изображение на основе следующих принципов:

- `erosion` проходит структурным элементом по изображению при помощи метода скользящего окна и устанавливает соответствующий пиксель результирующего изображения в 1, только если все пиксели, входящие в окно в исходном изображении, равны 1.

- `dilation` рисует копию структурного элемента на результирующем изображении с центром в соответствующем пикселе в том случае, если этот пиксель на исходном изображении равен 1.

В результате в классе `ButtonValidator` реализован следующий комбинированный подход:

- К исходному изображению применяется алгоритм адаптивной бинаризации (`adaptiveThresholding` в OpenCV).

- К бинаризованному изображению применяются последовательно сначала erosion, а затем dilation — сначала с ядром $(1, 15)$, а затем $(15, 1)$. Это нужно для того, чтобы найти горизонтальные и вертикальные линии на экране. Полученные изображения объединяются с помощью побитового ИЛИ.

- Снова применяется `dilation` — на этот раз с ядром $(3, 3)$ для того, чтобы компенсировать скругленные углы на кнопках и замкнуть соответствующие контуры.

- При помощи метода `findContours` OpenCV на изображении ищутся контуры. В нашем случае будут найдены замкнутые прямоугольные элементы. Так как `findContours` сохраняет информацию о вложенности контуров друг друга, мы можем использовать ее, чтобы найти только внутренние прямоугольники (пока мы работаем только с терминальными элементами).

- Для каждого контура считается ограничивающий его прямоугольник. Прямоугольники, по площади меньшие определенного порога (200 пикселей) отсекаются. Это сделано для того, чтобы в качестве потенциальных кандидатов не появлялись прямые, которые могут остаться после применения морфологических операций к тексту; например, вертикальная прямая в букве Т, если та достаточно большая, вертикальные и горизонтальные линии, визуально разделяющие элементы интерфейса, и другие элементы, которые в силу своих размеров не могут содержать внутри себя читаемый текст.

- Текст, принадлежащий каждому прямоугольнику, распознается с помощью системы оптического распознавания символов (OCR) Tesseract и проверяется на соответствие данным, заданным в коде.

Особенностью реализации является то, что Tesseract натренирован на то, чтобы распознавать текст в изображениях с высоким DPI, поэтому мы должны записать в метаданные валидируемого изображения, что оно имеет высокий DPI. Если это действительно так (например, мы используем отрисованную в высоком разрешении форму), то никаких дополнительных действий не требуется, иначе же мы можем воспользоваться любой интерполяцией, которая качественно работает с изображениями, преимущественно заполненными контурными объектами (например, фильтр Ланцоша [2]), и увеличить его разрешение.

Отдельно рассмотрим случаи, когда текст не вмещается целиком внутрь компонента; в этих случаях обычно отображается какое-то количество первых и последних символов, а прочие символы заменяются многоточием. Например, в случае недостаточного размера кнопки текст «Text» внутри нее может превратиться в «T...t». По этой причине мы будем сравнивать не строки целиком, а их префиксы и суффиксы, и считать, что компонент прошел валидацию, если префикс до троеточия и суффикс до многоточия строки, полученной с изображения, равны префиксу и суффиксу строки из кода (разумеется, учитывая, что эти префикс и суффикс не должны пересекаться).

Рис. 3: Пример поиска кнопки

3.4 Поиск надписей на изображении

В Tesseract уже реализована сегментация текста на изображении, но лишь в четырех режимах: отдельные символы, слова, строки текста и абзацы.

Поиск строк ищет строки, занимающие всю ширину изображения, что нам не очень подходит: нас интересуют группы рядом стоящих слов. Для того, чтобы склеить слова, используем `dilation` с горизонтальным структурным элементом, ширина которого должна быть больше интервала между словами, но такой, чтобы случайно не склеить две соседние надписи.

Распознанные блоки текста распознаем при помощи Tesseract. Если нам нужно, можем игнорировать ошибки распознавания, опираясь на расстояние Хаффмана или Левенштейна между искомой надписью и распознанным текстом, однако это способно вызвать ложноположительные срабатывания.

3.5 Алгоритм валидации

Пусть для каждого компонента мы нашли некоторый список ограничивающих прямоугольников-кандидатов. Нам необходимо найти такую конфигурацию, чтобы ни один прямоугольник не был использован дважды и при этом число найденных элементов было максимальным.

Простейшим из таких алгоритмов для такой задачи является перебор с возвратом, который был реализован сначала.

После проведения случайного тестирования оказалось, что достаточно часты случаи, когда на форме удается найти все элементы, кроме одного двух. Поскольку перебор с возвратом возвращает только такие конфигурации, в которых каждому элементу управления удалось сопоставить кандидата, поиск конфигурации оканчивается безрезультатно.

В итоге перебору с возвратом был предпочтен полный перебор, а для каждого элемента был добавлен вариант вообще не найти кандидата. В случае, когда конфигураций несколько, выбирается та, которая максимизирует количество прямоугольников, совпадающих с ограничивающими прямоугольниками, рассчитанными во время исполнения для формы-эталона,

а в случае нескольких таких конфигураций — дающая наибольшее среднее IoU для всех таких пар[1].

При переборе вариантов для проверки того, что прямоугольники R_1 и R_2 не совпадают и не вложены друг в друга, использовалась следующая эмпирическая формула (где S_R — площадь прямоугольника R):

$$\max\left\{\frac{S_{R_1\cap R_2}}{S_{R_1}}, \frac{S_{R_1\cap R_2}}{S_{R_2}}\right\} < 0.5 \tag{3}$$

В случае, если данное условие истинно, это значит, что прямоугольники различны. Такое условие отсекает случаи, когда прямоугольники совпадают либо вложены в друг в друга и не отсекает случаи, когда границы прямоугольников частично залезают друг на друга.

3.6 Случайные тесты

Случайное дерево из n вершин можно равновероятно сгенерировать при помощи генерации кода Прюфера и восстановления дерева из него [11].

На основе дерева можно построить форму следующим образом:

- Листьям дерева соответствуют терминальные компоненты.
- Нетерминальным вершинам соответствует `JPanel` с одним из четырех стандартных для Swing менеджеров расположения элементов: `BoxLayout` с `X_AXIS`, `BoxLayout` с `Y_AXIS`, `FlowLayout` и `GridLayout`. При этом `GridLayout` мы разбиваем число детей на произведение двух сомножителей (количество строк и количество столбцов) случайным образом, чтобы не было пустых ячеек.
- Чтобы не было такого, что единственный `JPanel` вложен в `JPanel`, просто при обходе не создаем вложенный `JPanel`, если число детей у соответствующей вершин равно 1, а используем компонент, соответствующий этому ребенку.

На сгенерированной форме вызываем `pack`, чтобы она приняла размер, гарантирующий корректную отрисовку. Проверить, корректно ли мы

[1] отношение площади пересечения двух прямоугольников к площади их объединения

распознали элемент, можем, посчитав IoU[2] границ элемента относительно контейнера верхнего уровня и найденного прямоугольника: если $IoU > 0.5$, это значит, что мы нашли объект корректно.

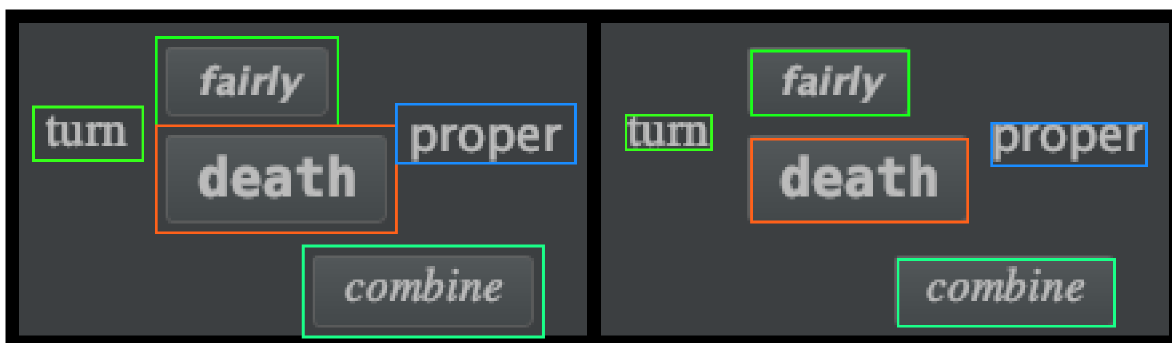


Рис. 4: Пример работы алгоритма на случайной форме. Слева — истинные границы элементов, справа — найденные при помощи описанных подходов

Были проведены эксперименты по подсчету среднего recall на 50 наборах по 20 случайно сгенерированных форм, где формы строились на основе деревьев размером до 20 вершин:

Метод	**avg. recall ± stdev**
Без увеличения разрешения	$72.6 \pm 4.0\%$
С увеличением разрешения	$84.6 \pm 4.4\%$
С векторной отрисовкой	$85.9 \pm 4.5\%$

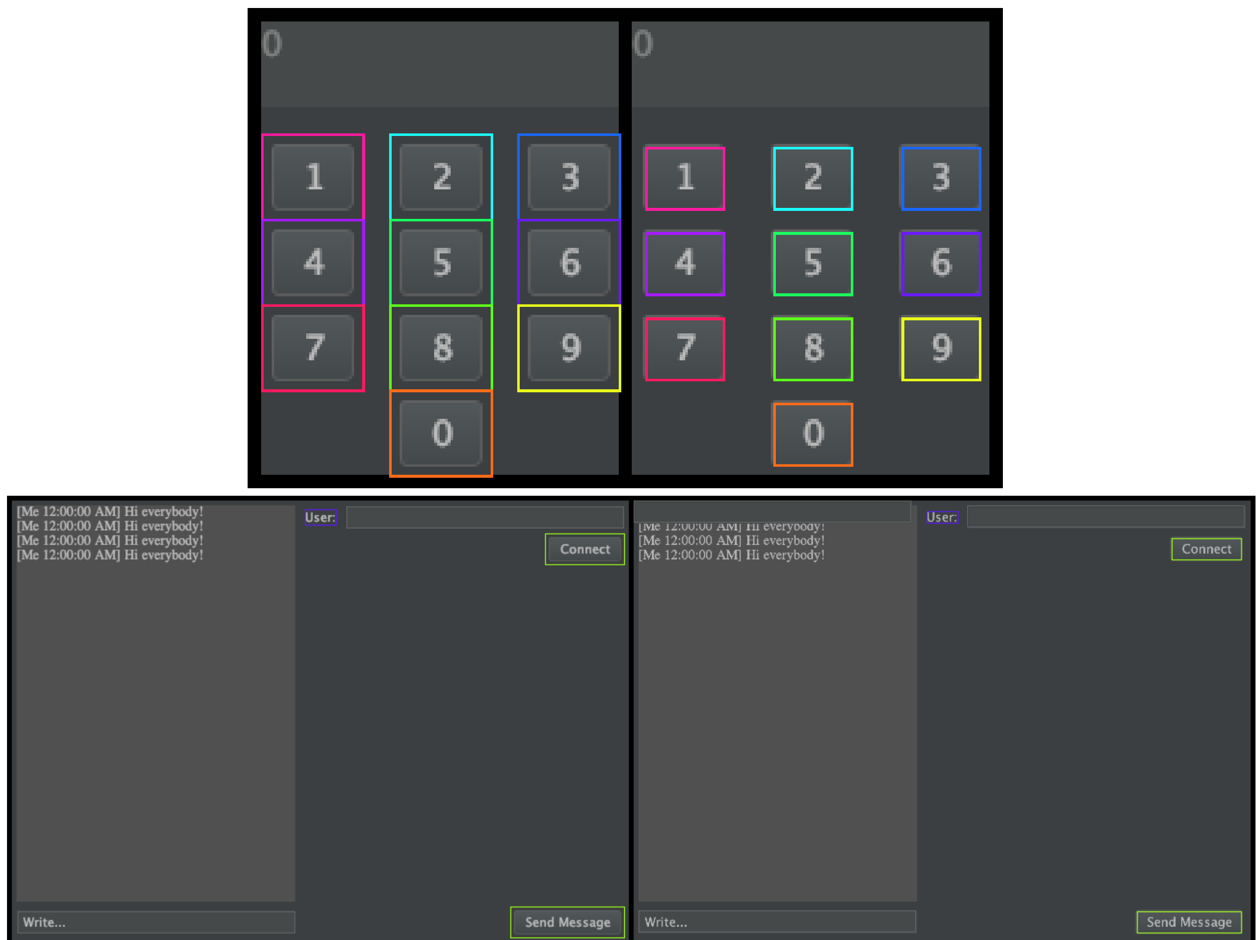
Основные причины, по которым элемент может быть не найден — неправильное распознавание текста на изображении. Для этого предпринимались попытки дообучить Tesseract на небольшом наборе синтетически сгенерированных изображений с текстом низкого разрешения, однако ввиду отсутствия оборудования и датасета нормальных размеров качество удалось увеличить лишь несущественно, поэтому полноценных экспериментов для дообученного Tesseract пока не проводилось.

3.7 Ручные тесты

Перед запуском эксперимента качество распознавания проверяется на простых, но при этом приближенных к реальным, формах. Вот примеры

[2]отношение площади пересечения двух прямоугольников к площади их объединения

некоторых из таких форм (слева — реальные границы элементов, справа — найденные):



Однако, есть ряд форм, на которых валидация будет работать неудовлетворительно. Например, на форме следующего вида валидация будет работать слишком долго:

Это вызвано тем, что для каждой кнопки существует пара, поэтому будет найдено 2^{15} различных конфигураций. Кроме того, комбинаторный

рост вызовов метода поиска кнопки на изображении побуждает сохранять информацию о прямоугольниках и тексте, а не запускать Tesseract каждый раз. Для того, чтобы найти необходимую конфигурацию быстрее, можно, например, использовать информацию о менеджере расположения родительского `JPanel`, но на настоящий момент это не являлось приоритетной задачей и пока не было реализовано.

4 Заключение

В ходе работы мы получили минимальный рабочий прототип фреймворка для валидации пользовательских форм, разработанных с использованием Swing, для темной цветовой схемы IntelliJ IDEA. Однако реализованные нами подходы пока покрывают лишь небольшое множество пользовательских элементов управления (кнопки, надписи и недеформирующиеся элементы), а алгоритм валидации требует дополнительных оптимизаций и отсечений.

Полученный recall хоть и достаточно высок, но все равно недостаточен для того, чтобы полноценно использовать данный фреймворк в продакшне — примерно 15% элементов управления все равно придется валидировать вручную. Задача повышения recall упирается в обучение Tesseract на изображениях низкого разрешения, что требует датасета приемлемых размеров, полученного с реальных форм.

Итого были выполнены следующие задачи:

1. Реализация распознавания на скриншоте кнопок при помощи методов компьютерного зрения (с использованием OpenCV/JavaCPP) и Tesseract.
2. Реализация распознавания на скриншоте надписей.
3. Реализация алгоритма валидации.
4. Реализация генератора случайных форм для оценки качества распознавания.
5. Реализованные подходы оформлены в расширяемую архитектуру.
6. Связка IDEA + OpenCV + Tesseract + данный фреймворк оформлены в виде Docker-контейнера.
7. Предприняты попытки дообучить Tesseract с целью улучшения качества распознавания.

Список литературы

[1] Ballard D.H. Generalizing the Hough transform to detect arbitrary shapes // Pattern Recognition. — 1981. — Vol. 13, no. 2. — P. 111–122.

[2] Burger Wilhelm, Burge Mark J. Principles of digital image processing: core algorithms. — Springer, 2009. — P. 231–232.

[3] Design Patterns: Elements of Reusable Object-Oriented Software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. — Addison-Wesley, 1994.

[4] JavaCPP Documentation. — Access mode: `http://bytedeco.org/javacpp/apidocs/`.

[5] OpenCV Documentation. — Access mode: `https://docs.opencv.org/`.

[6] Performance Metrics for Evaluating Object and Human Detection and Tracking Systems / Afzal Godil, Roger Bostelman, Will Shackleford et al. — NIST, 2014. — July.

[7] Selenium WebDriver. — Access mode: `https://www.seleniumhq.org/projects/webdriver/`.

[8] Szeliski Richard. Computer Vision: Algorithms and Applications Computer Vision: Algorithms and Applications. — Springer, 2010.

[9] Tesseract OCR. — Access mode: `https://github.com/tesseract-ocr/tesseract`.

[10] Tsibulskaya Anna. Screenshots in Automated Testing: When? How? Why? / SeleniumConf. — 2017. — Access mode: `https://www.youtube.com/watch?v=MtvN4Rmh5vs`.

[11] Wang Xiaodong, Wang Lei, Wu Yingjie. An optimal algorithm for Prufer codes // Journal of Software Engineering and Applications. — 2009. — Vol. 2.02, no. 111.