UNIVERSIDADE ESTADUAL DO CEARÁ

CENTRO DE CIÊNCIAS E TECNOLOGIA

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JONNAS CHRISTIAN SOUSA DE PAIVA

MOTOR DE SELF-HEALING PARA TESTES FUNCIONAIS AUTOMATIZADOS: UMA ABORDAGEM GENÉRICA BASEADA EM ANÁLISE DE DOM E ADAPTAÇÃO DINÂMICA DE SELETORES

FORTALEZA – CEARÁ

2025

SUMÁRIO

1 INTRODUÇÃO

A prática de testes de software é fundamental para assegurar a qualidade e a confiabilidade de sistemas modernos. Garantir que uma aplicação funcione corretamente e atenda aos requisitos especificados é um objetivo central em qualquer projeto de software, tornando a qualidade uma característica inegociável. Para alcançar essa validação de forma eficiente, especialmente em contextos nos quais novas funcionalidades são frequentemente integradas, torna-se indispensável o uso de testes automatizados. Esses testes permitem verificar, de maneira consistente e ágil, o comportamento esperado da aplicação, reduzindo falhas e mantendo a integridade do software em todas as etapas do desenvolvimento (ICARO Tech, 2025).

Um dos principais desafios enfrentados pelas equipes que adotam testes automatizados é a fragilidade diante de mudanças frequentes nas interfaces web. Alterações aparentemente simples no DOM (*Document Object Model*)[1], como a troca de um `id`, uma `class` ou o reposicionamento de elementos, podem tornar seletores obsoletos e causar falhas em testes que antes funcionavam corretamente. Segundo Khankhoje (2023), frameworks com suporte a *self-healing* surgem como uma solução promissora para os desafios de manutenção em testes automatizados, permitindo maior eficiência e qualidade mesmo diante de alterações frequentes nas interfaces web. Esses frameworks se baseiam em mecanismos capazes de identificar mudanças e adaptar automaticamente os testes, reduzindo o esforço manual necessário para mantê-los atualizados.

Diante disso, esta proposta tem como objetivo desenvolver um motor de *self-healing* voltado para testes funcionais automatizados, capaz de lidar com mudanças na interface das aplicações web e reduzir o esforço de manutenção. A solução busca contribuir para a robustez dos testes, oferecendo maior adaptabilidade e integração com diferentes ferramentas de automação.

1.1 MOTIVAÇÃO

A manutenção é frequentemente apontada como um dos principais desafios da automação de testes. À medida que a complexidade do software cresce, aumenta também a quantidade de scripts necessários para acompanhar essa evolução, o que sobrecarrega as equipes com tarefas de atualização e correção. Segundo Battina (2019), cerca de 40% do tempo dos testadores é consumido apenas com manutenção, incluindo esforços para depurar e ajustar testes que falham devido a alterações no sistema.

[1] O Document Object Model (DOM) é uma representação em árvore da estrutura HTML de uma página web, permitindo que scripts acessem, naveguem e modifiquem seus elementos dinamicamente.

Além dos desafios relacionados à manutenção constante, ferramentas existentes que propõem mecanismos de self-healing também apresentam limitações técnicas que comprometem sua adoção em cenários mais amplos. O *Healenium*[2], por exemplo, embora seja uma solução robusta e de código aberto, possui suporte restrito a determinadas linguagens e frameworks, não sendo compatível com outras ferramentas amplamente utilizadas, como o *Cypress*[3]. Além disso, sua configuração pode ser complexa e pouco flexível (KHANKHOJE, 2023). Algumas soluções modernas também dependem de técnicas de IA (Inteligência Artificial) para adaptação de seletores, o que pode tornar a implementação mais pesada, menos transparente e limitada a contextos específicos. Essas limitações evidenciam a necessidade de mecanismos de autoajuste que sejam simples, explicáveis e aplicáveis a diferentes ferramentas e realidades de automação.

1.2 OBJETIVO GERAL

Desenvolver uma biblioteca em `Python` com capacidades de *self-healing* para testes funcionais automatizados, capaz de identificar alterações no DOM e corrigir seletores de forma inteligente, promovendo maior estabilidade e eficiência na execução dos testes.

1.3 OBJETIVOS ESPECÍFICOS

- Implementar a funcionalidade de extração de elementos do DOM, com suporte a diferentes tipos de seletores;
- Criar um mecanismo de comparação entre versões do DOM antes e depois das alterações;
- Elaborar estratégias para readequação automática de seletores utilizados nos testes;
- Integrar a solução com frameworks populares de automação de testes, como *Selenium*, *Cypress* e *Robot Framework*;
- Avaliar a eficácia da biblioteca em cenários reais de automação e analisar a viabilidade de versões alternativas, como uma versão leve (*Lite*) e outra completa (*Full*).

[2] *Healenium* é uma biblioteca de código aberto voltada para testes automatizados com mecanismo de auto-recuperação de seletores, compatível principalmente com Selenium em Java.

[3] *Cypress* é um framework moderno para testes end-to-end em aplicações web, baseado em JavaScript e amplamente utilizado por sua integração com o navegador e facilidade de uso.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos que sustentam a proposta deste trabalho. Com base em fundamentos consolidados da área de testes de software, discute-se o papel da automação e a evolução para mecanismos de *self-healing*, apontando caminhos para soluções mais resilientes, flexíveis e alinhadas aos desafios atuais da engenharia de software.

2.1 TESTES DE SOFTWARE

Testes de software são um conjunto de atividades realizadas com o propósito de avaliar a qualidade de um sistema e identificar possíveis defeitos em seus artefatos. Essas atividades envolvem tanto a verificação, que consiste em assegurar que o sistema atenda aos requisitos especificados, quanto a validação, que verifica se o produto final satisfaz as necessidades dos usuários e demais partes interessadas (ISTQB, 2023). Os testes também têm como objetivos detectar defeitos, avaliar produtos de trabalho, validar o funcionamento do sistema e fornecer informações que apoiem a tomada de decisões. Além disso, contribuem para aumentar a confiança na qualidade do produto e reduzir riscos relacionados à não conformidade com requisitos funcionais ou legais.

De acordo com a CTFL (ISTQB, 2023), os testes de software podem ser classificados em funcionais, que verificam se o sistema executa corretamente suas funcionalidades; não funcionais, que avaliam atributos que caracterizam qualidade; de caixa preta, que derivam testes a partir das especificações para verificar o comportamento do sistema; e de caixa branca, que se baseiam na estrutura interna do sistema, como código e fluxos de controle. Além dessas classificações, os testes também são organizados em diferentes níveis, de acordo com o escopo da verificação: teste de unidade, que verifica componentes isolados; teste de integração, que avalia a interação entre módulos ou sistemas; teste de sistema, que considera o sistema completo; e teste de aceitação, que valida se o produto atende aos critérios definidos pelo cliente ou usuário final.

2.2 AUTOMAÇÃO DE TESTES

A automação de testes consiste na utilização de ferramentas para executar, verificar e comparar os resultados de testes de software, com o objetivo de reduzir o esforço manual, aumentar a eficiência e garantir a repetibilidade no processo de verificação. Trata-se de uma

prática estratégica que permite o reaproveitamento de testes, a detecção de regressões com maior agilidade e a padronização das execuções. Os artefatos utilizados e gerados nesse processo compõem o chamado *testware*[1]. Embora traga benefícios em escala e consistência, a automação também impõe desafios quanto ao esforço inicial, à manutenção de scripts e à instabilidade provocada por alterações frequentes nas interfaces (FEWSTER; GRAHAM, 1999).

A norma ISO/IEC/IEEE (2022) ressalta que a automação deve ser aplicada com critério, considerando o contexto do projeto, os objetivos dos testes e os recursos disponíveis. Nem todos os tipos de teste são adequados para automação — especialmente aqueles que envolvem julgamento subjetivo ou são executados com pouca frequência. Fatores como a estabilidade da interface, a maturidade do sistema e a frequência de mudanças impactam diretamente na viabilidade da automação. Para que os benefícios sejam sustentáveis, é necessário considerar os custos de implementação e manutenção dos scripts, a capacitação da equipe e a escolha criteriosa das ferramentas. Como reforçado por Fewster e Graham (1999), uma abordagem mal planejada pode transformar o que seria uma otimização em um caos automatizado.

2.3 SELF-HEALING

O conceito de *self-healing* em testes automatizados refere-se à capacidade de um sistema identificar e corrigir automaticamente falhas nos scripts de teste, geralmente causadas por alterações na estrutura da interface da aplicação. Essas falhas normalmente ocorrem devido a mudanças no DOM, como alterações em atributos, reorganização de elementos ou substituição de identificadores. Frameworks com suporte a *self-healing* monitoram a execução dos testes, detectam quando um seletor não localiza o elemento esperado e tentam, com base em regras ou comparações estruturais, encontrar alternativas viáveis para permitir a continuidade da execução com sucesso.

De acordo com Khankhoje (2023), soluções de *self-healing* geralmente operam a partir de uma arquitetura composta por módulos de monitoramento, detecção de falhas, reidentificação de elementos e atualização de scripts. Essas soluções podem empregar desde estratégias baseadas em IA até mecanismos mais simples, como análise de atributos do DOM, comparação de versões e aplicação de heurísticas estruturais. Independentemente da técnica utilizada, o objetivo é manter os testes funcionais automatizados mesmo com mudanças na interface, reduzindo a manutenção e aumentando a confiabilidade da automação.

[1] *Testware* é o conjunto de artefatos utilizados e produzidos durante os testes, como scripts, dados de entrada, saídas esperadas, ferramentas de apoio e relatórios gerados.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta os principais trabalhos relacionados ao tema deste trabalho, com foco em propostas que visam aumentar a resiliência e a eficácia dos testes automatizados. Para isso, são exploradas abordagens distintas, desde soluções baseadas em técnicas de IA até estratégias heurísticas mais simples. A organização do capítulo reflete essa diversidade: inicia-se com frameworks avançados baseados em IA, passa-se por algoritmos heurísticos voltados à robustez dos seletores e finaliza-se com uma abordagem prática sem dependência de IA, que mais se aproxima da proposta desenvolvida neste trabalho. Os trabalhos discutidos servem de base teórica e prática para a construção da solução apresentada neste projeto.

3.1 FRAMEWORK DE TESTES SELF-HEALING COM INTELIGÊNCIA ARTIFICIAL E APRENDIZADO DE MÁQUINA

O trabalho de Saarathy, Bathrachalam e Rajendran (2024) propõe um framework de automação de testes com capacidade de auto-recuperação, baseado em IA e ML (Aprendizado de Máquina). O objetivo central é reduzir o esforço de manutenção causado por falhas em testes automatizados devido a mudanças frequentes na interface das aplicações web. A proposta combina técnicas como identificação dinâmica de localizadores, espera inteligente e detecção de anomalias, aplicadas a partir de modelos treinados com dados históricos. Os autores validaram sua abordagem em uma aplicação real, observando aumento na confiabilidade da suíte de testes (de 70% para 90%) e redução de 70% no tempo de manutenção, além de economia estimada de 60% nos custos relacionados.

Embora o trabalho apresente resultados relevantes, sua implementação exige um ecossistema complexo de ferramentas e uma infraestrutura robusta para o treinamento e uso de modelos de IA e ML, o que pode limitar sua aplicabilidade em cenários com restrições técnicas. A proposta deste projeto, por sua vez, aborda a mesma problemática sob uma perspectiva mais acessível, priorizando soluções genéricas baseadas em heurísticas e estrutura do DOM, independentes de grandes volumes de dados ou configurações avançadas, e com maior facilidade de integração a ferramentas de mercado.

3.2 ROBULA+: GERAÇÃO DE LOCALIZADORES XPATH ROBUSTOS PARA TESTES WEB

O trabalho de Leotta *et al.* (2016) apresenta o algoritmo ROBULA+, voltado à geração automática de localizadores `XPath` mais robustos para testes de aplicações web. A proposta parte do reconhecimento de que os testes automatizados, embora amplamente utilizados, são altamente sensíveis a mudanças estruturais no DOM, sendo os localizadores um dos principais pontos de fragilidade.

O algoritmo ROBULA+ aplica um conjunto de heurísticas para construir expressões `XPath` que mantenham sua validade mesmo após atualizações na interface da aplicação. Entre as estratégias utilizadas estão a priorização de atributos considerados mais estáveis (como *id* e *name*), o uso combinado de múltiplos atributos, a exclusão de atributos frágeis (como `style`, `onclick`, entre outros) e a especialização progressiva de seletores a partir de um `XPath` genérico. Os testes realizados em oito aplicações reais mostraram que ROBULA+ reduz em até 90% a fragilidade dos localizadores em comparação com ferramentas tradicionais como *Selenium IDE*[1] e *FirePath*[2].

Embora o artigo não proponha um mecanismo de *self-healing* propriamente dito, ele contribui significativamente para o aumento da resiliência dos testes frente a mudanças estruturais. A proposta deste projeto se diferencia ao dar um passo além: além de buscar robustez na geração dos seletores, também trata da sua adaptação automática em tempo de execução, oferecendo uma abordagem reativa e autônoma à quebra de testes causada por alterações no DOM.

3.3 SELF-HEALING COM INTELIGÊNCIA ARTIFICIAL EM PROCESSOS DE CONFORMIDADE REGULATÓRIA

O trabalho de Tamraparani e Dalal (2023) propõe uma abordagem de automação de testes voltada para instituições financeiras, com foco em atividades relacionadas à conformidade regulatória. A solução utiliza IA e ML para gerar e corrigir automaticamente scripts de teste, reduzindo o esforço manual em processos como detecção de fraudes e análise documental.

Apesar de promissor, o estudo deixa claro que essa abordagem depende de grandes volumes de dados, infraestrutura robusta e mão de obra especializada. Além disso, por atuar

[1] *Selenium IDE* é uma ferramenta que permite gravar e reproduzir testes automatizados diretamente no navegador.

[2] *FirePath* era uma extensão do Firefox usada para inspecionar e gerar seletores XPath e CSS, descontinuada desde 2017.

em domínios altamente controlados e com regras bem definidas, a solução se mostra limitada diante de cenários em que os elementos da interface mudam com frequência e sem padrão previsível. Nesta proposta, o foco está justamente em enfrentar esse tipo de desafio: adaptar os testes a mudanças estruturais no DOM em ambientes onde não há garantia de consistência ou padronização. Com isso, busca-se oferecer uma solução que seja capaz de operar mesmo em contextos incertos e em constante evolução, como é comum em aplicações web modernas.

3.4 ESTRATÉGIAS HEURÍSTICAS E SELETORES ALTERNATIVOS BASEADAS EM FALLBACK

Enquanto muitos estudos recentes têm focado no uso de IA para tornar testes automatizados mais resilientes, algumas abordagens alternativas se destacam por utilizar estratégias heurísticas e mecanismos de *fallback*, sem recorrer a IA. Um exemplo prático é apresentado pela empresa Wopee.io (2024), no artigo técnico intitulado *Self-healing in SW Test Automation Explained*, no qual são aplicadas tentativas sequenciais de múltiplos seletores — como os seletores `id`, `name`, `XPath` e `CSS` — para localizar elementos na interface. Caso o seletor principal falhe, o sistema tenta alternativas previamente definidas, utilizando também heurísticas simples como a correspondência textual flexível (*fuzzy matching*)[3].

Embora não seja um estudo acadêmico formal, esse tipo de solução prática reforça a relevância de estratégias baseadas em regras para aumentar a robustez dos testes automatizados. Logo, este trabalho serve como referência ao demonstrar que é possível implementar mecanismos de auto-recuperação eficazes sem depender de técnicas complexas de IA, alinhando-se à proposta aqui desenvolvida de explorar seletores alternativos e heurísticas como caminho viável para testes mais resilientes.

[3] *Fuzzy matching* é uma técnica utilizada para comparar strings de forma aproximada, identificando similaridades mesmo quando há pequenas diferenças ou erros de escrita.

4 METODOLOGIA

Este capítulo descreve a metodologia utilizada para o desenvolvimento de uma biblioteca voltada à automação de testes com capacidade de auto-recuperação. A proposta visa permitir a detecção de mudanças no DOM e a adaptação automática dos seletores utilizados nos testes, minimizando o impacto de alterações estruturais na interface das aplicações.

4.1 FUNCIONAMENTO GERAL DA BIBLIOTECA

A biblioteca será desenvolvida com base na linguagem `Python`, utilizando o *Selenium* como ferramenta de extração do DOM. O processo envolve a captura da estrutura da página em dois momentos distintos: o tempo t_0 (antes da alteração) e o tempo t_1 (após a alteração). Os elementos extraídos em cada momento são armazenados em arquivos no formato JSON[1], permitindo que as diferenças estruturais entre as versões sejam detectadas com precisão.

A ferramenta será disponibilizada em duas versões: *Lite* e *Full*. A versão *Lite* terá como foco os 80% principais elementos da página, utilizando localizadores mais simples e diretos, como `id`, `name` e `class`, que costumam ser mais estáveis e eficientes para fins de identificação (LEOTTA *et al.*, 2016; BROWSERSTACK, 2024). Já a versão *Full* realizará uma extração completa do DOM, aplicando estratégias adicionais de localização como seletores `XPath`, `link text`, entre outros, com o objetivo de oferecer maior cobertura e flexibilidade em cenários complexos.

4.2 DETECÇÃO DE ALTERAÇÕES E CORREÇÃO DE SELETORES

Após a comparação entre as estruturas obtidas em t_0 e t_1, a biblioteca será capaz de identificar alterações em atributos, remoções ou reposicionamentos de elementos. Quando uma modificação relevante for detectada, a ferramenta irá gerar automaticamente uma nova sugestão de seletor, utilizando estratégias heurísticas para garantir maior estabilidade. Essas heurísticas consistem na identificação direta de mudanças nos atributos dos elementos entre as versões extraídas. Por exemplo, se em t_0 um elemento apresentava o atributo `id` com determinado valor, e em t_1 esse valor foi modificado, a ferramenta reconhece essa alteração e realiza automaticamente a substituição no seletor correspondente. Uma vez validada a correspondência, o seletor atualizado poderá ser imediatamente repassado ao framework de testes, com a possibilidade de atualização

[1] JSON (JavaScript Object Notation) é um formato leve e padronizado para armazenamento e troca de dados estruturados, amplamente utilizado em aplicações web.

direta nos arquivos de mapeamento de elementos.

A atualização poderá ocorrer por meio da sobrescrita automatizada dos arquivos de mapeamento utilizados pelos frameworks, como objetos ou estruturas auxiliares que armazenam os seletores. Com isso, os scripts de teste permanecem inalterados, utilizando os mesmos identificadores lógicos, enquanto a biblioteca se encarrega de manter os seletores atualizados de acordo com o DOM mais recente.

4.3 INTEGRAÇÃO E VALIDAÇÃO DA SOLUÇÃO

A biblioteca será integrada a três frameworks populares de automação de testes: *Selenium*, *Cypress* e *Robot Framework*. A escolha desses frameworks se deve à sua ampla adoção na indústria e à diversidade de estilos e linguagens, o que permitirá validar a flexibilidade da solução proposta em diferentes contextos. A integração será testada por meio da criação de suítes funcionais básicas para cada ferramenta, executando casos de teste em páginas com alterações planejadas. Inicialmente, os testes serão realizados em um site simulado desenvolvido especificamente para este projeto e, alternativamente, em páginas públicas disponíveis na web que permitam simulações controladas.

Para validar a eficácia da solução, serão realizados experimentos em páginas com diferentes níveis de complexidade, simulando alterações reais na estrutura da interface. Como critérios de avaliação, serão observadas a taxa de sucesso na correção automática dos seletores, o impacto na continuidade dos testes e a redução da necessidade de manutenção manual dos scripts. Apesar do potencial da abordagem, é importante destacar que cenários mais complexos, como alterações simultâneas em múltiplos elementos ou mudanças profundas na estrutura da página, podem representar desafios adicionais. Nessas situações, futuras versões da biblioteca poderão incorporar heurísticas mais avançadas ou critérios complementares para aumentar a precisão da identificação e substituição dos seletores afetados.

5 CRONOGRAMA

O desenvolvimento do projeto será dividido nas seguintes etapas:

Atividades	**Mar**	**Abr**	**Mai**	**Jun**	**Jul**
Extração e armazenamento do DOM	X				
Comparação entre versões JSON (t_0 e t_1) e detecção de alterações		X			
Geração e substituição automática de seletores		X	X		
Integração com frameworks de automação (*Selenium*, *Cypress* e *Robot*)			X	X	
Avaliação da eficácia da biblioteca (incluindo análise comparativa entre versões *Lite* e *Full*)				X	X
Escrita da documentação do TCC[1]				X	X
Defesa do TCC					X

Tabela 1 – Cronograma de desenvolvimento do projeto

[1] TCC refere-se ao Trabalho de Conclusão de Curso, etapa final obrigatória de cursos de graduação.

6 CONCLUSÃO

A automação de testes é fundamental para garantir qualidade em ambientes de desenvolvimento ágil, mas sofre com desafios de manutenção causados por mudanças frequentes na estrutura das interfaces. A proposta deste trabalho nasce desse contexto, com o objetivo de oferecer uma solução prática e genérica para tornar os testes automatizados mais resilientes, especialmente frente às alterações no DOM que comprometem a execução contínua das suítes de teste. Ao implementar uma biblioteca capaz de identificar modificações no DOM e adaptar seletores de forma autônoma, espera-se reduzir falhas recorrentes e o esforço manual associado à manutenção de testes. A ferramenta buscará se integrar de forma transparente a diferentes frameworks amplamente utilizados, respeitando os padrões e estruturas de cada um. Além disso, a proposta considera dois modos de operação, cuja avaliação prática permitirá entender melhor o equilíbrio entre desempenho, cobertura e complexidade da solução.

Mais do que entregar uma ferramenta funcional, este projeto busca contribuir para a evolução das práticas de automação, trazendo simplicidade e acessibilidade a um problema recorrente e ainda pouco tratado de forma padronizada. Ao final, espera-se não apenas validar a viabilidade técnica da abordagem, mas também demonstrar seu potencial de aplicação em cenários reais e sua utilidade prática para equipes de teste em contextos ágeis e dinâmicos.

REFERÊNCIAS

BATTINA, D. S. Artificial intelligence in software test automation: A systematic literature review. **Journal of Emerging Technologies and Innovative Research (JETIR)**, v. 6, n. 12, p. 1329–1336, 2019. Disponível em: <https://www.jetir.org/papers/JETIR1912176.pdf>.

BROWSERSTACK. **Locators in Selenium: A Detailed Guide**. 2024. Acesso em: 16 abr. 2025. Disponível em: <https://www.browserstack.com/guide/locators-in-selenium>.

FEWSTER, M.; GRAHAM, D. **Software Test Automation: Effective Use of Test Execution Tools**. New York: Addison-Wesley, 1999. ISBN 0201331403.

ICARO Tech. **A Importância dos Testes Automatizados no Desenvolvimento de Software**. 2025. <https://icarotech.com/blog/testes-automatizados-software>. Acesso em: abr. 2025.

ISO/IEC/IEEE. **ISO/IEC/IEEE 29119-1:2022 – Software and systems engineering — Software testing — Part 1: Concepts and definitions**. 2022. <https://www.iso.org/standard/72206.html>. Norma internacional sobre conceitos fundamentais de teste de software.

ISTQB. **Syllabus CTFL v4.0 – Nível Foundation: Versão Brasileira**. 2023. <https://bstqb.online/ctfl>. Tradução oficial publicada pela BSTQB (Brazilian Software Testing Qualifications Board).

KHANKHOJE, R. Effortless test maintenance: A critical review of self-healing frameworks. **International Journal for Research in Applied Science and Engineering Technology**, v. 11, p. 1–12, 2023.

LEOTTA, M.; STOCCO, A.; RICCA, F.; TONELLA, P. Robula+: An algorithm for generating robust xpath locators for web testing. **Journal of Software: Evolution and Process**, John Wiley & Sons, v. 28, n. 3, p. 177–204, 2016.

SAARATHY, S. C. P.; BATHRACHALAM, S.; RAJENDRAN, B. K. Self-healing test automation framework using ai and ml. **International Journal of Strategic Management**, v. 3, n. 3, p. 45–77, 2024.

TAMRAPARANI, V.; DALAL, A. Self-generating & self-healing test automation scripts using AI for automating regulatory & compliance functions in financial institutions. **Revista de Inteligência Artificial em Medicina**, v. 14, n. 1, p. 784–796, 2023. Available online. Disponível em: <https://redcrevistas.com/index.php/Revista>.

Wopee.io. **Self-healing in SW Test Automation Explained**. 2024. Acesso em: 15 abr. 2025. Disponível em: <https://wopee.io/blog/self-healing-in-sw-test-automation/>.