SideForge: Evoluindo a Automação de Testes Funcionais no Setor Público

Alexandre Almeida de Oliveira
PESC/COPPE, UFRJ
Rio de Janeiro, Brazil
aaoliveira@cos.ufrj.br

Rebeca Motta
UFF
Niterói, Brazil
rebecamotta@id.uff.br

Elaine Nunes
CAPGov - COPPETEC
Rio de Janeiro, Brazil
elaine@cos.ufrj.br

Gustavo Rocha Barreto Pinto
CAPGov - COPPETEC
Rio de Janeiro, Brazil
grbpinto@cos.ufrj.br

Rafael Maiani de Mello
UFRJ
Rio de Janeiro, Brazil
rafaelmello@ic.ufrj.br

Resumo

Embora a adoção de ferramentas de captura e *replay* simplifique a geração de novos testes funcionais, a manutenção destes testes enfrenta desafios consideráveis devido a limitações das *IDEs* disponíveis, à volatilidade da interface do usuário e às características tecnológicas de sistemas legados complexos. Este resumo da indústria introduz *SideForge*, uma ferramenta de automação de testes funcionais resiliente, desenvolvida pelo time responsável pelos testes automatizados do principal sistema *Web* de uma instituição pública brasileira. O *SideForge* foi projetado para ir além dos recursos oferecidos pelo *Selenium IDE*, solucionando as principais dores relatadas pelos profissionais do time na gravação e manutenção dos testes funcionais. Entre outros recursos, *SideForge* permite a filtragem e ordenação dinâmica de seletores, a correção do teste em tempo de execução e a persistência de variáveis de teste em memória. Também relatamos neste artigo as principais lições aprendidas com essa jornada.

Palavras-chave

Teste de Software, Automação de Testes, Testes Funcionais, Manutenção de Testes, Setor Público

1 Introdução

A adoção de testes automatizados funcionais de ponta a ponta é uma prática indispensável para viabilizar a condução de testes de regressão abrangentes, especialmente no caso de sistemas de software com alta complexidade de negócio. Neste cenário, testes de regressão costumam ser realizados antes de cada *deploy*, tornando-se importante evidência de qualidade do produto. Neste sentido, a tecnologia de *capture-replay* contribuiu para a disseminação da prática. Ferramentas de *capture-replay* como o *Selenium IDE* [4] permitem a gravação e execução de testes funcionais através da coleta de interações manuais e de sua conversão para *scripts* de teste. A princípio, esta abordagem pode reduzir consideravelmente o esforço na criação dos testes. Entretanto, a manutenção de tais testes pode se tornar ainda mais custosa do que a abordagem tradicional de escrita de testes diretamente em código, principalmente devido às limitações da *IDE* [2]. Este custo de manutenção concentra-se, em boa parte, no reparo manual de localizadores de elementos da interface de navegação — problema conhecido como *fragilidade* do teste [3]. Este trabalho relata a concepção e evolução do *SideForge*, uma ferramenta de automação de testes funcionais que tem por objetivo otimizar os benefícios de *capture-replay* e combiná-los às vantagens da manutenção direta em código. *SideForge* foi concebido no âmbito de uma equipe de testes funcionais responsável pelo principal sistema *Web* de uma instituição pública brasileira do setor jurídico [5]. Ao longo de cinco *sprints* com *feedback* contínuo de profissionais de teste, a ferramenta evoluiu de um simples conversor de *scripts* para um ambiente de execução e depuração resiliente em código nativo, que soluciona os principais problemas relatados no uso do *Selenium IDE* através de recursos como anti-fragilidade, gravação robusta e mascaramento de dados sensíveis.

2 Contexto e Motivação

A experiência relatada neste trabalho insere-se no contexto de um programa de parceria de desenvolvimento de soluções de software para o setor público brasileiro, no caso específico, uma instituição do setor jurídico. O programa desenvolveu e mantém, desde 2015, o principal sistema de informação desta instituição. Trata-se de um sistema *Web* de alta complexidade de negócio, que ao longo dos anos cresceu em módulos, volume de usuários e complexidade, tornando os testes de regressão indispensáveis a cada *deploy*. Nos últimos cinco anos, a prática da equipe de testes funcionais evoluiu de uma abordagem *ad-hoc* para uma estrutura dedicada de garantia de qualidade centrada em automação [5]. Atualmente, a equipe de testes possui um time de profissionais dedicados à automação de testes, sendo também responsável pela condução de testes de regressão a cada *deploy*. Estes profissionais adotaram o *Selenium IDE* como ferramenta padrão para as atividades de automação desde 2023, gerando um extenso acervo de testes funcionais continuamente revisado e atualizado. No entanto, a preocupação com a obsolescência do *Selenium IDE* levou a equipe de testes a buscar alternativas. A ferramenta, distribuída como extensão de navegador no padrão *Manifest V2*, foi diretamente afetada pela transição do navegador *Google Chrome* para o *Manifest V3* em 2025, levando a sua não disponibilização neste navegador. Neste cenário, o time de testes automatizados iniciou o desenvolvimento de uma solução de contorno para preservar a reutilização do acervo de testes: um conversor em lote (*batch converter*) que transcreveria os arquivos `.side` gerados pelo *Selenium IDE* em *scripts Python* padronizados via *Pytest*, acoplado a um executor básico. Durante a concepção do conversor em lote, o time de testes automatizados refletiu sobre os desafios e limitações deste contorno a longo prazo. Manter *scripts* de testes funcionais diretamente em código traria grande impacto

operacional para o time, exigindo uma curva de aprendizagem considerável e tornando o processo de criação de novos testes mais trabalhoso. Além disto, a edição dos testes diretamente no código representava um maior risco operacional de introdução de defeitos nos testes, comprometendo a produtividade e a confiabilidade dos testes de regressão. Considerando estas questões, a equipe de testes optou por investir no desenvolvimento de *SideForge*, uma ferramenta que visa preservar a experiência de uso do *Selenium IDE* e contornar suas principais limitações.

3 Visão Geral da Solução

O *SideForge* é organizado em módulos (Figura 1) com responsabilidades distintas, articulados em torno de um *pipeline* de quatro estágios: *converter*, *executar*, *recuperar* e *curar*. O módulo *tradutor* converte cada teste do *Selenium* `.side` em uma *classe pytest*, mapeando os comandos originais do *Selenium IDE* para chamadas do *Selenium WebDriver*. O *motor de execução* — a classe-base da qual todo teste herda — provê a infraestrutura de espera, interação resiliente, registro (*logging*) e recuperação interativa. Módulos de *gestão* cuidam da criação e edição de testes, da fila de execução por demanda e da geração de relatórios. A principal decisão de *design* consistiu em tratar o arquivo `.side` como base de dados estrutural e o *script* Python derivado como um artefato efêmero, recompilado em tempo de execução. Ao invés de o profissional de testes editar o código gerado — o que está sujeito a erros sintáticos —, todas as alterações (de seletores, valores ou da própria sequência de passos) são persistidas no `JSON` do `.side`. Uma preocupação no *design* do *SideForge* para promover a adesão do time de testes automatizados foi minimizar a curva de aprendizagem na transição do *Selenium IDE* para a nova ferramenta. Para tal, *SideForge* reproduz no terminal a metáfora visual do *Selenium IDE*. Por exemplo, o painel de execução do *SideForge* exibe a lista de passos dos testes no mesmo formato do *Selenium IDE*, com o passo corrente destacado e indicadores de *status* e *breakpoint* por linha. A familiaridade no uso estende-se ao navegador, com o diferencial de não depender da instalação de extensões. Durante a execução, o motor da ferramenta injeta uma barra de controle fixa e minimizável sobreposta à página sob teste, sinalizando que o *SideForge* está ativo, disponibilizando botões para controle da gravação. A seguir, apresentamos os principais problemas identificados na experiência de uso do *Selenium IDE* ao longo dos anos e as respectivas soluções oferecidas pelo *SideForge*. As soluções providas pelo *SideForge* foram concebidas com base no *feedback* continuamente coletado junto à equipe sobre limitações e dificuldades no uso do *Selenium IDE* ao fim de cada *sprint*.

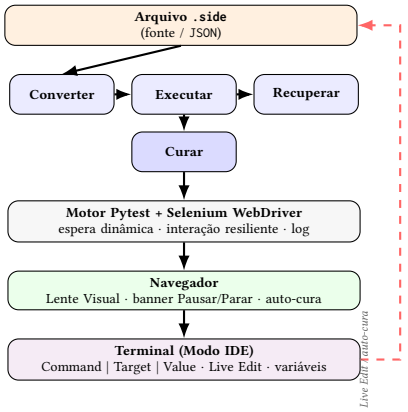

Figura 1: Visão geral do *SideForge*.

3.1 *Live Edit* e Depuração ao Vivo

Problema: O *Selenium IDE* assume que falhas na execução do caso de teste são determinísticas e aborta a execução do teste (*teardown*). **Solução:** O *SideForge* oferece um comportamento resiliente: ao sinal de falha no teste, o motor pausa o navegador e exibe um painel com o passo corrente, os passos vizinhos e os *logs*. A partir deste painel, o operador dispõe de um menu de *recuperação do teste* que inclui as seguintes alternativas: repetir o passo (*retry*), pular o passo (*skip*), saltar a execução para um passo arbitrário (*jump*), inspecionar as variáveis em memória, marcar *breakpoints* e abrir o *Live Edit*. Nesta última opção, o analista pode editar os comandos do teste, além de permitir a captura de um novo alvo na navegação (Lente Visual) sem a necessidade de sair da execução. Na sequência, um mecanismo de *Hot Reload* aplica as mudanças realizadas no teste à memória e ao `.side`, retomando o teste no ponto exato da interrupção, eliminando a necessidade de reiniciar o caso de teste.

3.2 Persistência de Variáveis em Memória

Problema: No *Selenium IDE*, variáveis definidas durante o teste existem apenas enquanto a sessão está ativa. Quando o teste falha, este conteúdo é perdido, levando à necessidade de reexecutar o teste do início. Em fluxos de teste extensos, com muitos dados acumulados, este reinício torna-se consideravelmente oneroso. **Solução:** O *SideForge* elimina esta limitação ao manter as variáveis em uma estrutura persistente em memória, dissociada do ciclo de vida do passo individual. Quando a execução do teste pausa por uma falha, o estado acumulado permanece intacto e pode ser inspecionado a qualquer momento pelo painel, que exibe o conjunto corrente de variáveis. Combinado ao *Hot Reload*, isso significa que o analista corrige o passo problemático e retoma a execução com todo o contexto preservado — sem a necessidade de reiniciar o teste.

3.3 Auto-Cura Anti-Frágil

Problema: O *Selenium IDE* possui uma fragilidade notável na captura de seletores. A depender do tipo do seletor, a automação pode se tornar mais frágil ou mais estável. Quando um seletor frágil é escolhido como primário (*default*) pelo Selenium IDE mas o teste falha depois de certo tempo, os analistas precisam substituir os seletores problemáticos um a um manualmente para manter a integridade do teste. Nesse contexto, fez-se necessário explorar motores que reparassem seletores de forma assíncrona e não seletiva [1, 3]. **Solução:** A rotina de auto-cura (*self-healing*) do *SideForge* percorre a matriz de *fallbacks*; se o seletor primário falha e um alternativo funciona, o seletor vencedor é promovido a primário e a alteração é persistida. Para não degradar a suíte a longo prazo, um *filtro anti-fragilidade* recusa a cura quando o seletor vencedor é sabidamente instável, classificando os candidatos em níveis de robustez e descartando, por exemplo: identificadores dinâmicos `j_idt`, típicos de arquiteturas JSF/PrimeFaces, que mudam a cada *build*; *XPaths* puramente

posicionais; e seletores *CSS* compostos apenas por uma *tag HTML* genérica. Nesses casos, o seletor alternativo é usado apenas em memória para concluir a execução corrente.

3.4 Robustez na Navegação

Problema: O *Selenium IDE* adota um tempo de espera estático e curto para o surgimento de novas janelas, o que produz falsos negativos quando a aba demora a ser registrada pelo navegador. Isto era uma fonte recorrente de falhas intermitentes em fluxos que envolvem *pop-ups*, novas abas ou janelas de autenticação. **Solução:** O *SideForge* adota a estratégia de *espera dinâmica*, que monitora a abertura de novas janelas até um limite global de tempo, detectando a nova aba assim que ela se materializa. De modo complementar, a operação de troca de janela é rigorosa: quando um identificador de janela chega inválido ou vazio — situação comum após *timeouts* de abertura —, o motor recupera-se automaticamente, alternando para a janela mais recente em vez de abortar a execução.

3.5 Mascaramento e Relatórios Diferenciados

Problema: No *Selenium IDE*, não havia mecanismos ativos de mascaramento de dados. Isto é particularmente preocupante no setor público, onde existe o intenso tráfego de dados sensíveis. Como contorno, os profissionais de teste realizavam tratamentos manuais nos registros de testes, representando risco operacional. **Solução:** O *SideForge* oferece dois importantes recursos para este problema. O primeiro deles é o recurso de *mascaramento automático de dados sensíveis*. Através dele, o registrador de *logs* detecta campos de senha, usuário e identificadores pessoais e ofusca os valores digitados antes de persisti-los. O segundo recurso consiste na *geração de relatórios em duas versões* a partir da mesma execução: um relatório de nível *tester*, completo, com *screenshots* de falha e a trilha técnica de recuperação e auto-cura; e um relatório de nível *cliente*, livre de ruído técnico, voltado a interlocutores não técnicos e visando atender ao fornecimento de evidências de execução dos testes de regressão para a instituição pública.

4 Lições Aprendidas

O desenvolvimento do *SideForge* foi incremental e guiado por *feedback* contínuo ao longo de cinco *sprints* de manutenção evolutiva do sistema jurídico (aproximadamente seis meses). Além do desenvolvedor da ferramenta, outros dois profissionais do time de automação atuaram como usuários-piloto em quatro *sprints*, exercitando-a em seus fluxos reais de testes de regressão; ao todo, cerca de 1.400 testes funcionais foram executados com diferentes versões do *SideForge*, cobrindo funcionalidades de diferentes complexidades e recursos de navegação. Os profissionais registravam em uma planilha compartilhada as falhas, limitações e melhorias percebidas, o que realimentava novas regras de resiliência e comportamentos da ferramenta — a maioria das heurísticas do filtro anti-frágil e de auto-cura, por exemplo, emergiu da observação desses padrões de falha recorrentes. Essa interação próxima entre uso e evolução ao longo de *sprints* reais foi decisiva para adequar a ferramenta às particularidades do sistema sob teste. A condução desta jornada produziu lições que julgamos poder contribuir com equipes de testes com desafios similares. **(1) A fonte do teste deve ser estrutural, não código:** tratar o artefato editável como dado (`JSON` do `.side`) eliminou defeitos de sintaxe e indentação e viabilizou edição assistida segura. **(2) Nem toda auto-cura é desejável:** promover automaticamente qualquer seletor que "funcione imediatamente" degrada a suíte, pois alvos instáveis (e.g., `j_idt` em *JSF*) quebram na *release* seguinte — curar com critério vale mais do que curar sempre. **(3) Corrigir durante a execução supera a análise *post-mortem*:** intervir pontualmente nos testes em execução mostrou-se mais eficaz do que analisar falhas depois do fato, sobretudo dada a complexidade do sistema. **(4) Resiliência exige conservadorismo:** endurecer a interação (desbloquear campos, forçar valores) resolve instabilidades reais, mas deve ser registrado no *log* técnico para não mascarar defeitos legítimos do sistema sob teste.

5 Conclusão

Nossa experiência sugere que a potencial obsolescência de ferramentas não implica necessariamente em descartar o conhecimento nelas acumulado e seus benefícios. Ao tratar o acervo de testes legado como principal referência e construir, ao seu redor, uma camada de execução resiliente e de depuração assistida, foi possível preservar anos de regras de negócio codificadas em testes e, ao mesmo tempo, modernizar a infraestrutura subjacente. Acreditamos que esse padrão — migração incremental guiada por dores operacionais — é transferível a outras organizações que enfrentam a descontinuidade de ferramentas de teste. Como trabalhos futuros, pretendemos instrumentar métricas de cura (taxa de sucesso por tipo de seletor, tempo de manutenção antes/depois em *releases* reais); incorporar *Data-Driven Testing*, executando o mesmo fluxo `.side` com múltiplas parametrizações via planilhas externas; e amadurecer o modo de execução não interativo (*Pipeline Mode*) para integração contínua (*CI/CD*), fechando o ciclo entre a criação assistida e a homologação automatizada.

Disponibilidade de Artefatos

Não são disponibilizados artefatos associados a este trabalho por razões éticas e legais. A tecnologia apresentada opera no contexto de uma instituição pública, manipulando dados sensíveis e informações protegidas por sigilo institucional.

Referências

[1] Healenium. 2024. Healenium: Self-Healing Library for Selenium-based Automated Tests. https://healenium.io. Projeto de código aberto. Acesso em: 2026.

[2] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2013. Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE 2013)*. IEEE, Koblenz, Germany, 272–281. doi:10.1109/WCRE.2013.6671302

[3] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2016. Robula+: An Algorithm for Generating Robust XPath Locators for Web Testing. *Journal of Software: Evolution and Process* 28, 3 (2016), 177–204. doi:10.1002/smr.1771

[4] Selenium Project. 2024. Selenium IDE: Open Source Record and Playback Test Automation for the Web. https://www.selenium.dev/selenium-ide/. Acesso em: 2026.

[5] Bruno Souza, Rebeca Motta, Jessica Costa, Elaine Nunes, Gustavo Pinto, and Rafael Mello. 2025. A Journey of Functional Test Evolution in the Public Sector. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 147–149. doi:10.5753/sast.2025.13869