HMI TEST AUTOMATION MODEL IN ATC SYSTEM

Paulo Henrique de Almeida Soares Pimenta
Atech Soluções Tecnológicas S.A.
Tel: +551130103460; email: phpimenta@atech.com.br

Camila Lainetti de Morais
Atech Soluções Tecnológicas S.A.
Tel: +551130103460; email: cmorais@atech.com.br

Caio Cesar Fattori
Atech Soluções Tecnológicas S.A.
Tel: +551130103460:1179; email: cfattori@atech.com.br

ABSTRACT

ATC systems are considered critical systems and, due to the complexity of their activity fluxes, they require tests that consumes a lot of human and computational resources. Automating tests in HMI reduces the workload of software testers and the time between the development and the search for failures. This task demands a deep knowledge of programming language which is not required from test analysts. At the same time, the identification of failures is not required from developers. Such automation can occur optimally through the integration of developers and test analysts. The objective of this article is to present a test automation model in critical systems that allows test analysts to implement HMI test cases with no need a deep knowledge of the programming language, by using a set of functions which was implemented by developers from testers request.

Keywords: ATC System, automation testing, test development, test construction, test automation, IHM test

1. INTRODUÇÃO

Uma importante etapa no ciclo de desenvolvimento de produtos é aquela na qual se realizam os testes que irão verificar e validar os requisitos do projeto. Havendo falhas nessa fase, o produto retorna à equipe de desenvolvimento e as devidas correções devem ser implementadas. Tal prática é, portanto, imprescindível para sistemas críticos, uma vez que diminui possíveis prejuízos e falhas que podem acarretar em danos ambientais e humanos. Contudo, a confecção de casos de testes torna-se difícil a medida que se aumentam a quantidade e a complexidade dos requisitos de sistema (Fillyaw, Friedman, Prabhu, 2007).

Em determinados setores produtivos estima-se que 40% do valor final do produto é oriundo do desenvolvimento do software (Fillyaw, Friedman, Prabhu, 2008). Tal valor é consideravelmente alto e concentrando em somente um ponto. Neste sentido a verificação de possíveis falhas ou quebra de requisitos são necessárias e justificadas.

Os testes avaliam o comportamento esperado e não esperado de uma execução. Softwares possuidores de interface homem-máquina (IHM), possuem como principal característica a interação do usuário e elementos de comando e monitoramento presentes na tela. No presente artigo será apresentada uma metodologia usada para automatização de testes no SAGITARIO, um software ATC (do inglês *Air Trafic Control*, ou controle de espaço aéreo) possuidor de IHM.

Nessa metodologia, são stakeholders da automação os desenvolvedores e os analistas de teste. O primeiro conhece bem a linguagem de programação, enquanto o segundo possui o conhecimento de como criar testes e encontrar falhas. Por seus conhecimentos da linguagem de programação, os desenvolvedores tem papel de manter e propor mecanismos facilitadores para elaboração dos casos de teste. Já os analistas de teste tem papel de elaborar os casos de teste de acordo com os requisitos de sistema e de software.

A metodologia idealizada consiste, principalmente, na criação de uma ferramenta simplificada de implementação de testes automatizados, que foi chamada “linguagem de interface”. O objetivo da criação desse instrumento (e também dessa metodologia) é possibilitar o isolamento da criação e implementação dos testes automatizados com os analistas de testes. Todos requisitos apresentados na seção metodologia tem como finalidade prover recursos para os analistas de teste para auxiliar a implementação de casos de teste automatizados.

Os analistas de teste usam a linguagem de interface para cumprir os passos descritos nos casos de teste através das funções que promovem interação com a tela do software. Essas funções foram criadas pelos desenvolvedores que mapearam os elementos das telas do software seguindo um padrão que será discutido com detalhes na seção Metodologia.

O desenvolvimento de testes automatizados por meio da linguagem de interface, representa um vários ganhos como:

- Isolamento dos testes com os analistas, evitando que o teste seja realizado de forma alinhada a solução proposta ou código implementado;
- Maior integração entre as equipes de desenvolvimento e testes criando um ambiente em que a troca de experiências otimiza os resultados.

Além disso, são mantidas as vantagens várias da automatização de testes, como a diminuição do tempo de execução (com reduções consideráveis de tempo) e aumento dos horários de execuções, como finais de semana, feriados e outros momentos sem expediente.

2. BACKGROUND

A automação de testes consiste em usar software para executar testes de programas computacionais, em complemento ou substituição de realizar a mesma tarefa manualmente. É um processo que visa reduzir o tempo e o esforço despendidos em testes manuais, e que oferece muitos ganhos. A automação, porém, pode ser menos eficaz e tendenciosa caso seja implantada pelos próprios desenvolvedores que terão seu código testado, uma vez que estes conhecem os

pormenores da implementação, o que dificulta idealizar o teste de maneira neutra. Visando examinar com mais eficiência o programa em teste, é proposto nesse artigo uma metodologia de criação de testes automatizados que permite isolar a geração dos casos de teste com os analistas de teste.

Dessa maneira, são stakeholders na criação de casos de testes automatizados o *desenvolvedor* e o *analista de testes*.

Essa pesquisa se faz oportuna pois é grande a necessidade de garantir eficácia nos testes de software, especialmente no caso de sistemas críticos como o SAGITARIO. A metodologia desenvolvida também mitiga a necessidade de consumir recursos, como a contratação de novo desenvolvedor para atuação apenas em implementação de testes.

2.1. Desenvolvedores

Sobre o desenvolvedor, é possível afirmar que é de seu perfil o amplo conhecimento de lógica e linguagem de programação. Nada pode-se dizer, porém, sobre sua habilidade em identificar falhas, uma vez que essa característica não é exigida para o desempenho de sua função.

Para a equipe de desenvolvedores essa forma de criação de testes automatizados é interessante porque retira deles a responsabilidade de automatizar um teste do próprio código, o que pode levar à maior dificuldade de encontrar falhas. Além disso, também reduz o tempo que a equipe usa automatizando testes, permitindo o foco em desenvolvimento de *features* e correção de *bugs*. Outra vantagem do processo de criação é a possibilidade de integração contínua. A partir dela, o desenvolvedor terá seu código testado a cada interação, podendo verificar quais requisitos ele está cumprindo.

O papel do desenvolvedor nessa metodologia será dar ferramentas para que o analista de teste, que será comentado a seguir, sozinho, tenha como produzir um caso de teste automatizado programado.

2.2. Analista de testes

Sobre o analista de testes, é possível afirmar que é de seu perfil identificar falhas, porém nada garante seu conhecimento de linguagem de programação, pois não é requisito para realização de suas tarefas.

Para a equipe de testes, essa forma de criação de testes automatizados é potencialmente vantajosa pois viabiliza que o analista produza casos de teste e o automatize sem necessitar da intervenção do desenvolvedor.

O papel do analista de testes, portanto, será não apenas descrever o caso de teste, mas também implementar sua automatização através de ferramentas pré-construídas pelo desenvolvedor. Também será dele a responsabilidade de fazer a manutenção ao longo do tempo dos casos de teste criados, incluir os testes novos no software de build contínuo e ainda acompanhar os resultados do mesmo, a fim de localizar e descrever falhas.

3. METODOLOGIA DE CRIAÇÃO

A metodologia de criação de casos de teste automatizados baseado nos stakeholders define cinco requisitos:

- Os elementos da tela devem ser criados no caso de teste com suas respectivas classes e organizados na mesma forma como apresentados na tela;
- As ações e verificações associadas aos elementos da tela devem ser implementadas como métodos associados às classes e devem ser compostas por um conjunto de ações e verificações atômicas que retornem uma resposta booleana;
- As macroações e macro-verificações devem conter nomes conhecidos e devem ser implementadas no nível de equipamento;
- Todas macroações e macro-verificações devem ser documentadas de maneira a possibilitar seu uso;
- Os casos de teste devem ter um fluxo sequencial e simplificado.

Os quatro primeiros requisitos viabilizam a realização do último requisito, sendo que esse é a forma adotada para garantir

que tanto os analistas de teste sejam capazes de compreender os casos de teste automatizados, quanto sejam capazes de implementá-los por si próprios.

A seguir, cada um dos requisitos da metodologia serão apresentados em detalhes.

3.1. Elementos da interface

O primeiro requisito específico refere-se a como os elementos de interface devem ser acessíveis para o usuário. Primeiro, os elementos devem ser caracterizados quanto a seu tipo ou suas funções, ou seja, dois botões diferentes podem provocar resultados diferentes na interface, mas ambos tem a mesma função que é ser clicado. O conceito se estende ao demais elementos, como caixa de texto, caixa de seleção, janela, aba e outros. Com essa separação, o mapeamento dos elementos pode ser feito usando as funções disponíveis pela linguagem da interface. Dessa forma, pode-se disponibilizar essas mesmas funções para o elemento durante o teste.

Alguns objetos no teste não são da mesma classe na interface, seja por serem reimplementações particulares da classe ou por serem realmente outra classe já existente que foram utilizadas por questões estéticas, principalmente. Essa é a analogia que se pode fazer entre um link e um botão. Do ponto de vista de classe de interface, um link tem semelhança a uma legenda, mas do ponto de vista operacional, um link tem as mesmas funções de um botão.

Dessa maneira, o mapeamento dos elementos deve levar em conta, principalmente, suas funções.

Outro ponto desse requisito é a organização dos elementos. Dado que os analistas tem conhecimento dos elementos que estão na interface, eles o conhecem hierarquicamente na ordem de sua ocorrência; ou seja, se um botão aparece na interface dentro de uma janela, para o analista o botão pertence à janela e o caminho para acessar o botão passa antes pelo acesso à janela.

Mesmo que essa hierarquia não exista do ponto de vista de implementação da interface e dos testes, a organização do acesso aos elementos no teste pode ser feita da mesma maneira como visualizada, ou seja, se um botão está dentro de uma janela, o acesso ao botão em código deve ser $janela \rightarrow botão$.

Da mesma forma, caso uma janela seja aberta a partir de uma ação realizada dentro de outra janela, a maneira mais fácil de localizar essas janelas é organizando-as de forma dependente, $janela \rightarrow janela$.

Com isso, o acesso a um elemento depende, além de conhecer a função do elemento, conhecer a organização dos objetos na tela. Esse é um conhecimento que se pressupõe para analistas de teste. A figura 1 representa o mapeamentos por esse requisito.

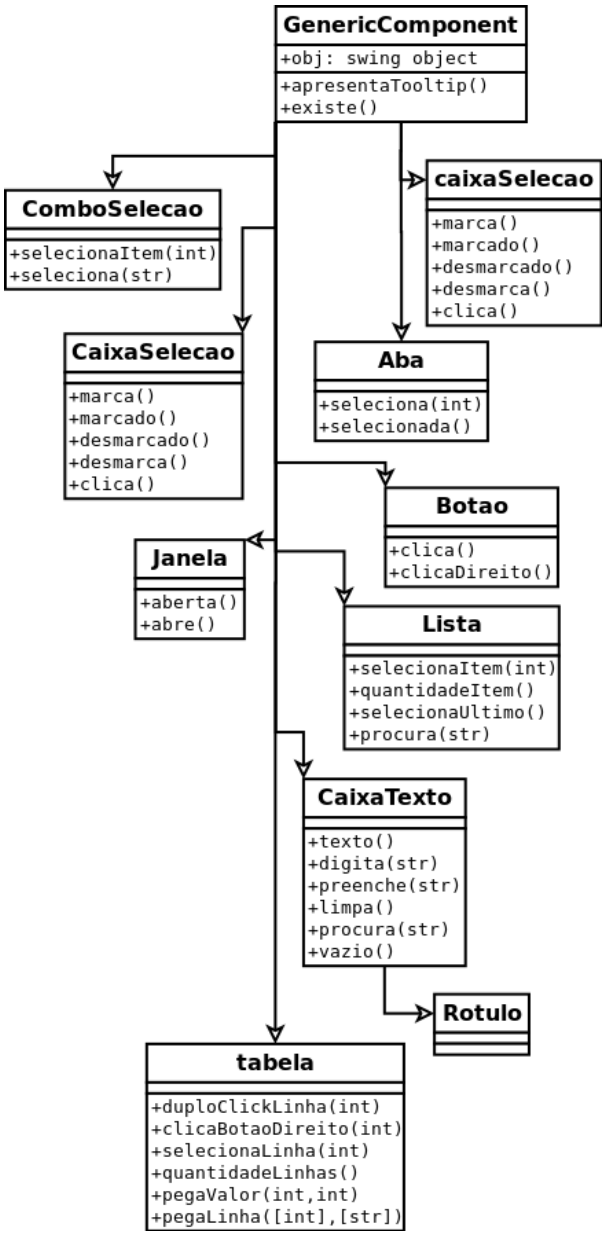


Figura 1 - Mapeamento dos tipos de objetos da tela

3.2. Ações e verificações

O segundo requisito envolve as funções dos elementos.

A ação de um elemento é, por exemplo, o clique de um botão. A essa ação estão associadas algumas outras ações e verificações que tornam o fluxo mais complexo.

Por exemplo, para clicar em um botão de uma janela, primeiro é preciso verificar se a janela está aberta. Caso a janela não esteja aberta é preciso abrir a janela e, caso a tentativa de abrir a janela provoque uma falha, deve-se retornar que a ação falhou. Nos demais casos, ainda é preciso verificar se o botão está visível e habilitado para ser clicado. Caso não esteja, a função também deve retornar a falha. Ainda, por fim, caso todas as condições anteriores estejam atendidas, a própria ferramenta de teste pode ter um comportamento falho ao tentar clicar no botão e esse comportamento também deve ser retornado. Caso nenhuma falha ocorra, a ação de clicar no botão retorna sucesso.

Por isso, uma ação dos elementos é na verdade um fluxo não sequencial de ações e verificações de menor grau, consideradas atômicas.

Os analistas de teste fazem essas mesmas ações e verificações atômicas de modo "natural", pois esses são apresentados visualmente indicados para ele, ou seja, um analista sabe que um botão não pode ser clicado por não estar vendo ele na janela ou mesmo não vendo a janela, e ao mesmo tempo, sabe que deve abrir a janela antes. Como a ferramenta de teste não tem essa mesma capacidade de interpretar o indicador visual, ela efetua esse processo no nível das funções e propriedades que o elemento tem no nível de código da interface.

Caso essas ações e verificações atômicas fossem incluídas no código do caso de teste, esse ficaria complexo pela quantidade de fluxos e demandaria dos analistas de teste o conhecimento da lógica da linguagem de programação.

3.3. Macroações e macro-verificações

As macroações são ações que não estão associadas a um só elemento, e dependem de um conjunto de ações de vários elementos de interface, além de outras lógicas que aumentam a complexidade do processo.

Essas macroações devem ser nomeadas de forma conhecida. Por exemplo, a troca de marcha em um carro inclui as ações de verificar a velocidade, retirar o pé direito do pedal do acelerador, colocar o pé esquerdo no pedal da embreagem, verificar que o carro está desengatado, mover o câmbio para a posição desejada, verificar que houve encaixe na posição do câmbio e, por fim, em paralelo, retirar o pé esquerdo do pedal da embreagem enquanto coloca o pé direito no pedal do acelerador, observando o giro do motor.

Esse processo de exemplo tem uma complexidade quando descrita e é composto de ações e verificações que não são atômicas. Além disso, é representada num fluxo que pode variar de acordo com modelo de equipamento ou de técnica. Entretanto, o resultado esperado dessa macroação, que no exemplo é a troca de marcha, é conhecido por todos os usuários desse equipamento pelo nome "Troca de marcha".

Esse motivo que faz essas ações requererem o nome conhecido.

Ao mesmo tempo, como essas macroações não são vinculadas a um único elemento da interface, mas a um conjunto de elementos, define-se as macroações como ações do nível do equipamento.

Algumas das macroações e macro-verificações são dependentes do tempo, como as verificação de trocas de mensagens, processamento de mensagem, tempos de espera e outras que incluem uma indeterminação no fluxo.

Isso significa que não é possível colocar um tempo de espera para realização da ação, pois esse tempo sofreria influência de fatores externos. Um valor conservativo de aguardo (maior do que o máximo tempo de espera possível) também não é uma boa solução, pois tornaria o processo de teste automatizado demorado.

Então, para esses processos dependentes do tempo, as macroações utilizam um fluxo de loop de verificações com pequenos tempos de aguardo, em que a cada espera, o teste de tempo indeterminado é refeito para saber se a

verificação resultou correta, ao invés de retornar falha após uma verificação.

Como não há garantias de que essa etapa de verificação está correta, é preciso incluir ao loop uma cláusula de timeout, em que após uma quantidade de tentativas de verificação sem sucesso, a macroação interrompe a verificação e retorna a falha.

Essas macroações e macro-verificações são a base para formulação de novos casos de teste, pois sua proximidade do conhecimento dos usuários faz com que um processo de teste escrito em linguagem não formal possa ser replicado para a linguagem de programação do caso de teste.

Outra possível macroação é uma sequência repetitiva de ações que não necessariamente são descritas como uma coisa só. Por exemplo, ver a cor e o modelo de um carro são ações completamente separadas, mas podem ser analisadas ao mesmo tempo em muitas situações.

Dessa forma, criar uma macroação e/ou macro-verificações para processos independentes, mas que podem ser associados, facilitam no elaboração de novos casos de teste.

3.4. Documentação das macroações e macro-verificações

Feita conjuntamente com a criação de seu código, a documentação das macroações e macro-verificações deve ser clara e objetiva em relação aos passos que ela realiza. Cada ação feita, como clicar em um botão ou abrir uma janela deve ser descrita.

Para fins de exemplo, é dado o caso em que analistas de teste de um software necessitem testar uma funcionalidade dentro de uma janela chamada “Janela1”. Supõe-se que todo teste comece com o software e um painel de botões abertos, no qual se acessam todas as janelas a serem testadas, chamado “Painel Principal”.

Para fazer o teste do exemplo, é necessário, em primeiro lugar, abrir a Janela1. O botão do PainelPrincipal que abre a Janela1 é chamado “botãoAbreJanela1”. Ele deve ser clicado para o início do teste.

Sabe-se agora que o teste a ser executado será a verificação de funcionamento do botão que está dentro da Janela1 e a fecha, chamado “botãoFechaJanela1”.

Pode-se dividir esse teste em duas partes. A primeira parte será uma função para abrir a Janela1, que pode ser comum a todos os testes que interagem com a Janela1. A segunda parte, específica para o teste, verificará a funcionalidade do botãoFechaJanela1 de fechar a Janela1.

Sendo assim, o teste será dividido entre as funções “AbreJanela1”, uma macroação simples, e “VerificaBotãoFechar”, uma macro-verificação simples. Suas documentações devem descrever com precisão o que cada uma delas faz e como elas o fazem.

A documentação da macroação “AbreJanela1” será como o seguinte:

“Essa função abre a janela com o nome “Janela1”. Ela retorna o booleano True caso verifique que a Janela1 está visível e False caso verifique que a Janela1 não está visível. Para ter esse retorno, a função segue os seguintes passos:

- O botãoAbreJanela1 do PainelPrincipal é clicado;
- É retornado o atributo visibilidade da Janela1.”

É possível verificar que a documentação da macroação tem três partes. Primeiro é descrito o seu funcionamento, dizendo 1) que ação realiza e 2) como funciona seu retorno. Por fim, lista a sequência de passos que descreve.

A documentação da macro-verificação “VerificaBotãoFechar” é assim descrita:

“Essa função verifica se o botão chamado “botãoFechaJanela1” que se encontra dentro da Janela 1 de fato tira a visibilidade da Janela 1. A função retorna o booleano True caso seja verificado que o botãoFechaJanela1 realmente retira a visibilidade da Janela1 e False caso contrário. A sequência de ações que a função realiza para fazer essa verificação é a seguinte:

- O atributo visibilidade da Janela1 é verificado. Caso seja True, os próximos passos serão executados, pois sabe-se que a Janela1 está aberta. Caso contrário, a função retornará False pois já é sabido que não é possível tentar fechar a Janela1

uma vez que esta já encontra-se fechada;

- O botãoFechaJanela1 é clicado;
- É retornado True caso o atributo visibilidade da Janela1 for False e vice-versa.”

A documentação da macro-verificação também tem três partes. Em primeiro lugar é descrita qual a verificação que será feita. Em seguida, conta-se como seu retorno é produzido. Por fim, lista-se os passos descritos pela função e como as verificações são feitas.

Dessa maneira, fica claro que a documentação segue um padrão para ambos os tipos de funções da linguagem de interface, e que esta auxilia o analista de teste na implementação de um teste automatizado. Para isso, o analista irá encontrar a função correta para seu objetivo pelo seu nome facilitado e pela comparação dos passos que faria manualmente com os passos descritos pela documentação da função.

3.5. Caso de teste sequencial e simplificado

Os requisitos anteriores tendem a viabilizar o atendimento desse requisito.

O propósito de um caso de teste ser sequencial é reduzir a necessidade do elaborador de um caso de teste de conhecer a lógica de programação da linguagem em que o caso de teste é implementado.

Na maioria das linguagens, o fluxo sequencial é mais simples de ser compreendido, pois utiliza rotinas escritas uma linha após a outra. Assim, sabe-se exatamente o instante em que uma rotina terminou e a outra pode começar, e a determinação do estado atual do teste pode ser determinado de forma mais simples.

Isso não significa que todo o caso de teste tenha sido executado em uma sequência, mas que os fluxos não sequenciais no nível de código do caso de teste foram escondidos. Assim, cabe aos analistas determinar apenas qual macroação da sequência que ocorreu falha

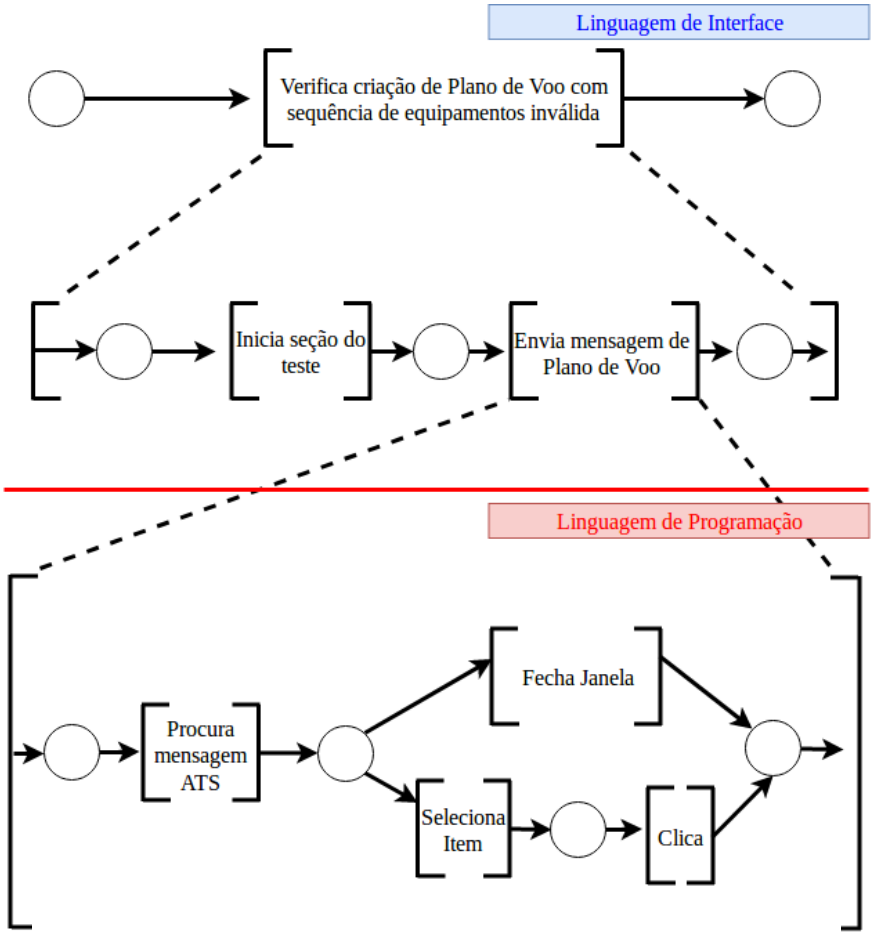


Figura 2 - Exemplo de metodologia top-down

e que demanda uma verificação mais detalhada.

A metodologia de desenvolvimento para esse requisito é a metodologia top-down, em que um processo é descrito no nível mais alto como uma só atividade e essa atividade deve ser expandida para níveis mais abaixo em fluxos um pouco mais complexos e detalhados.

O código do caso de teste deve estar em um nível da metodologia top-down em que todas as atividades ainda sejam sequenciais.

A figura 2 mostra um exemplo dessa metodologia top-down e o local em que se encontra o caso de teste.

Pela lógica adotada nessa metodologia, um caso de teste é considerado em falha se uma das macroverificações apresentar resultado de falha. Internamente a macroverificações muitas falhas podem ocorrer e o seu resultado ainda assim apresentar sucesso. Esse é o caso, por exemplo, de um fluxo iterativo, em que o fluxo aguarda certa condição ser verdadeira por tempo determinado. A falha da macroverificação, nesse caso, é dada por não ter o recebimento de sucesso dentro do período determinado.

4. RESULTADOS

A partir dos casos de teste eleitos para serem automatizados (*citar artigo anterior*), iniciou-se a implementação das classes necessárias, o mapeamento dos elementos da tela e codificação das macroações e macro-verificações genéricas. Dessa forma, à medida que os testes eleitos iam tendo suas funções implementadas, elementos do SAGITARIO como janelas, botões, caixas de textos e demais tiveram suas classes incluídas e seus objetos mapeados.

A partir do mapeamento e das macroações e macro-verificações é criada a linguagem de interface, que será em seguida usada pelos analistas de teste. Ela evolui a cada caso de teste implementado, incluindo cada vez mais elementos de tela do SAGITARIO e mais funções criadas para a execução de tarefas dentro dos passos de teste.

O enriquecimento da linguagem de interface, somado à finalização da fase inicial de configuração de ambiente de testes e ao aumento de experiência dos desenvolvedores com o uso das ferramentas, foi tornando a implementação dos testes automatizados eleitos cada vez mais rápida, como pode ser notado pela figura 3.

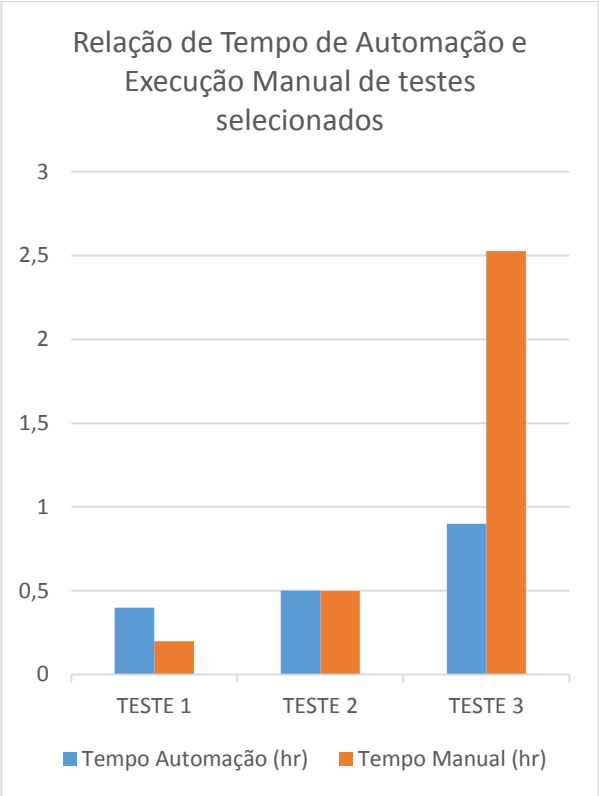


Figura 3- Gráfico da relação de tempo (em horas) de automação e execução manual de testes selecionados

Na figura 3 são mostrado três casos de teste implementados em momentos diferentes. Eles foram selecionados dado terem Tempo Automação e Tempo Manual típicos dos mesmos, respectivamente, no início, meio e fim de suas codificações. O Tempo Automação refere-se ao tempo gasto para programar o caso de teste automatizado e o Tempo Manual refere-se ao tempo gasto pelos analistas de teste para executar o caso de teste manual.

É bastante visível que no início desse processo de implementação o tempo gasto para automatizar um teste era relativamente alto dado que os testes eram simples, com tempo de execução manual baixo. No meio do tempo de implementação o tempo de automação foi se aproximando e chegou a se igualar com o tempo Manual. Já no fim do processo de implementação, quando já estão sendo codificados testes mais extensos e complexos, os desenvolvedores muitas vezes conseguiram

superar o Tempo Manual no Tempo Automação.

Além da linguagem de interface, cuja evolução impactou muito na agilidade crescente de implementação dos testes, outra melhoria a ser mencionada é a preparação do ambiente pela suíte do caso de teste automatizado. Foram feitos agrupamentos de casos de teste automatizados que utilizavam as mesmas configurações de ambiente e automatizadas as mesmas, em linguagem de interface, economizando tempo de execução dos testes e possibilitando que os analistas de testes também configurem seu ambiente no futuro. A figura 4 ilustra um caso de teste automatizado simples, que será comentado a seguir.

```
# -*- coding: utf-8 -*-
import sys
from os import path

sys.path.append(path.dirname( path.dirname( path.abspath(path.split(__file__)[0]))))
from lib.master import Master

def main():
    master = Master([])
    BDS = "APPBRTMA"

    #TESTE 1
    mensagem_teste = """1- (PLJ - BDS-A) <Apresenta mensagem: Atenção. A soma dos setores
    Mistos cadastrados não cobre a totalidade do espaço aéreo da TMA.(SAG_F1_0713)
    SRS0678 SRS0679 MSG0055"""
    master.iniciaSecao(mensagem_teste)
    gbds = master.abreGBDSGeografico(BDS)
    gbds.Geografico.espacoAereo.clica()
    gbds.Geografico.ValidaEspacoAereo.clica()
    verif=gbds.verificaMSGvalidacao()
    gbds.Geografico.FechaValidaEspacoAereo.clica()
    gbds.fechaGBDSGeografico()
    master.teste(verif, "O espaço aéreo da BDS selecionada é apresentado.")
    master.finalizaSecao()
```

Figura 4 - Exemplo de teste automatizado

Nesse teste do exemplo é possível notar várias das funções citadas durante as seções Metodologia e Resultados. A inicialização do teste, que é feita através da função “master.iniciaSecao(mensagem_teste)”, é um exemplo de macroação. Ela envia a mensagem de descrição do teste para um log de identificação. Outro exemplo de macroação é a abertura de uma seção da posição GBDS, responsável pelo gerenciamento do banco de dados, chamada através da função “master.abreGBDSGeografico(BDS)”. É também possível ver uma macro-verificação nesse caso, responsável pela verificação de funcionalidade no teste. Ela é chamada como “gbds.verificaMSGValidacao()” e afere se uma determinada mensagem de validação é verificada.

Uma integração contínua foi adicionada à implementação dos casos de teste automatizados. Usando servidor open-source automatizado, os casos de testes implementados foram continuamente disponibilizados para sua mais breve possível utilização.

Para os próximos casos de testes automatizados, é colocado que assim que são implementados pelos analistas, são logo incluídos em uma suíte de testes conveniente e, em seguida, a suíte é adicionada ou atualizada na máquina em que roda o servidor de automação. A suíte é responsável por parte do processo de integração contínua, uma vez que nela o software é chamado, as condições necessárias são configuradas e em seguida os testes são executados.

Sobre as suítes, os analistas são treinados para, uma vez que o teste está pronto, usar rotinas previamente programadas para construí-las. Eles igualmente são orientados a como incluir as suítes na máquina e configurar a execução regular daquelas que são de seu interesse, utilizando aplicação web do servidor de automação.

Para os desenvolvedores, a utilização da integração contínua é bastante simples, bastando a agregação de seus códigos ao software e a execução das suítes utilizando a aplicação web disponibilizada pelo servidor da

automação, já com as suítes de teste disponibilizadas pelos analistas.

A aceitação da linguagem de interface por parte dos analistas de teste foi extremamente importante para o sucesso da pesquisa. Este resultado se deu por conta da facilidade de uso da linguagem e da colaboração entre as equipes de desenvolvimento e de analistas de testes.

Sendo assim, a metodologia criada não só gerou uma integração entre ambas as equipes, mas também um esforço conjunto de tornar melhor e mais precisa a verificação, validação e correção de novas versões geradas do SAGITARIO.

5. CONCLUSÕES

A automação de testes utilizando a metodologia apresentada alcançou parcialmente, até o momento, os objetivos traçados pela equipe idealizadora desse processo. O isolamento da criação e implementação dos testes automatizados com os analistas de testes está ainda em implementação, mas a aceitação deles da linguagem de interface acena positivamente no sentido de alcançar plenamente as metas.

A integração entre as equipes de desenvolvimento e testes também foi realizada, dado que foram aumentadas suas responsabilidades codependentes.

Durante a implementação da metodologia, foi também concluído que a nomeação mnemônica das funções de macroação e macro-verificação e sua documentação completa e cuidadosa é fundamental para a utilização das mesmas, uma vez que essas não foram implementadas pelas pessoas que as usarão. A utilização de palavras chaves para o encontro facilitado da ação procurada na documentação também é relevante.

Em relação a quantidade de casos de teste possíveis de implementar com a utilização exclusiva das funções criadas até então, é difícil prever com precisão qual porcentagem coberta. Sabe-se que a área de cobertura é ampla, pois muitas macroações comuns foram implementadas e uma quantidade muito alta de elementos foi mapeada. Ainda assim, só a contínua implementação de casos de testes dará melhor dimensão do conteúdo faltante.

Por fim, em relação ao diferencial de tempo economizado utilizando o processo descrito, não apenas automatizando os casos de testes testes, é concluído que foram bastante importantes dois fatores. O primeiro é o fato da linguagem de interface incorporar a etapa de configuração de ambiente, pois apenas uma configuração servirá para toda uma suíte com vários casos de teste. O segundo é a integração contínua, que imprime facilidade e agilidade na execução de testes múltiplos testes já organizados.

Assim, apesar de não haver alcançado em totalidade os objetivos ainda, a inclusão dessa forma de automatização de casos de testes foi considerada positiva pelos ganhos até o momento e pela previsão do isolamento completo da automatização dos casos de teste com os analistas de testes a curto prazo.

6. REFERENCIAS

IVAO. International Virtual Aviation Organization. Available at: <http://www.ivao.aero>. Accessed on 23 Mar. 2009.

FURTADO, V. H. et al. “Aspectos de Segurança na Integração de Veículos Aéreos Não Tripulados (Vant) No Espaço Aéreo Brasileiro”. Aerial Transportation Symposium, 2008, Rio de Janeiro. *Annals of the VII SITRAER*. Rio de Janeiro: E-Papers, 2008. v. 1 and 2. p. 92-99.

VISMARI, L. F. Vigilância dependente automática no controle de tráfego aéreo: avaliação de risco baseada em modelagem em redes de Petri fluidas e Estocásticas. 2007. Dissertation (Master’s Degree) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2007.