UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GIOVANE GALVÃO

MARIA CAROLINA DE OLIVEIRA

UTILIZANDO TESTES DE ACEITAÇÃO AUTOMÁTICOS NO DESENVOLVIMENTO DE UMA APLICAÇÃO *WEB*

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2014

GIOVANE GALVÃO

MARIA CAROLINA DE OLIVEIRA

UTILIZANDO TESTES DE ACEITAÇÃO AUTOMÁTICOS NO DESENVOLVIMENTO DE UMA APLICAÇÃO *WEB*

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Richard Duarte Ribeiro

Coorientador: Prof. Dr. Willian Massami Watanabe

PONTA GROSSA

2014

TERMO DE APROVAÇÃO

UTILIZANDO TESTES DE ACEITAÇÃO AUTOMÁTICOS NO DESENVOLVIMENTO DE UMA APLICAÇÃO WEB

por

GIOVANE GALVÃO
MARIA CAROLINA DE OLIVEIRA

Este Trabalho de Conclusão de Curso foi apresentado em 13 de Novembro de 2014 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. Os candidatos foram arguidos pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
RICHARD DUARTE RIBEIRO
Prof. Orientador

_____
WILLIAN MASSAMI WATANABE
Membro titular

_____
LUIZ RAFAEL SCHMITKE
Membro titular

_____
Profª. Drª. Tânia Lúcia Monteiro
Responsável pelos Trabalhos de Conclusão de Curso

_____
Profª. Drª. Simone de Almeida
Coordenadora do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

– O Termo de Aprovação assinado encontra-se na Coordenação do Curso –

AGRADECIMENTOS

Agradecemos primeiramente a Deus por sua bênção, amor, carinho e cuidado. O Senhor nos trouxe amparo nos momentos difíceis, esperança e força quando achamos que nada daria certo.

Aos nossos pais, pela paciência, amor, educação e incentivo. Obrigada pelos conselhos e pelo carinho. Nosso eterno respeito e amor a vocês, papai e mamãe.

Dedicamos especial agradecimento aos professores Willian Massami Watanabe e Luiz Rafael Schmitke pela ajuda oferecida ao longo do desenvolvimento das pesquisas.

Nosso muito obrigado ao professor Richard Duarte Ribeiro pela paciência e incentivo na orientação deste trabalho.

Aos demais professores, nossa gratidão pela dedicação ao ensino, pela paciência aos atrasos, e pelo incentivo aos estudos.

Não tem como esquecer de vocês, nossos amigos e colegas. Nosso muito obrigado pela cumplicidade, confiança, e afeto.

Finalmente, agradecemos a todos que direta ou indiretamente fizeram parte da nossa formação.

RESUMO

GALVÃO, G., OLIVEIRA, M.C. **Utilizando testes de aceitação automáticos no desenvolvimento de uma aplicação *Web***. 2014. 81f. Trabalho de Conclusão de Curso de Tecnologia em Análise e Desenvolvimento de Sistemas - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2014.

Este trabalho aplicou testes de aceitação automáticos juntamente com a prática de desenvolvimento de *software* conhecida como *"Integração Contínua"*, muito utilizada nas metodologias ágeis. A realização foi feita através do seu uso no desenvolvimento de uma aplicação *Web* baseada em *JavaScript*, *PHP* (*HyperText Preprocessor*), *CSS* (*Cascading Style Sheets*), *HTML*(*HyperText Markup Language*) e SGBD (Sistema Gerenciador de Banco de Dados) *MySQL*. Foram elaborados testes de aceitação adotando as ferramentas *Selenium IDE* e *Ant*. A ferramenta *Git* foi utilizada para efetivar o controle de versão através do *GitHub*. Ao final deste trabalho verificou-se que o *feedback* instantâneo dos erros encontrados no desenvolvimento de uma aplicação contribuiu para a agilidade do processo.

Palavras-chave: Testes. Aceitação. Automatização. Desenvolvimento. *Software*. Integração Contínua. Métodos Ágeis.

ABSTRACT

GALVÃO, G., OLIVEIRA, M.C. **Using automated acceptance tests in the development of a *Web* application**. 2014. 81s. Tecnologia em Análise e Desenvolvimento de Sistemas – Federal University of Technology – Paraná. Ponta Grossa, 2014.

This work applied automated acceptance tests with the software development practice known as "Continuous Integration", often used in agile methodologies. The realization was made through its use in developing a web application based on JavaScript, PHP (Hypertext Preprocessor), CSS (Cascading Style Sheets), HTML (HyperText Markup Language) and DBMS Manager (System Database) MySQL. We acceptance tests developed by adopting the tools Selenium IDE and Ant. The Git tool was used to enforce versioning via the GitHub. At the end of this work it was found that the instant feedback of the errors encountered in the development of an application contributed to the faster process.

Keywords: Test. Acceptance. Automated. Development. Software. Continuous Integration. Agile methods.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIAÇÕES E SIGLAS

XP - *Extreme Programming*

SI – Sistemas de Informação

CSS – Cascading Style Sheets

HTML – *Hypertext Markup Language*

PHP – Hypertext Preprocessor

CI – *Continuos Integration*

IC – Integração Contínua

CVS – *Concurrent Versions System*

Selenium IDE – *Selenium Integrated Development Environment*

SGBD – Sistema Gerenciador de Banco de Dados

Selenium RC – *Selenium Remote Control*

ATDD - Desenvolvimento Orientado a Testes de Aceitação

SUMÁRIO

1 INTRODUÇÃO

O projeto de *software* envolve uma série de atividades que incluem rotinas técnicas e administrativas, as quais precisam ser monitoradas frequentemente para que o produto final seja realizado dentro do prazo e dos custos inicialmente definidos pela empresa (PRESSMAN, 2011). Para o alcance deste objetivo visando a qualidade e a satisfação do cliente “é necessário o uso de métodos, técnicas e ferramentas” (NETO, 2014).

A área de Engenharia de *Software* define vários destes métodos e técnicas para o processo de produção de sistemas, porém, mesmo com a aplicação destas melhorias as organizações possuem a dificuldade em cumprir as metas do projeto (PRESSMAN, 2011).

Esta dificuldade aumenta quando o desenvolvimento é iniciado com falhas, causando impacto na confiabilidade e qualidade do produto final (DEUTSCH, 1979 apud PRESSMAN, 1995).

> “Erros podem começar a acontecer logo no começo do processo, onde os objetivos podem estar imperfeitamente especificados, além de erros que venham a ocorrer em fases de projeto e desenvolvimento posteriores. Por causa da incapacidade que os seres humanos têm de executar e comunicar com perfeição, o desenvolvimento de *software* é acompanhado por uma atividade de garantia de qualidade.” (DEUTSCH, 1979 apud PRESSMAN, 1995).

Perin (2006) afirma que “a atividade de teste de *software* é um elemento crítico da garantia de qualidade de *software* (...)” (PERIN, 2006). As empresas são incentivadas a ter uma planejada e minuciosa prática de testes quando percebem os altos custos gerados pelas falhas no *software*, fato que contribui para a economia e prevenção destes gastos (NIETO; CARDOSO, 2012).

Segundo Perin (2006) normalmente as empresas dedicam 40% de todo o projeto para a fase de testes. Em sistemas que podem envolver a segurança humana tais como os utilizados em controle de voo, os testes demandam “de três a cinco vezes mais que todos os outros passos da engenharia de *software* juntos.” (PERIN, 2006).

Este trabalho traz conceitos e sugestões para a implementação de testes automatizados utilizando a prática de Integração Contínua, visando maior agilidade para este processo tão importante na criação de sistemas.

1.1 OBJETIVOS

Estudar e implementar testes automatizados na fase de codificação de uma aplicação *Web* utilizando Integração Contínua, contribuir para a melhora no processo de desenvolvimento, bem como detectar e resolver erros encontrados no *software* ainda na fase de implementação. Para isso, definiram-se os seguintes objetivos específicos:

- Estudar os padrões de desenvolvimento de aplicações *Web*;
- Estudar técnicas de teste de *software* que devem ser utilizadas no processo de Integração Contínua;
- Estudar sobre Integração Contínua, aprendendo a utilizar sistemas de controle de versões e ferramentas de integração;

1.2 PROBLEMA

Segundo Pressman (2001 apud SOARES, 2004, p. 2), a primeira tentativa de organização para o processo de elaboração de *software* aconteceu com o surgimento do modelo Clássico ou Sequencial, o qual e compreende a elaboração de uma documentação em cada etapa cumprida do desenvolvimento. As metodologias clássicas, também chamadas de tradicionais ou orientadas a documentação, “devem ser aplicadas apenas em situações em que os requisitos do sistema são estáveis e requisitos futuros são previsíveis” (SOARES, 2004, p.1).

> “Os processos orientados a documentação para o desenvolvimento de *software* são, de certa forma, fatores limitadores aos desenvolvedores e muitas organizações não possuem recursos ou inclinação para processos pesados de produção de *software*. Por esta razão, as organizações pequenas acabam por não usar nenhum processo.” (SOARES, 2004).

No decorrer do desenvolvimento, os requisitos podem sofrer alterações, causando mudanças no código e no projeto (SOARES, 2004, p.1). Para empresas que possuem equipes de desenvolvimento pequenas e que utilizam o modelo clássico, a realização destas mudanças pode representar uma atividade de alto custo, podendo gerar atrasos na entrega do sistema. (SOARES, 2004, p.1).

O estudo realizado pelo The Standish Group (1995 apud SOARES, 2004, p. 1) chamados de “CHAOS report” mostra que apenas 16,2% dos 8380 projetos foram entregues com todos os requisitos corretos e dentro dos prazos e custos definidos na fase de planejamento. A porcentagem de cancelamento antes da conclusão do projeto alcançou 31%, e a maior parte – em torno de 52,7% - desta amostra resultou na entrega com custos e prazos maiores que os estipulados, além de não satisfazer o cliente com relação aos requisitos (THE STANDISH GROUP, 1995 apud SOARES, 2004, p. 1).

Como citado anteriormente, as metodologias tradicionais traziam grande dificuldade para alterações no projeto. Visto que esta metodologia não se adequava à realidade de muitas empresas, foram criados outros métodos de planejamento e documentação, tais como o *Scrum* (SCHWABER; BEEDLE, 2002) e a *XP* (BECK, 1999).

Estas metodologias foram nomeadas Metodologias Ágeis, as quais levam em consideração o fato de que além da demanda de tempo para a elaboração de um plano de desenvolvimento, é preciso focar nos envolvidos no projeto, como os desenvolvedores e suas iterações (SOARES, 2004, p.3). Realizar a inclusão do cliente no processo de desenvolvimento através de reuniões frequentes para a retirada de dúvidas referentes aos requisitos é um exemplo de prática destas metodologias (SOARES, 2004, p.3).

Mesmo com as mudanças propostas pelas metodologias ágeis, as empresas ainda tendem a ter muita dificuldade na questão administrativa e gerencial dos projetos (SANTOS, 2008, p.4). Quando se envolve falhas no *software*, a questão fica ainda mais delicada, pois a frustação do cliente é demasiada grande (FOWLER, 2006).

Desta forma, existem pontos cruciais do desenvolvimento que precisam de mais atenção, como, por exemplo, a integração de funcionalidades, os testes, o cumprimento dos requisitos e a implantação do sistema (FOWLER, 2006).

1.3 JUSTIFICATIVA

O mito mencionado por Beizer (1990, p.1 apud PRESSMAN, 1995, p.787) relata o fato de que os programadores estão sujeitos a cometer erros no código de um *software*. Mesmo com o uso das técnicas que visam a perfeição de um sistema e o não acarretamento de *bugs*, as verificações e revisões em um *software* devem ser realizadas (BEIZER 1990, p.1 apud PRESSMAN, 1995, p.787). Desta forma, a atividade de teste e o projeto de casos de testes são soluções que temos disponíveis para resolver estes problemas (BEIZER 1990, p.1 apud PRESSMAN, 1995, p.787).

Entre os principais benefícios do teste destacam-se (FUKUMORI; SANTOS; MORRO, 2008):

- Detectar falhas;
- Melhorar a qualidade do *software;*
- Tornar o sistema funcional e confiável;

A fim de aplicar as atividades de teste no processo de desenvolvimento de um sistema, destacam-se técnicas de desenvolvimento de *software* tais como Integração Contínua (IC) ou em inglês *Continuos Integration* (*CI*) (FOWLER, 2006).

> “Integração Contínua é uma prática de desenvolvimento de *software* onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por uma *build* automatizada (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times reportam que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva *software* coeso mais rapidamente.” (FOWLER, 2006).

Segundo Guerra (2014), a Integração Contínua é descrita como uma integração que possui *builds* automáticas que executam os testes, permitindo a detecção falhas em cada parte do sistema. Dentre os principais benefícios da Integração Contínua destacam-se a melhoria do *feedback* na detecção e solução

rápida dos erros, pois a cada *commit* realizado no repositório todos os testes são executados novamente(GUERRA,2014).

Além disso, a IC possibilita ter um sistema sempre atualizado e disponível para ser testado, fato que contribui para a diminuição dos riscos gerenciais e técnicos (FOWLER, 2006). Assim será possível ter conhecimento da quantidade de funcionalidades desenvolvidas, melhorando a capacidade de estimar o tempo para entrega de novas funcionalidades (GUERRA, 2014).

1.4 ESTRUTURA DO TRABALHO

Este trabalho está dividido em seis capítulos, sendo o primeiro destinado à descrição dos objetivos e a motivação para a realização desta pesquisa.

O segundo capítulo apresenta a fundamentação teórica, onde são encontradas informações e conceitos sobre metodologias de processos ágeis, Integração Contínua e testes de *software*.

O terceiro capítulo detalha o trabalho realizado no desenvolvimento de uma aplicação *web* e o funcionamento da parte de testes no trabalho.

O quarto capítulo aborda os resultados obtidos nos testes que foram desenvolvidos, além disso, trás uma descrição dos erros encontrados e uma discussão dos resultados obtidos.

O quinto capítulo apresenta a conclusão do trabalho, onde são discutidos de forma resumida os benefícios da Integração Contínua para o desenvolvimento.

Por fim, o sexto capítulo sugere possíveis indicações para trabalhos que poderão ser desenvolvidos futuramente.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda os conceitos relacionados a Metodologias Ágeis, Testes de *software* e Integração Contínua.

2.1 METODOLOGIAS DE DESENVOLVIMENTO DE *SOFTWARE*

Soares (2004) relata a diferença entre os métodos de desenvolvimento tradicional e ágil. A comparação enfatiza as dificuldades, tais como documentação, alteração de requisitos do projeto, definições de tempo e o custo total.

> "As metodologias tradicionais, conhecidas também como pesadas ou orientadas a planejamentos, devem ser aplicadas apenas em situações em que os requisitos do sistema são estáveis e requisitos futuros são previsíveis. Entretanto, em projetos em que há muitas mudanças, em que os requisitos são passíveis de alterações, onde refazer partes do código não é uma atividade que apresenta alto custo, as equipes são pequenas, as datas de entrega do *software* são curtas e o desenvolvimento rápido é fundamental, não pode haver requisitos estáticos, necessitando então de metodologias ágeis. Além disso, o ambiente das organizações é dinâmico, não permitindo então que os requisitos sejam estáticos." (SOARES, 2004).

De acordo com estudos realizados pelo The Standish Group (1995), a maioria dos sistemas desenvolvidos com a metodologia tradicional não alcançou a total satisfação do cliente e custou mais do que o planejado, sendo que o caso mais comum era o de atrasos na entrega do projeto. Soares (2004) fala sobre este problema do curto prazo para o desenvolvimento abordando a metodologia clássica, onde se prioriza a documentação. Ainda nesta metodologia, a mudança de requisitos também não era desejada, uma vez que o custo aumentava de forma exponencial a cada fase do projeto (SOARES, 2004, p.9).

A mudança de requisitos até hoje é um problema para quem não possui um planejamento satisfatório na empresa (SANTOS, 2008). Por mais que no início o modelo clássico ajudasse com a separação de tarefas e na melhor organização no desenvolvimento, quando o cliente desejava alterar um requisito muitas vezes já não

era possível, ou demandaria maior tempo para desenvolvê-lo (SOARES, 2004, p.12).

As metodologias ágeis surgiram como uma resposta a esses problemas. De acordo com Soares (2004), o termo “ágil” se deve ao fato da metodologia não ser orientada a documentação, diferentemente da tradicional. Isso não significa que a documentação não tenha importância, ela apenas aparece em plano secundário, ficando o enfoque primário nos indivíduos e interações, *software* executável, colaboração do cliente e resposta rápida para mudanças (SOARES, 2004, p.10).

Portanto, as metodologias ágeis visam a implementação, comunicação com o cliente, simplicidade e incrementação no código para cada ciclo/fase, facilitando a adição ou exclusão de requisitos no sistema (SOARES, 2004, p.10). Neste caso o tempo e o custo atuavam de forma equilibrada para o desenvolvimento das funcionalidades no projeto (SOARES, 2004, p.10).

> “Enquanto as metodologias ágeis variam em termos de práticas e ênfases, elas compartilham algumas características, como desenvolvimento iterativo e incremental, comunicação e redução de produtos intermediários, como documentação extensiva. Desta forma existem maiores possibilidades de atender aos requisitos do cliente, que muitas vezes são mutáveis.” (SOARES, 2004).

Um exemplo de metodologia ágil é a *XP* (ou *Extreme Programming*), comumente utilizada em pequenas e médias empresas (BECK, 1999 apud SOARES, 2004). Essa metodologia tem como características relevantes a abordagem incremental do *software*, a frequente interação com o cliente e o *feedback* constante das atividades que estão sendo realizadas (KOSCIANSKI; SOARES, 2007). Entre as práticas utilizadas por esta metodologia destacam-se o teste e a Integração Contínua, os quais permitem “a validação do projeto durante todo o processo de desenvolvimento” (SOARES, 2004).

De acordo com Beck (1999 apud SOARES, 2004), a *Extreme Programming* se baseia em doze práticas relacionadas abaixo:

1. Planejamento: baseado em requisitos atuais. As tarefas são divididas para cada equipe, e controladas através de um cronograma de execução.

2. Entregas Frequentes: a cada mês ou no máximo a cada dois meses, uma parte do *software* é entregue ao cliente. Desta forma o *feedback* do cliente é mais rápido e aumenta as possibilidades da sua satisfação na entrega final.

3. Metáfora: os termos técnicos são substituídos por termos mais simples para a compreensão da maioria dos envolvidos.

4. Projeto Simples: o projeto deve ser pequeno e de fácil entendimento por todos, implementando os requisitos atuais, e projetando os que possam aparecer.

5. Testes: são feitos em todo o processo de desenvolvimento.

6. Refatoração: simplificação de funcionalidade já existente.

7. Programação em Pares: a programação é realizada em dupla, onde é feito o revezamento de programação e verificação do código.

8. Propriedade Coletiva: o projeto é de conhecimento de todos, não há separações no desenvolvimento. Todos os integrantes conhecem o código e essa é uma vantagem, pois caso alguém deixe a equipe, o desenvolvimento continuará sem muitas dificuldades.

9. Integração Contínua: a interação e a construção do sistema ocorrem várias vezes, possibilitando processos rápidos. Deve-se incluir apenas um conjunto de alterações de cada vez em uma máquina dedicada às aplicações de integração. No caso dos testes acusarem falhas, os mesmos deverão ser corrigidos com urgência pela equipe que agregou as últimas linhas de código.

10. Quarenta horas de trabalho semanal: essa metodologia recomenda que não sejam feitas horas extras constantemente.

11. Cliente Presente: o cliente participa de todo o do processo de desenvolvimento e “deve estar sempre disponível para sanar todas as dúvidas de requisitos, evitando atrasos” (SOARES, 2004, p.5).

12. Código Padrão: padronização na organização do código, assim como na nomenclatura de variáveis, para que seja um código compreendido por todos.

Outra metodologia ágil existente é a *Scrum* (SCHWABER; BEEDLE, 2002). Ela utiliza ideias referentes ao controle de etapas industriais, e compreende que o desenvolvimento possui variáveis técnicas e ambientes que podem sofrer mudança no decorrer do projeto, sendo necessário, portanto, utilizar práticas que permitam a flexibilidade (SOARES, 2004, p.5). Soares (2004) corrobora esta ideia afirmando que

o “foco da metodologia é encontrar uma forma de trabalho dos membros da equipe para produzir o *software* de forma flexível e em um ambiente em constante mudança” (SOARES, 2004, p. 5).

Os princípios da metodologia *Scrum* são parecidos com os da metodologia *XP*. Existem similaridades como o foco em grupos de trabalho pequenos, requisitos pouco estáveis, e iterações curtas (SOARES, 2004, p.5).

Cada iteração, ou fase, é chamada de *sprint,* e, segundo Soares (2004, p. 5) possui duração de trinta dias. São realizadas reuniões diárias e rápidas de quinze minutos para que a equipe mostre o trabalho feito e os avanços no desenvolvimento (SOARES, 2004, p.5). Nessas reuniões são definidas as metas e pendências para o dia seguinte. As três principais fases da metodologia *Scrum* são (SOARES, 2004, p.5):

1. Pré-planejamento: os requisitos do sistema são elaborados; as estimativas de prazo são feitas, bem como a definição da equipe e das ferramentas, riscos e treinamentos.

2. Desenvolvimento: as variáveis técnicas e do ambiente para a análise dos riscos e acompanhamento de mudanças são controladas para melhor flexibilidade. É nessa fase que ocorrem os *sprints*, cada um contendo análise, projeto, desenvolvimento e teste.

3. Pós-Planejamento: são realizadas reuniões para efetuar a entrega de cada funcionalidade terminada em um *sprint* para o cliente. Nesta fase acontecem os testes e integrações.

As ferramentas que suportam as metodologias ágeis já existem, e efetuam o controle das métricas de cada equipe, de cada desenvolvedor, executam a divisão das tarefas e a integração e testes (SOARES, 2004, p.6).

Soares (2004, p. 13) comenta os resultados do uso dessas metodologias para o desenvolvimento de *software* e argumenta que a satisfação dos clientes tem aumentado. As empresas também se adaptaram muito bem, e reduziram seus custos (SANTOS, 2008). Porém, um quesito que deixa a desejar é referente aos diagramas e a documentação, que normalmente são deixados em segundo plano (SOARES, 2004, p.13).

2.2 INTEGRAÇÃO CONTÍNUA – CONCEITO

O conceito de Integração Contínua pode ser definido através da citação de Fowler (2006) abaixo:

> "Integração Contínua é uma prática de desenvolvimento de *software* onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por uma *build* automatizada (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva *software* coeso mais rapidamente." (FOWLER, 2006).

Rocha (2014) comenta os benefícios que a Integração Contínua traz para o processo de desenvolvimento de *software*, sendo um método muito eficaz para o ganho de tempo na união de funcionalidades. Para a sua utilização, é desejável que a empresa implante um sistema de controle de versão centralizado, como por exemplo, *CVS, Subversion e Git* (ROCHA, 2014). O objetivo desse controle é facilitar o trabalho em equipe, que e é essencial para metodologias ágeis. (FOWLER, 2006).

Também são importantes os testes automatizados para que seja possível a análise de erros (FOWLER, 2006). De acordo com Fowler (2006), para uma boa integração, além dos testes e do controle de versão, a automatização das *builds* é um ponto crucial. As ferramentas *Ant* (Java), *Maven*, *Nant* (*.net*), e *MSBuild* (*.net*) , por exemplo, realizam essa automatização (ROCHA, 2014) .

A partir do momento em que se tem um controle de versão centralizado, testes e *builds* automatizadas, podemos então realizar a integração, a qual irá unir todos estes itens para trabalhar em conjunto (ROCHA, 2014). A citação abaixo comenta de forma breve como as alterações do código são feitas a partir deste ambiente integrado.

> "O único pré-requisito para um desenvolvedor lançar suas alterações na versão principal é que ele consiga executar perfeitamente o código. Isso, claro, inclui passar pelos testes da *build*. Como com qualquer ciclo de lançamento de código, o desenvolvedor primeiro atualiza sua cópia de trabalho para coincidir com a versão principal, resolve qualquer conflito e então gera a *build* em sua máquina local. Se a *build* passar, então ele estará liberado para lançar suas alterações na versão principal." (FOWLER, 2006).

Existem várias ferramentas que realizam a Integração Contínua, tais como *Hudson* e *Jenkins* (ROCHA, 2014). Essas ferramentas permitem que sejam configurados o sistema, o ambiente de desenvolvimento, o sistema de *build* automatizado, a integração com o repositório de controle de versão e o envio de e-mails de notificação quando mudanças são realizadas no código (ROCHA, 2014).

Os benefícios da Integração Contínua num ambiente de desenvolvimento são inúmeros (FOWLER, 2006). Podemos citar a redução de riscos no projeto, a rápida detecção e solução de erros, a facilidade e praticidade para os trabalhos em equipe (ROCHA, 2014).

2.3 TESTES

Teste de *software* é a execução de um conjunto de verificações que determinam se o sistema desenvolvido “atingiu suas especificações e funcionou corretamente no ambiente para o qual foi projetado” (NETO, 2014).

Testes de caixa branca e testes de caixa preta são as técnicas mais difundidas de teste de *software* (SCOTT, 2012).

Segundo Pressman (2005 apud NETO, 2014), a técnica de caixa branca avalia o comportamento e os aspectos internos da aplicação, tais como o fluxo de dados, iterações, e os caminhos lógicos implementados, focando diretamente sobre o código fonte.

A técnica de caixa preta não necessita do acesso ao código fonte, pois são verificadas as funcionalidades da aplicação, auxiliando na identificação de falhas na interface e nos comandos realizados pelo usuário, bem como erros de comportamento ou desempenho (SCOTT, 2012).

De acordo com Rocha (2001 apud NETO, 2014), existem vários níveis de teste, dentre eles estão:

- Teste de Unidade: também pode ser chamado de teste unitário. Seu principal foco é avaliar cada pequena parte do código separadamente, procurando por falhas na lógica desenvolvida.

- Teste de Integração: tem por objetivo verificar falhas atreladas nas “interfaces entre os módulos quando esses são integrados para construir a estrutura do *software* que foi estabelecida na fase de projeto” (NETO, 2014).
- Teste de Sistema: realiza a utilização do *software* no nível de um usuário final para buscar falhas durante a interação e problemas referentes ao ambiente em que o *software* está sendo executado. Neste teste os requisitos também são verificados e validados com o cliente.
- Teste de Aceitação: feito por um pequeno grupo de usuários finais que “simulam operações de rotina do sistema de modo a verificar se o seu comportamento está de acordo com o solicitado” (NETO, 2014).

O teste de Regressão também é um método existente para minimizar as falhas no *software*, entretanto, de acordo com Rocha (2001 apud NETO, 2014), não pode ser considerado como um nível de teste por ser complementar a qualquer fase. Este teste possui o objetivo de executar “a cada nova versão do *software* ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema” (NETO, 2014).

Rocha (2001 apud NETO, 2014) menciona que existem vários níveis de teste, os quais devem ser planejados de acordo com o andamento do desenvolvimento da aplicação. Na figura 1 é demonstrado este procedimento.

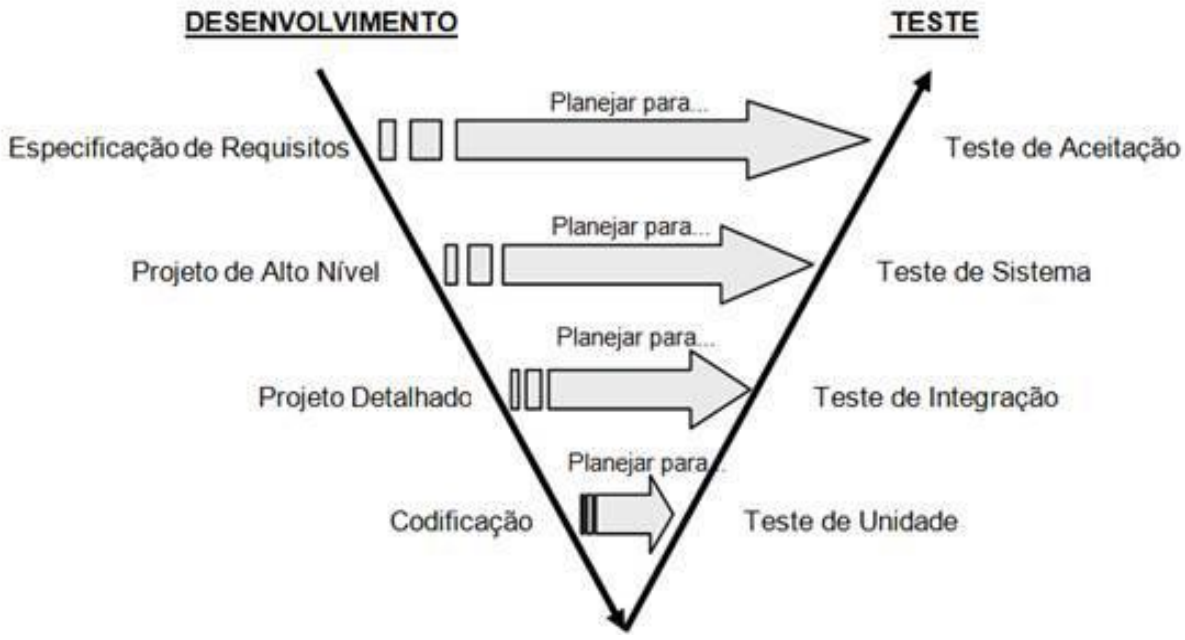


Figura 1 - Níveis de testes no decorrer do projeto.
Fonte: Craig e Jaskiel (2002)

De acordo com a figura 1, Neto (2014) explica que o planejamento dos testes segue a sequência da seta apontando para baixo, ou seja, para a elaboração dos testes de aceitação é utilizado o documento de especificação dos requisitos; para a elaboração dos testes de sistema é utilizado o projeto de alto nível; e assim por diante.

Após a elaboração dos testes, a execução é realizada seguindo a sequência da seta apontando para cima, sendo testada primeiramente a codificação através dos testes de unidade, e por fim, os requisitos através dos testes de aceitação (NETO, 2014).

Logo, os testes que são desenvolvidos em uma aplicação contribuem para que o sistema desempenhe suas funções com o mínimo de falhas possível, aumentando a qualidade, confiabilidade e a integridade do *software* para o usuário final (PRESSMAN, 2011).

2.3.1 Testes implementados com *JUnit*

A implementação dos testes neste trabalho foi baseada nas técnicas de testes de aceitação (caixa preta) (ABREU, 2009) – seção 3.5.

O teste de aceitação é feito junto ao cliente, sendo o último nível de teste antes do *software* ser implantado (MEDEIROS, 2014). Os objetivos desse teste incluem a verificação se o sistema está pronto para ser utilizado, e se os requisitos foram desenvolvidos corretamente (PRESMANN, 2011).

Outra prática de teste muito conhecida é a de testes unitários. Dentre seus benefícios podemos mencionar a prevenção de *bugs,* confiabilidade no código, realização de métricas do projeto e conhecimento sobre a regra de negócio utilizada (MEDEIROS, 2014).

Existem vários *frameworks* que permitem facilitar a elaboração dos testes, como por exemplo, o *JUnit* (MEDEIROS, 2014).

O *JUnit* pode ser utilizado tendo como referência os passos descritos por Medeiros (2014), adaptado para este trabalho:

- Definir o teste a ser realizado;

- Elaborar uma classe de teste (*test case* ou *JUnit class*) com seus métodos;
- Programar o código mais simples que rode o teste;
- Rodar o *JUnit;*
- Analisar o resultado.

O *JUnit* mostra como resultado de teste três situações: sucesso, falha e erro (MEDEIROS, 2014). Quando o resultado é sucesso, indica que seu teste foi rodado de forma a não encontrar nenhum erro, entretanto, se o resultado acusar erro, o mesmo é demonstrado juntamente com o valor obtido no teste e o valor esperado (MEDEIROS, 2014). Por fim, se o resultado for falha significa “que o comportamento operacional do *software* foi diferente do esperado pelo usuário” (NETO, 2014).

Em relação a cada *JUnit class* – classe de teste - existem três tipos de anotações: a *@test* que indica que é um teste do *JUnit*, a *@before* que executa o método com a anotação antes de rodar os testes e a *@After* que executa o método com a anotação depois de rodar os testes (MOREIRA, 2014). Podemos perceber cada uma destas anotações na figura 2.

```
import com.thoughtworks.selenium.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.regex.Pattern;

public class CadastroUsuarioSemCampos {
        private Selenium selenium;

      @Before
        public void setUp() throws Exception {
                selenium = new DefaultSelenium("localhost", 4444, "*chrome", "http://localhost/carro_7/estacionamento.php");
                selenium.start();
        }

      @Test
        public void testCadastroUsuarioSemCampos() throws Exception {
                selenium.open("/carro_7/estacionamento.php");
                selenium.click("id=clickcadastro");
                selenium.click("id=enviar");
                assertEquals("Por favor, verifique os campos novamente", selenium.getAlert());
        }

      @After
        public void tearDown() throws Exception {
                selenium.stop();
        }
}
```

Figura 2 - Classe do *JUnit* - Exemplo
Fonte: Autoria Própria

Neste trabalho as classes *JUnit* foram geradas automaticamente através da ferramenta *Selenium*, e executadas através do servidor *Selenium RC* pelo *prompt* de comandos.

2.4 TRABALHOS RELACIONADOS

Nesta seção serão apresentados trabalhos onde foram utilizados ferramentas e padrões das metodologias ágeis *Extreme Programming* e *Scrum*. A aplicação destes métodos visa uma produtividade rápida e com qualidade para o desenvolvimento de *software*.

2.4.1 Padrões e Métodos Ágeis

O texto de Garcia e Penteado (2005) apresenta padrões organizacionais e de processo, os quais complementam a utilização dos métodos ágeis visando melhorias e agilidade na elaboração dos códigos de um sistema. As metodologias ágeis *Extreme Programming* (*XP*) e *Scrum* foram escolhidos por estes autores para exemplificar os padrões destacados.

> "Os padrões organizacionais e de processo podem ser destacados pela relação que possuem com os métodos ágeis. Esses padrões melhoram o processo de desenvolvimento de *software*, acelerando seu desenvolvimento. Alguns desses padrões já são usados como práticas nos métodos ágeis e outros podem ser integrados para apoiar o desenvolvimento de *software*." (GARCIA; PENTEADO, 2005).

Estes padrões são utilizados como práticas nos ambientes empresariais, e contribuem inclusive para a organização e administração do projeto (GARCIA; PENTEADO, 2005).

No Quadro 1 foram destacados os padrões mais comuns:

Nome	Resumo
Size the Organization	Refere-se ao tamanho da equipe de desenvolvimento. Quando a equipe possui muitos indivíduos, há a dificuldade da entrega do projeto no prazo e orçamento previsto, além da comunicação ser um item agravante. Se a

	equipe é pequena, a produtividade vai diminuir. O recomendado é escolher aproximadamente dez pessoas para compor a equipe de desenvolvimento e evitar acrescentar pessoas depois de iniciado o projeto.
Developing in Pairs	Melhora a desenvoltura individual dos programadores quando são organizados em pares para realizar o desenvolvimento.
Community of Trust	O relacionamento social tem um impacto significante na efetividade da equipe. Este padrão recomenda que todos tenham uma boa comunicação e confiança no trabalho do colega.
Unity of Purpose	Frequentemente, as pessoas têm ideias diferentes de como o produto final deveria ser. Assim, deve-se contratar um líder do projeto que coloca para todos os membros da equipe uma visão comum e propósito geral do *software*.

Quadro 1 - Padrões Organizacionais e de Processo e Métodos Ágeis
Fonte: Garcia e Penteado (2005)

Dentre estes padrões, o benefício do desenvolvimento em pares pode ser destacado através de estudos realizados por Cockburn (2002), onde este afirma que a programação em dupla é mais rápida devido ao companheirismo entre desenvolvedores referente às dúvidas que surgem no decorrer do projeto. “Eles aprendem mais sobre o projeto e o sistema, o código desenvolvido é menor e o *software* é produzido com menos defeito” (GARCIA; PENTEADO, 2005).

Os outros padrões relacionados se referem à união da equipe, o bom ambiente de trabalho, gratificações para as conquistas realizadas durante o projeto e separação de tarefas (GARCIA; PENTEADO, 2005).

Desta forma, estas práticas sugerem dicas para que se evitem atrasos no desenvolvimento, as quais também são válidas para ajudar a manter um bom relacionamento entre os membros da equipe (GARCIA; PENTEADO, 2005).

2.4.2 Desenvolvimento Orientado a Testes com Integração Contínua

Bowyer e Hughes (2006) descrevem um estudo onde alunos de engenharia de *software* realizam o desenvolvimento orientado a testes com Integração Contínua num ambiente muito similar aos utilizados pelas empresas.

A metodologia escolhida para a implementação foi a *Extreme Programming (XP)* com a prática do padrão *Developing in Pairs,* onde o desenvolvimento é feito a cada dois programadores na mesma máquina (BOWYER; HUGHES, 2006).

Os alunos programaram em Java, cuja linguagem já lhes era familiar, entretanto, para cada par se deu o tempo de apenas 20 horas semanais para

realizar o projeto do *software*, o qual se baseava numa biblioteca de DVDs e um sistema de *e-commerce*, o mais simples possível (BOWYER; HUGHES, 2006).

Em tal sistema, os desenvolvedores mantêm seu código e seus testes de unidade em um servidor de controle de versão, o qual é monitorado constantemente por outro servidor de Integração Contínua (BOWYER; HUGHES, 2006).

Quando uma modificação é realizada, o servidor executa um *script* de construção para aquela versão do projeto (BOWYER; HUGHES, 2006). Bowyer e Hughes (2006) relatam que na maioria das vezes este *script* busca as últimas versões de todo o código e as classes de teste, compila o código e os testes, e em seguida executa os testes. Se um código não compilar ou um teste falhar, a construção acusa erro, caso contrário, informa sucesso na execução (BOWYER; HUGHES, 2006). Este resultado é então publicado aos desenvolvedores, normalmente enviado por e-mail ou através de páginas intranet (BOWYER; HUGHES, 2006).

De acordo com o trabalho de Bowyer e Hughes (2006), os servidores de Integração Contínua mais utilizados são o *CruiseControl* e o *Anthill*. Ambos aplicam a ferramenta baseada em *Java Apache Ant* para construir *scripts*, e suportam uma variedade de servidores de controle de versão (BOWYER; HUGHES, 2006). O *CruiseControl* foi usado nesse caso, juntamente com o *Visual SourceSafe* da Microsoft (BOWYER; HUGHES, 2006).

Para cada par de estudantes, foi realizada a criação de um projeto no *CruiseControl*, um repositório *SourceSafe*, um arquivo de construção *Ant*, e a ligação entre todos estes itens (BOWYER; HUGHES, 2006).

O projeto do *CruiseControl* realizava verificações no repositório *SourceSafe* de cada par procurando por mudanças, e caso elas fossem detectadas, o *CruiseControl* invocava o arquivo de construção *Ant* correspondente (BOWYER; HUGHES, 2006).

O arquivo *Ant* possuía a função de recuperar a última versão de todos os arquivos do repositório *SourceSafe* de cada par e guardar em um diretório temporário (BOWYER; HUGHES, 2006). Logo após, iniciava o compilador Java e executava o *JUnit* para rodar os testes de unidade (BOWYER; HUGHES, 2006).

Realizado estes procedimentos, o *CruiseControl* prosseguia com o encaminhamento de e-mail das *builds* para os estudantes e uma cópia para os tutores (BOWYER; HUGHES, 2006). O conteúdo deste e-mail baseava-se na

identificação do par de desenvolvedores, ID da construção, data e hora da última mudança, resultado dos testes de unidade e um resumo das modificações feitas desde a última versão (BOWYER; HUGHES, 2006).

Os resultados também eram automaticamente postados em um servidor *CruiseControl web* rodando *Tomcat web Server* (BOWYER; HUGHES, 2006). Os estudantes adquiriram uma visão geral do sistema e de suas funcionalidades, e criaram seus testes de unidade e código (BOWYER; HUGHES, 2006).

Os resultados finais avaliaram o desempenho dos estudantes e a qualidade referente às falhas do *software*, com medidas quantitativas geradas pelo *CruiseControl* (BOWYER; HUGHES, 2006). A abordagem, segundo os autores Bowyer e Hughes (2006), alcançou os objetivos, pois mostrou aos participantes as vantagens do uso de uma ferramenta gerencial no processo de desenvolvimento de *software* com Integração Contínua.

2.4.3 Integração Contínua - Uma Abordagem Prática

Fowler (2006) explica como funciona a Integração Contínua em um ambiente de desenvolvimento. Abaixo foram relacionados os tópicos relevantes desta explicação:

- Uma cópia da versão atual do projeto é criada na máquina do desenvolvedor;
- Cria-se uma nova funcionalidade neste código;
- São gerados e executados os testes e as *builds* (versões do *software*).
- Caso não acuse nenhum erro, é criada uma *build* no servidor de integração com essas modificações, e então testada novamente.
- Se a construção for finalizada sem erros, então o desenvolvedor poderá realizar o *commit* (salvar as mudanças no repositório do projeto) no servidor.
- Se houver qualquer erro, o desenvolvedor deverá consertar as *builds* e testá-las novamente.

Fowler (2006) comenta sobre as práticas da Integração, que deixa o processo mais organizado e evita falhas administrativas.

De forma resumida, o quadro 2 apresenta estas práticas citadas:

Prática	Resumo
Manter um único repositório de código	Os arquivos que compõem o projeto estão todos em um local comum e acessível por todos os desenvolvedores, inclusive os arquivos para a geração das *builds* como *scripts* de teste e arquivos de configuração.
Automatizar a *build*	Visa evitar falhas, e principalmente ganho de tempo. A principal ideia é gerar e lançar o sistema rodando os *scripts* em um único comando. Os *scripts* normalmente tem uma plataforma, por exemplo, os projetos em *Java* utilizam o *Ant*.
A *build* deve ser auto-testável	Uma *build* é a compilação de todo o material necessário para ter um programa executando. Incluir testes automáticos no processo de *build* faz com que o índice de falhas diminua. As ferramentas *xUnit* é ponto inicial para tornar seu código auto testável.
Modificações Diárias	Para que uma mudança seja efetivada, o desenvolvedor deve rodar as modificações no projeto que está no repositório, desta maneira qualquer incompatibilidade de versões deverá ser resolvida antes do *commit* ser realizado.
Cada *commit* deve atualizar o repositório principal em uma máquina de integração	No ambiente de trabalho deve-se assegurar que as *builds* ocorram numa máquina de integração, e somente se a *build* de integração passar com sucesso é que o *commit* deve ser considerado. Isso evita que erros aconteçam por falhas nas versões que os desenvolvedores utilizam para fazer as modificações.
Manter a *build* rápida	"Na maior parte dos projetos, a linha de *build* do *XP* de 10 minutos é (...) perfeita. Uma vez que a IC demanda *commits* frequentes, ela economiza muito tempo." (FOWLER, M., 2006).
Testar o *software* no ambiente que ele será utilizado pelo cliente	Os testes devem ser feitos no ambiente onde o *software* irá rodar para o cliente. Caso o teste ocorra de maneira diferente, o projeto terá o risco de não executar da maneira perfeita após sua entrega.
Tornar fácil o acesso ao último executável.	Geralmente o resultado final das *builds* da iteração é salvo junto ao arquivo executável. Este arquivo deve ser de fácil acesso e o local deve ser conhecido por todos.
Visibilidade do andamento das *builds*	Integração Contínua está estritamente ligada à comunicação, ou seja, num ambiente de desenvolvimento deve haver a transparência do andamento do trabalho de cada um. Os gerenciadores permitem a visibilidade do andamento de uma *build*, se o *commit* está sendo realizado, se houve falha ou sucesso na construção, quais as modificações foram feitas e quem as realizou, a hora e a data, etc. Também é possível ter esse controle por acesso ao site do gerenciador.
Automatizar a Implantação do Sistema	Realização de testes no ambiente do cliente com um *software* que não está completo, permitindo ter um *feedback* da parte que já está pronta. Desta maneira, caso algo ocorra de maneira imprevista, o projeto ainda não estará finalizado, sendo mais fácil realizar as alterações.

Quadro 2 - Práticas da Integração Contínua
Fonte: Adaptado de Fowler (2006)

Fowler (2006) fala dos benefícios destas práticas, onde o principal é a eliminação de riscos referente aos *bugs* no projeto. Com o desenvolvimento mais rápido, o cliente pode dar um dar um *feedback* para os desenvolvedores conforme as funcionalidades ficam prontas (FOWLER, 2006).

A implantação da Integração Contínua é aconselhável para todos os ambientes, e caso a empresa esteja iniciando este processo, o primeiro passo é ter uma *build* automatizada (FOWLER, 2006).

2.5 GERENCIADORES, LINGUAGENS E OUTRAS FERRAMENTAS UTILIZADAS PARA O DESENVOLVIMENTO DO TRABALHO

Nesta seção serão apresentadas as ferramentas utilizadas para o desenvolvimento do trabalho.

2.5.1 *Selenium*

O *Selenium* é uma ferramenta *open source* que engloba várias funcionalidades utilizadas para a elaboração de testes de *software* automatizados (GONÇALVES, 2011).

Uma das vantagens observadas é “a possibilidade de executar os testes em qualquer navegador com suporte para *JavaScript*” (PEREIRA, 2012, p. 15).

Um componente importante é o *Selenium Integrated Development Environment* (*IDE*), o qual consiste num “ambiente de desenvolvimento integrado para os testes” (MEIRELES; RODRIGUES, 2013).

Segundo Meireles (2013), este componente opera como um *plugin* do navegador Firefox, provendo ao programador uma interface simples, elaborada para facilitar o desenvolvimento e a execução dos testes.

O *Selenium IDE* facilita a automatização dos testes por ser uma ferramenta *record-and-playback*, a qual realiza o armazenamento das “ações executadas pelo testador e gera um script que permite a re-execução das ações feitas” (MEIRELES; RODRIGUES, 2013).

Na figura 3 é demonstrada a interface desta ferramenta.

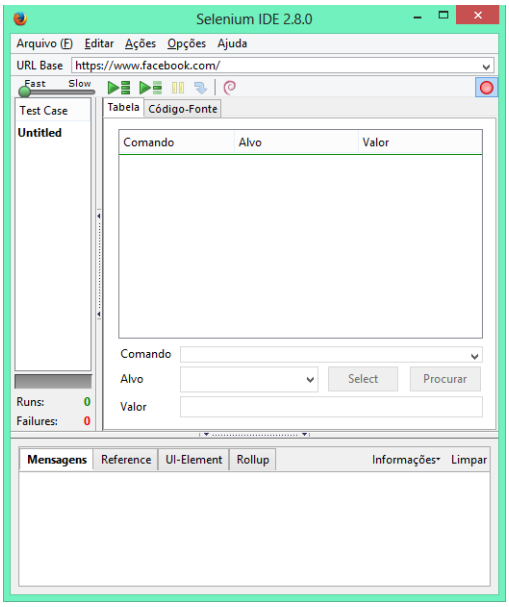


Figura 3 - Interface *Selenium IDE*
Fonte: Autoria Própria

2.5.2 *GitHub*

É um serviço que permite armazenar códigos através dos repositórios criados no próprio ambiente, com funcionalidades semelhantes a uma rede social em que os usuários podem efetuar mudanças nos projetos que estão compartilhados (GITHUB, 2014).

Esta ferramenta permite a gerência das configurações e a visualização de gráficos que informam como os desenvolvedores trabalham nos seus repositórios (GITHUB, 2014).

A figura 4 demonstra a página de visualização dos repositórios de um usuário no *GitHub*.

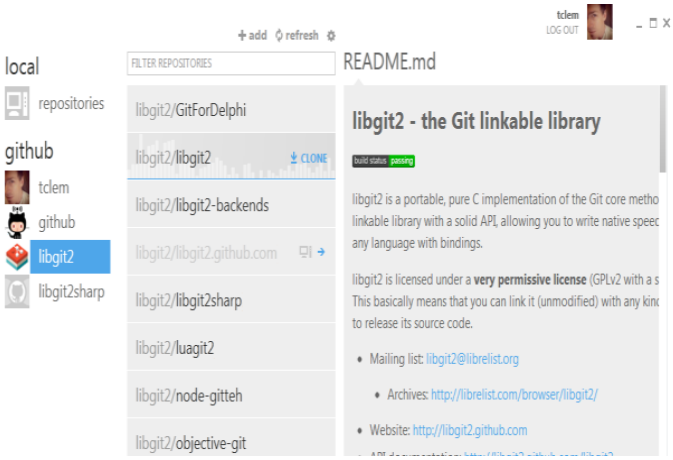


Figura 4 - Interface *GitHub*
Fonte: *GitHub* (2014)

2.5.3 *Jenkins*

É uma ferramenta de Integração Contínua *open source* que gerencia a construção de testes automatizados de *software* (JENKINS, 2014). O aplicativo monitora a execução de trabalhos repetitivos, como de criação de *buildings* de projetos de *software* ou a execução de rotinas do *cron* (SOUZA, 2014). A figura 5 ilustra a página inicial da ferramenta.

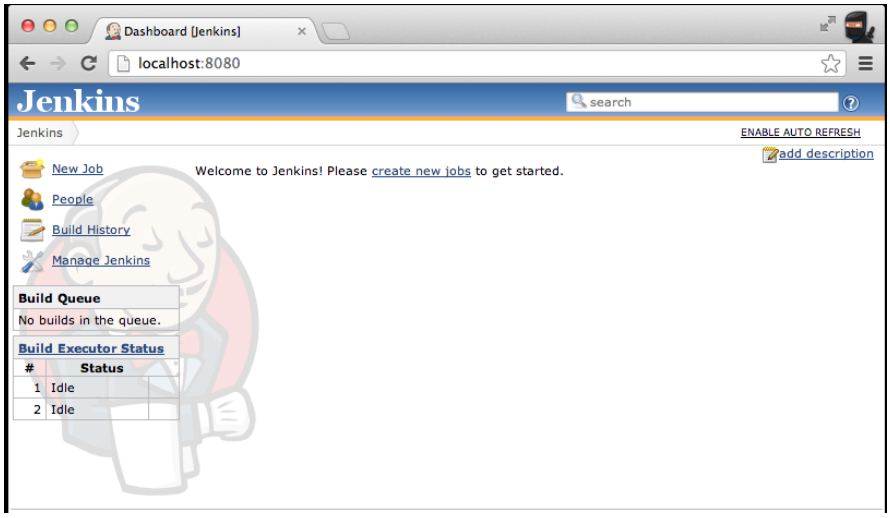


Figura 5 - Página Inicial – *Jenkins*
Fonte: Antonini (2014)

2.5.4 *Ant*

É uma biblioteca feita para execuções em linha de comando elaborada na linguagem Java, utilizada para automatizar comandos descritos em arquivos chamados de *builds*, os quais são versões compiladas de um *software* (APACHE ANT, 2014).

No contexto deste trabalho esta biblioteca foi utilizada para automatizar a construção dos testes com base em arquivos no formato *XML*, os quais descrevem o processo de construção e suas dependências. “Por padrão este arquivo *XML* tem o nome de *build.xml*”(MOREIRA,2014).

Essa ferramenta pode ser utilizada para uma série de tarefas internas que permitem compilar, iniciar, testar e executar aplicativos na linguagem Java ou em outras linguagens como C ou C ++ (APACHE ANT, 2014).

2.5.5 *JUnit*

O *JUnit* é um exemplo de *framework* de código aberto que possibilita a interpretação de testes criados utilizando a linguagem Java (MEDEIROS, 2014). Entre seus principais objetivos destacam-se a verificação de cada unidade de código, facilidade de criação de testes, execução automática de testes e a demonstração dos resultados (MEDEIROS, 2014).

Para a instalação do *JUnit* é necessário realizar o *download* na página <*www.junit.org*> e copiar o arquivo com extensão “.jar” “para o CLASSPATH, caminho que o seu compilador Java procura pelas classes” (SANTOS, 2011). De acordo com Medeiros (2014), este *framework* já está incluso e configurado em várias ferramentas de desenvolvimento, tais como *Netbeans* e *Eclipse*, podendo ser utilizado diretamente nestas interfaces de desenvolvimento, ou através de comandos.

2.5.6 *HTML*

HyperText Markup Language (*HTML*) é uma linguagem de marcação utilizada

na construção e criação de *homepages*. Essa linguagem possibilita organizar as informações como *links*, os quais realizam a interligação entre várias páginas de diferentes conteúdos (ALVES, 2004). A *HTML* é derivada da *Standard Generalized Markup Language* (*SGML*), uma linguagem precursora de marcação, criada com o intuito de compor e apresentar documentos na *Web* (GUIMARÃES, 2005).

Segundo Eis (2011), os códigos que estruturam a linguagem *HTML* são conhecidos como *tags,* e têm o papel de organizar e definir os elementos contidos numa página. As demarcações de títulos, cabeçalhos, parágrafos e rodapés são exemplos das *tags* existentes. Portanto, é através destas marcações que o navegador interpreta o que representa cada fragmento de texto (EIS, 2011).

De acordo com Alvarez (2004), atualmente a *Web* possui um caráter multimídia, não envolvendo apenas elementos textuais. Alvarez (2004) menciona que para que se tornasse possível a utilização de vídeos, imagens e áudios, a linguagem *HTML* sofreu alterações. Estas mudanças se tornaram padrões, onde a cada versão são acopladas melhorias no código (ALVAREZ, 2004).

2.5.7 *CSS*

O *Cascading Style Sheets* (*CSS*) representa folhas de estilos em cascata (W3CBRASIL, 2014). Ele é utilizado em conjunto com determinadas linguagens de marcação como *HTML*, *XHTML* e *XML* para aprimorar a apresentação de páginas *Web* (PEREIRA, 2009).

A tecnologia *CSS* formata qualquer informação proveniente de um documento *Web*, incluindo textos, imagens, vídeos, áudios, entre outros (W3CBRASIL, 2014).

Seu funcionamento é simples, pois trata-se de um arquivo externo que contém todas as regras aplicadas, detalhes e o código que juntos definem como será o estilo da página (HTMLPROGRESSIVO, 2014). A outra vantagem é de que caso exista a necessidade de uma alteração no estilo, a mesma pode ser realizada de maneira simples e ágil (HTMLPROGRESSIVO, 2014).

Pereira (2009) afirma que o *CSS* fica a cargo de aplicar esses estilos para a apresentação da página, enquanto as linguagens de marcação são encarregadas de desempenhar suas funções básicas de marcação e estruturação de conteúdo. O

CSS também possibilita que *tags* semelhantes de uma página sejam apresentadas em estilos diferentes, o que pode ser observado através de seus menus em cascata e dos estilos de cabeçalho e rodapé (PEREIRA, 2009).

2.5.8 *PHP*

O *Hypertext Preprocessor* (*PHP*) “é uma linguagem de programação de código aberto utilizada para o desenvolvimento *Web*” (PHP, 2014). Em resumo, uma página criada com a tecnologia *PHP* nada mais é do que um documento gerado com a extensão “.php” que contém códigos que são executados em um determinado servidor *Web* (SILVA, 2014).

Uma das vantagens do *PHP* está no fato de que ele possibilita introduzir fragmentos de código na página *HTML* e permite a realização de tarefas de forma simples e eficaz (ALVAREZ, 2004). É importante ressaltar que a interpretação do *PHP* não ocorre localmente na máquina do usuário, ela é efetuada através de um servidor *Web* (PHP, 2014).

De acordo com o texto de Milani (2010), grande parte do mercado de servidores de hospedagem e de uso em domínios de internet utilizam *PHP*, e esses números vem crescendo devido à quantidade de desenvolvedores autônomos e empresas voltadas para *Web* e *e-commerce* que utilizam essa linguagem para suas aplicações.

2.5.9 *JavaScript*

JavaScript trata-se de uma linguagem de programação que se baseia em *scripts* e tem a finalidade de adicionar interatividade em uma página *Web* (KIOSKEA, 2014). Tal linguagem gera melhorias para linguagens de marcação como a *HTML*, e permite a execução de comandos provenientes do navegador local e não somente do servidor *Web* (KIOSKEA, 2014).

Existe um vasto número de linguagens de programação como *PHP*, *ASP*, Java, *Python*, *Ruby* entre outras, que tem a função de adicionar e realizar o processamento de dados em páginas *Web* (SILVA, 2010). Segundo Alvarez (2004),

com a utilização de *JavaScript* o tráfego na rede é melhorado por não efetuar tantas requisições ao servidor como as linguagens citadas acima.

“Dentre as várias funcionalidades do *JavaScript*, pode-se citar a manipulação do conteúdo e apresentação” (MENEZES, 2013). Silva (2010) explica em seu texto que através dessa linguagem pode-se, por exemplo, escrever um pedaço de código *HTML* e posteriormente inseri-lo em outro arquivo já existente.

Também é possível definir, controlar e alterar de uma forma dinâmica a apresentação de qualquer documento *HTML*, através da manipulação direta da folha de estilo relacionada ao documento, criando ou invalidando regras *CSS* (SILVA, 2010).

De acordo com Silva (2010), o *JavaScript* é capaz de interagir com formulários para efetuar validações, bem como interagir com um conjunto de outras linguagens com o intuito de realizar tarefas que lhes são complementares.

2.5.10 *jQuery*

jQuery é uma biblioteca *JavaScript* que tem por característica rapidez e leveza, além de ser rica em recursos, sendo normalmente utilizada para desenvolvimento de *scripts* que interagem com páginas *HTML* (CODIGO FONTE, 2014). Através desta biblioteca pode-se trabalhar com alteração ou criação de elementos, definir efeitos, dentre várias outras funções que possibilitam uma combinação de versatilidade e extensibilidade (CODIGO FONTE, 2014).

Segundo Silva (2014), a biblioteca *jQuery* possui um conjunto de arquivos em *JavaScript* – armazenados com a extensão “.js” – que traz facilidade na sintaxe e implementação do código do *software.* Devido a sua simplicidade, essa biblioteca permite que um número elevado de linhas de programação *JavaScript*, possa ser substituído por um número muito inferior, tornando o desenvolvimento compacto e simplificado (SILVA, 2009, p. 26).

Com a utilização do *jQuery* as páginas da *Web* ganham dinamismo e interatividade, características que aprimoram o *design* e trazem benefícios para o usuário através da acessibilidade e da usabilidade (SILVA, 2014). A biblioteca é aplicada para acrescentar efeitos e animações na página, acessar e manipular o

DOM (Modelo de Objetos de Documentos), permitir a busca por informações nos servidores sem a necessidade de recarregar a página em uso, e possibilitar a alteração de conteúdos, modificando a apresentação e estilização da página. (SILVA, 2009, p. 27).

De acordo com Silva (2009, p. 28), na criação da biblioteca houve a preocupação de que a mesma estivesse de acordo com os padrões *Web*, para que a compatibilidade com os sistemas operacionais e navegadores não fosse afetada. Desta forma, a sua arquitetura funciona perfeitamente com a instalação de variados *plugins* e extensões de navegadores (SILVA, 2009, p. 29).

2.5.11 *Ajax*

O *Asynchronous JavaScript and XML* (*Ajax*) trata-se de um conjunto de técnicas que partem do principio de utilizar tecnologias *Web* e recursos já existentes para otimizar a interatividade dos usuários com as páginas *Web* (GONÇALVES, 2006).

O *Ajax* possibilita carregar e renderizar páginas, utilizando para isso recursos de *scripts* que se comunicam com um servidor *Web*, efetuando a busca e carregamento dos dados em segundo plano, sem que exista a necessidade do navegador recarregar a página em utilização (ROSA, 2009).

Como citado anteriormente, Niederauer (2007) menciona que o *Ajax* utiliza um conjunto de tecnologias bem conhecidas para os desenvolvedores, entre elas podemos citar a linguagem *JavaScript*, o modelo DOM, o formato *Extensible Markup Language* (*XML*), as linguagens de marcação *HTML* ou *eXtensible Hypertext Markup Language* (*XHTML*) e as folhas de estilo *CSS*, e o objeto *JavaScript XMLHttpRequest*.

Todo o processo de requisição do *Ajax* funciona através da linguagem *JavaScript* (NIEDERAUER, 2007). O objeto de comunicação assíncrona *XMLHttpRequest* é a principal tecnologia envolvida, pois é este objeto que possibilita que apenas parte da página seja carregada (NIEDERAUER, 2007).

O modelo DOM faz parte da interação dinâmica no processo do *Ajax*, pois trata-se de um agregado de rotinas que garante o acesso e a modificação de

documentos no formato “.html”, contribuindo para o dinamismo e aperfeiçoamento na manipulação das informações utilizadas (NIEDERAUER, 2007).

De acordo com Majer (2008), com a utilização de *Ajax*, os formulários de cadastro, por exemplo, se tornaram muito mais dinâmicos, onde há a possibilidade de acessar um banco de dados e efetuar a busca de uma “identificação” digitada, e posteriormente a página retorna um resultado à consulta, no caso disponível ou indisponível, por exemplo.

Seu uso possibilita que a informação seja exibida de maneira mais ágil e precisa, além da vantagem de que apenas parte da página é carregada, possibilitando que o usuário realize outras tarefas enquanto a comunicação com o servidor é realizada (MAJER, 2008).

2.5.12 *MySQL*

Segundo PISA (2012), o *MySQL* trata-se de um sistema gerenciador de banco de dados relacional, desenvolvido com o intuito inicial de trabalhar com certas aplicações de pequeno a médio porte, mas com o passar dos anos passou a atender a grandes aplicações. Utiliza a linguagem *Structured Query Language* (*SQL*), que permite armazenar, gerenciar e recuperar dados (PISA, 2012).

De acordo com Milani (2007), o *MySQL* é o banco de dados de código aberto mais requisitado em aplicações para soluções *Web* e lojas virtuais, pois as mesmas exigem acesso rápido para gerar páginas em *HTML*. Devido ao fato de armazenar dados em tabelas utilizando códigos de baixo nível, o *MySQL* garante extrema rapidez e confiança (MILANI, 2007).

Milani (2007) explica em seu livro que dentre as características principais do *MySQL* encontram-se a alta capacidade de armazenamento e a realização do gerenciamento ao multiacesso e a integridades dos dados. O *MySQL* é uma aplicação compatível com vários sistemas, plataformas e compiladores, pois “seu desenvolvimento se deu a partir de linguagens de programação como C e C++” (MILANI, 2007).

3 DESENVOLVIMENTO

Nesta seção será apresentado o trabalho idealizado e sua execução, incluindo explicações sobre o processo de desenvolvimento do *software* juntamente com a elaboração dos testes.

3.1 TRABALHO IDEALIZADO

Este trabalho visa aplicar testes automatizados juntamente com a prática da Integração Contínua numa aplicação *Web*, utilizando um sistema de controle de versão, para que cada integrante da equipe implemente seus códigos e uma ferramenta de integração para realizar o monitoramento do processo de desenvolvimento.

Foram realizados estudos relacionados a testes e Integração Contínua, juntamente com a utilização das ferramentas *Jenkins*, *Selenium* e *GitHub* para efetuar as verificações das funcionalidades de uma aplicação desenvolvida na linguagem *PHP* (*Hypertext Preprocessor*) utilizando *CSS* (*Cascading Stytle Sheets*, HTML (*HyperText Markup Language*), *JavaScript* e um SGBD (Sistema Gerenciador de Banco de Dados).

3.2 DESCRIÇÃO DA APLICAÇÃO

A aplicação desenvolvida para a realização dos testes simula o controle de um estacionamento. Este controle utiliza a linguagem *PHP*, juntamente com *HTML*, *JavaScript* , *CSS*, e o sistema gerenciador de banco de dados *MySQL*.

As ferramentas utilizadas para o desenvolvimento foram *EasyPHP* e *Sublime Text 2*. Nos computadores onde os códigos eram elaborados, o *Git* foi instalado para efetuar a transferência das modificações no ambiente integrado. Através da ferramenta *Git* o controle das versões do *software* também pode ser verificado.

O acesso na página do sistema criado é realizado através de um ambiente *web*. O usuário pode utilizar o navegador de sua preferência, desde que este seja compatível com as ferramentas citadas anteriormente. É aconselhável verificar se o *JavaScript* é suportado pelo navegador, caso contrário, a aplicação não funcionará corretamente.

Utilizando os novos conceitos e bibliotecas – citados na seção 2 – que visam a maior interatividade do usuário com os sistemas web, a interface demonstrada na figura 6 foi elaborada:

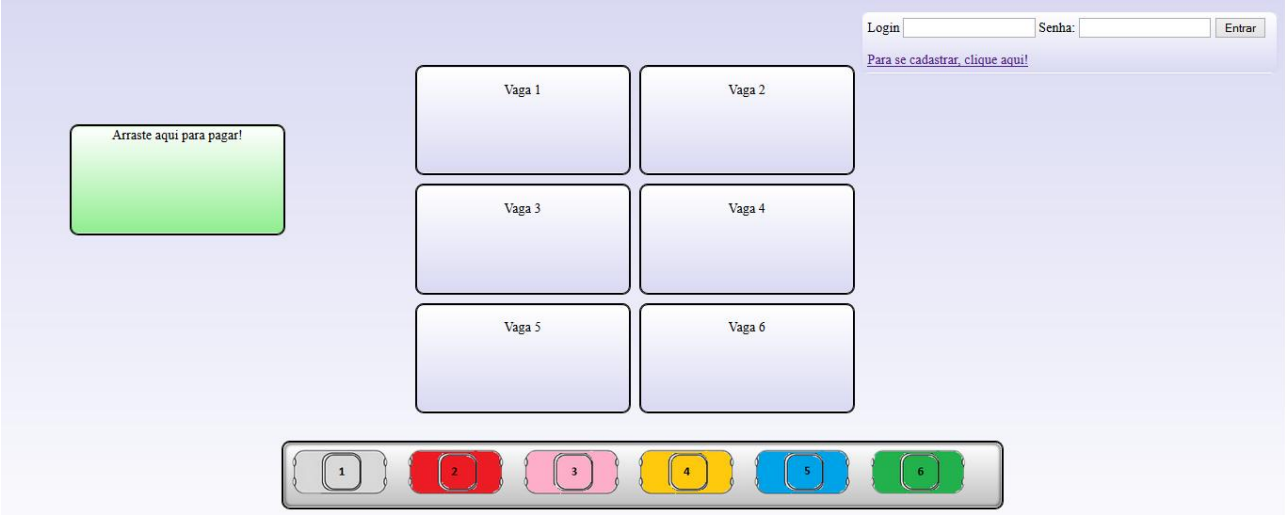


Figura 6 - Página Inicial do sistema desenvolvido.
Fonte: Autoria Própria

A página está organizada em *divs* (LEMON, 2005), as quais permitem organizar e estilizar os conteúdos separadamente. Cada *div* está utilizando o atributo "*id*" (W3SCHOOLS, 2014). Este atributo realiza a identificação e a unicidade de cada elemento do código (W3SCHOOLS, 2014). Por exemplo, a vaga nº 1 pode ser identificada pelo *id* "v1", e o carro nº 3 pelo *id* "c3".

É importante ressaltar a criação de duas *divs*: a que possui *id* "carros", e que possui *id* "vagas". Na *div id* "carros" foram agrupados todos os carros existentes na interface. Da mesma forma, na *div id* "vagas", todas as vagas existentes foram agrupadas.

Esta organização em conjunto permite a manipulação de todas as vagas ou de todos os carros de uma só vez. Para isso, é necessário utilizar como referência os *ids* definidos para o grupo, como por exemplo, *id* "vagas". Também é possível trabalhar com uma única vaga ou um único carro, basta ter como referência os *ids*

de cada elemento, como por exemplo, *id* “v1”, a qual faz menção apenas para a vaga nº 1.

O diferencial deste sistema foi forma do uso do *JavaScript* para controlar a interface. Os métodos *draggable* e *droppable* da biblioteca *jQuery* tornaram possíveis as interações de arrastar e soltar os carrinhos para preencher vagas, ou para efetuar pagamento.

A figura 7 exemplifica a ocupação de uma vaga realizada através dos métodos mencionados anteriormente.

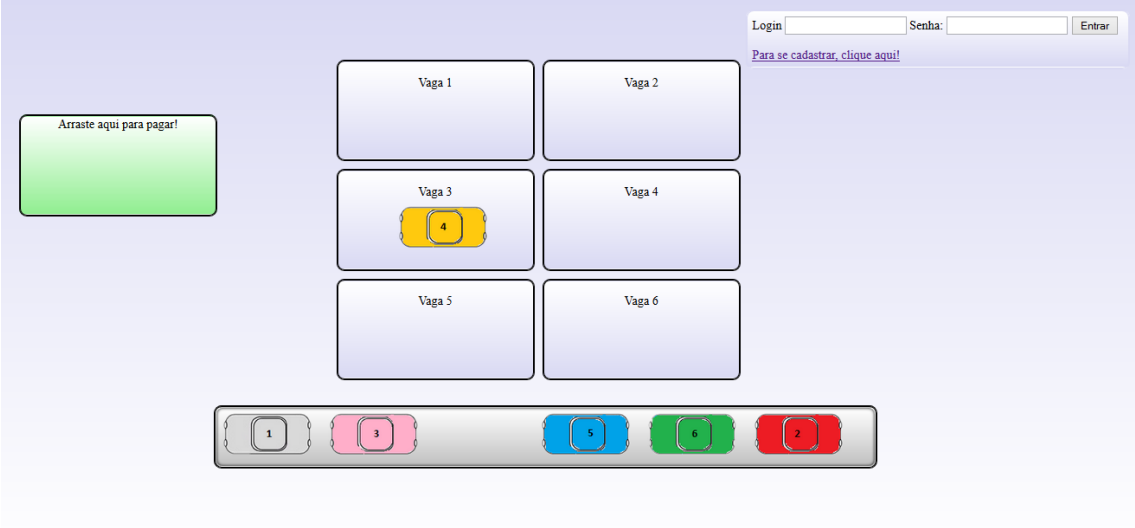


Figura 7 - Exemplo da ocupação de uma vaga.
Fonte: Autoria Própria

O método *draggable* (JQUERYUI, 2014) foi aplicado na imagem dos carrinhos. Com sua utilização foi possível mover estas imagens pela página. Para isto, basta o usuário clicar com o mouse em cima e arrastar o carrinho até o destino desejado. Abaixo é demonstrado o código da função *draggable* usado no sistema.

```
$("div#carros img").draggable({ revert: "invalid" });
```

O atributo *“revert”* foi definido como *“invalid”* para que os carrinhos retornem ao seu ponto inicial caso eles não sejam arrastados para um elemento *droppable*.

Um elemento *droppable* recebe elementos *draggable* (JQUERYUI, 2014). O método *droppable* (JQUERYUI, 2014) foi aplicado nas *divs* das vagas e do pagamento. Isto possibilitou a efetivação do recebimento da imagem dos carros

nestas *divs*. Para isto, basta o usuário soltar a imagem no destino desejado, seja para efetuar um pagamento ou ocupar uma vaga.

Na figura 8 é demonstrado um trecho do código da função *droppable* usado no sistema.

```
$("#vagas div").droppable({
    accept: function(event, ui){
        pai = event.parent().attr("id");
        ocupaVaga = event.attr("data-vaga");
        estado = $(this).attr("data-vazio");

        if(ocupaVaga==""){
            if(pai == "carros" && isVazio(estado)){
                return true;
            }
        }
        return false;
    },
    tolerance: "fit",
    drop: function(event, ui){
        estado = $(this).attr("data-vazio");
        vagaCarro = ui.helper.attr("data-vaga");

        if(isVazio(estado) && !carroPossuiVaga(vagaCarro)){
            idVaga = $(this).attr("id");
            idCarro = ui.helper.attr("id");

            ui.helper.attr("data-vaga", idVaga);
            $(this).attr("data-vazio","false");

            $("#"+idCarro).appendTo($(this)).removeAttr("style");

            $.ajax({
                 url: "banco/verifica.php",
                 type: "GET",
                 data:'idVaga='+ idVaga +'&idCarro='+ idCarro,
                 success: function(data){
                        if(data!="false"){
                            alert("Vaga ocupada com Sucesso");
                        }
                    }
            });
```

Figura 8 - Trecho de código do sistema – Função *droppable* utilizada nas vagas.
Fonte: Autoria Própria

Algumas validações se tornaram necessárias, como por exemplo, não permitir que dois ou mais carros ocupem a mesma vaga. Para isso, foi definido nas *tags* dos elementos um atributo de controle.

Nas *tags* referentes às vagas, o atributo “data-vazio” recebe o valor de “*true*” quando a vaga está desocupada, e “*false*” quando está ocupada por um carro. Assim, toda vez que o usuário tentar ocupar uma vaga, o valor deste atributo é verificado. Se o valor for “*true*”, o carrinho é aceito na vaga, caso contrário, ele volta para o seu posicionamento inicial.

O atributo de controle “data-vaga” das *tags* referente aos carros funciona de maneira semelhante. Ele armazena o *id* da vaga que está sendo ocupada. Caso o carro não esteja ocupando nenhuma vaga, o valor armazenado é nulo. Este atributo é verificado cada vez que o usuário tentar realizar uma ocupação. É através dele que o sistema bloqueia a troca do carro de uma vaga para outra.

Uma vez ocupada uma vaga, o único movimento aceito para retirar o carro daquela posição é movê-lo para a realização do pagamento. Logo após, o sistema trata de mover o carro para a lista inicial através do comando *appendTo* apresentado no trecho de código *JavaScript* abaixo:

```
$("#"+idCarro).appendTo("#carros").removeAttr("style");
```

A utilização do comando *removeAttr(“style”)* se refere à uma solução encontrada para que a imagem do carro não adquira um posicionamento aleatório. Assim, ela retorna exatamente para o retângulo da lista de carros, herdando o estilo correto definido para as demais imagens.

A recuperação e a gravação dos dados no banco *MySQL* ocorrem através da utilização do *Ajax*. Esta funcionalidade possibilita a atualização de informações na página sem a necessidade de efetuar o *reload* no navegador.

Após as verificações apresentadas, no pagamento, por exemplo, é realizada uma requisição *Ajax*. Através desta requisição, o caminho para o arquivo que fará a manipulação dos dados deve ser especificado, bem como as informações da vaga e do carro. Também é aconselhável definir o tipo de transferência, seja ele *GET* ou *POST*. O trecho de código *JavaScript* a seguir mostra uma das requisições realizadas no sistema.

```
$.ajax({
        url: "banco/pagamento.PHP",
        type: "GET",
        data:'idVaga='+ vagaCarro +'&idCarro='+ idCarro,
        success: function(data){
                mostrapagamento(data);
                $("#"+idCarro).appendTo("#carros").removeAttr("style");
        }
});
```

A função de *callback* “*success”* (GRONER, 2014) representa o código que será executado caso a conexão com a página definida na “*url”* seja efetuada com êxito. A variável “data” contém o retorno do servidor. Esta variável representa o resultado dos comandos realizados com base nos dados transferidos, onde neste caso são os *ids* da vaga e do carro.

Os comandos existentes dentro da função “mostrapagamento(data)” servem para alertar ao usuário as informações referentes ao pagamento, como demonstrado na figura 9.

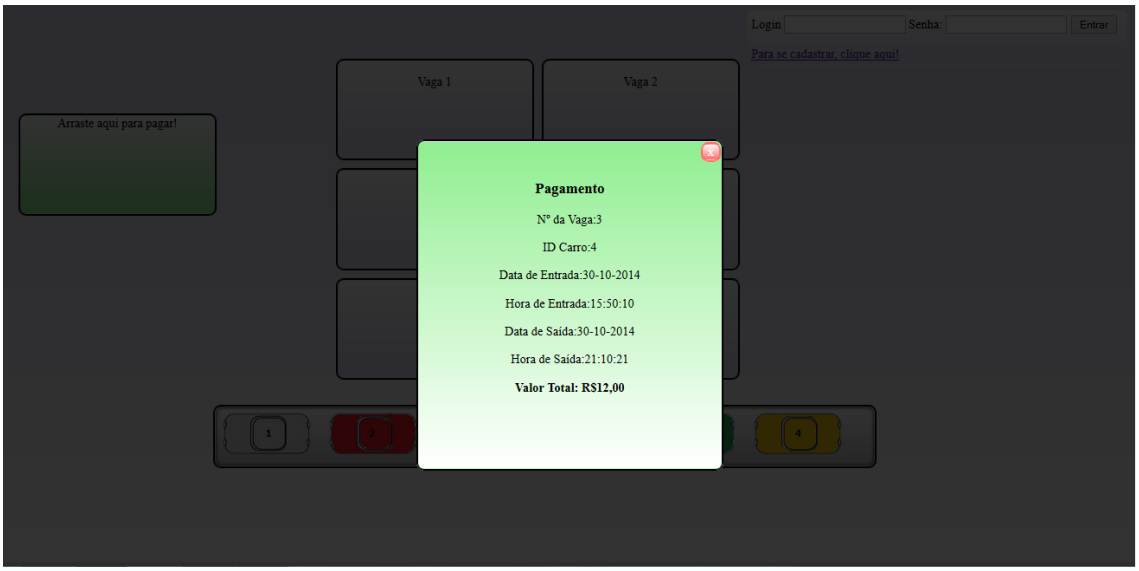


Figura 9 - Página que demonstra os dados de um pagamento.
Fonte: Autoria Própria

Os valores das datas e dos horários são inseridos nas tabelas do banco *MySQL* através do tipo de dado definido como “*default CURRENT_TIMESTAMP*” (MYSQL, 2014). A diferença entre o horário de entrada e o horário de saída da vaga também é calculado diretamente pelo Sistema Gerenciador de Banco de Dados através da função “*TIMEDIFF ()*” (MYSQL, 2014).

Para efeitos de pagamento, o sistema armazena diretamente no código fonte o valor monetário cobrado por hora utilizada pelo carro. Este valor está definido como R$ 2,00.

Nesta aplicação é realizado o arredondamento do tempo ocupado em uma vaga a fim de obter uma hora completa. Sendo assim, basta efetuar o cálculo entre o preço e a permanência calculada do carro na vaga. Estas informações são

armazenadas na tabela “pagamento” no banco de dados através de um arquivo *PHP*.

O sistema também possui uma função de verificação inicial. No primeiro acesso à pagina, é realizada uma busca no banco de dados através do *Ajax*. Esta consulta verifica a situação de cada vaga. Caso ela esteja sendo ocupada, a imagem do carro que a está ocupando é movida para o seu devido local. Assim, o sistema inicia mostrando ao usuário as vagas livres, e as vagas ocupadas pelos carrinhos.

No canto superior direito, nota-se um formulário de *login*. O usuário também pode efetuar seu cadastro no sistema, clicando no *link “*Para se cadastrar clique aqui!*”.* Na figura 10 é demonstrada a tela de formulário para o usuário informar seus dados.

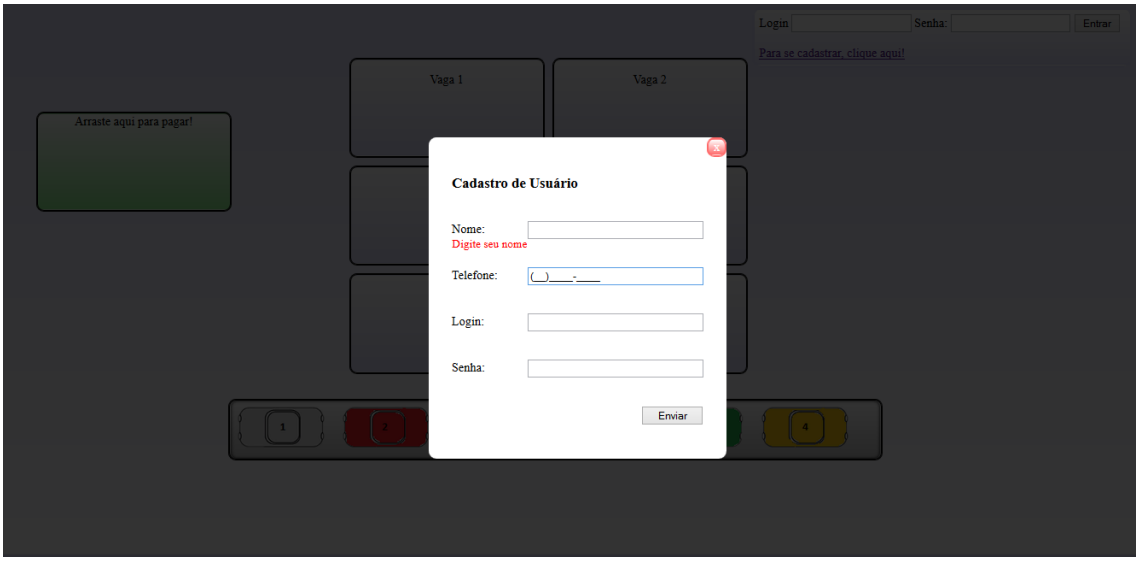


Figura 10 - Formulário de cadastro de usuário.
Fonte: Autoria Própria

Para a realização deste formulário foram utilizadas novamente as funcionalidades da biblioteca *jQuery*, tais como os métodos “*mask”* (JQUERY,2014) e “*validate”* (JQUERYVALIDATION,2014).

O método “*mask”* possibilitou a definição da quantidade de caracteres a ser preenchido pelo usuário no campo telefone, bem como a inserção de parênteses para separar o código de área. O código abaixo demonstra como isso foi realizado de forma simples e objetiva.

```
$("#telefone").mask("(99)9999-9999");
```

O método “*validate”* contribuiu para as validações nos campos do formulário. Podem-se citar como exemplo os alertas para informar ao usuário a exigência mínima de caracteres ou campos em branco.

Pela figura 10 é possível perceber a mensagem “Digite seu nome”, pois o usuário clicou primeiramente no campo de preenchimento telefone. Estas mensagens podem ser definidas pelo desenvolvedor dentro da função “*messages”*, a qual faz parte do método “*validate”*.

Ao clicar no botão “Enviar”, os dados são enviados para a verificação no banco de dados através do *Ajax*. Caso o *login* não conste na tabela de usuários, é emitida uma mensagem de que o cadastro foi efetuado com sucesso, senão, será necessário digitar esta informação com outro valor. O *login* confere a unicidade de cada usuário no sistema.

Para sair da tela de cadastro, basta clicar no “x” em vermelho, localizado no canto superior direito. Quando isto ocorre, a página inicial é mostrada novamente.

O formulário de *login* realiza a autenticação mediante as informações digitadas pelo usuário. Novamente utilizamos *Ajax* para a consulta no banco de dados. Se o usuário não estiver cadastrado ou digitar uma informação incorreta, a mensagem “Login não efetuado” é mostrada logo abaixo do formulário. Caso o *login* seja efetuado com sucesso, os dados são armazenados numa sessão (PHP, 2014), e a mensagem de boas vindas é emitida ao usuário, juntamente com um *link* para sair do sistema.

Este *login* não protege as alterações efetuadas no controle de estacionamento, sendo desenvolvido com o objetivo de contribuir para a fase de testes.

3.3 REPOSITÓRIO DE INTEGRAÇÃO

Para a conclusão da aplicação, várias versões foram realizadas, as quais serão abordadas na seção de Resultados (seção 4). A cada nova versão, esta era submetida ao repositório do *website GitHub* (GITHUB, 2014), a qual foi chamada de “IntegraçãoContínua”.

A realização destas transferências foi mediante a instalação da ferramenta *Git.* As configurações foram realizadas no *prompt* de comandos *Git Bash*. A definição de usuário e senha e a criação de uma chave para o acesso ao repositório “IntegraçãoContínua” podem ser citadas como exemplos.

Inicialmente foi necessário efetuar a cópia do repositório do ambiente *web* para a pasta destinada à integração do computador. O comando utilizado foi:

$ git clone git://github.com/gionvanegalvao/IntegracaoContinua.git

Efetuado este comando (GIT-SCM, 2014), todos os arquivos existentes no repositório do *website* definido no *link* serão submetidos para a pasta local.

Após o desenvolvedor realizar as suas modificações nestes arquivos, é necessário submeter essas alterações para o *website*. Para que isto ocorra, é os *commits* são realizados (GITREF, 2014).

Deve-se abrir o *prompt* de comandos *Git Bash* do repositório e efetuar os seguintes comandos (CODEXICO, 2010):

- Adicionar um repositório ao *git* que foi iniciado neste diretório (*add remote)*, elaborar um nome para o projeto (*origin)*, e citar o endereço do repositório do *website (git@github.com:giovanegalvao/IntegracaoContinua.git*): *$ git remote add origin git@github.com:giovanegalvao/IntegracaoContinua.git*
- Baixar (*pull*) o projeto mencionando a origem (*origin)* e o destino (*master)*: *$ git pull origin master*
- Realizar o *commit* nas alterações, definindo uma mensagem entre as aspas duplas: *$ git commit -m “meu primeiro commit”*
- Enviar (*push)* as alterações: *$ git push origin master*

Para verificar se o repositório no *website* recebeu as alterações, basta acessá-lo pela internet. O site do *GitHub* informa qual usuário realizou o último *commit*, e a data em que ele ocorreu.

3.4 PROBLEMAS ENCONTRADOS NA CONFIGURAÇÃO DO *JENKINS*

Foi encontrado um problema de configuração durante a instalação do *Jenkins*. Ao rodar um *job*, os testes acusavam erro ao iniciar o navegador através do usuário anônimo - usuário que o *Jenkins* utiliza para rodar os testes.

Como todos os testes dependiam de executar o navegador para abrir a aplicação, buscou-se a solução para o problema. No entanto, este processo estava levando um tempo considerável, o qual comprometeria o andamento do projeto.

As configurações básicas da ferramenta, tais como indicar os caminhos do *JDK* instalado, *Ant* para compilação dos testes, *Git* para realizar comunicação dos arquivos na máquina com o repositório do *GitHub* e a configuração da chave de acesso ao *GitHub*, foram realizadas com êxito.

O grande obstáculo foi a configuração de um nó que torna possível o acesso às informações da máquina para o usuário do *Jenkins* efetuar os testes.

Para manter o foco do trabalho na automatização dos testes e na Integração Contínua, decidiu-se utilizar a ferramenta *Ant* que também faz parte do *Jenkins*. Assim, optou-se por aplicar o *Ant* para rodar os testes implementados no sistema de controle de estacionamento.

3.5 IMPLEMENTAÇÃO DOS TESTES

Neste trabalho foram implementados testes de aceitação (caixa preta) – seção 2.3.1. O primeiro passo tomado para desenvolver os casos de testes foi conhecer as técnicas utilizadas pelo desenvolvimento e planejar as tarefas.

Na figura 11 é possível verificar a interface da versão inicial do projeto elaborado – seção 3.2.

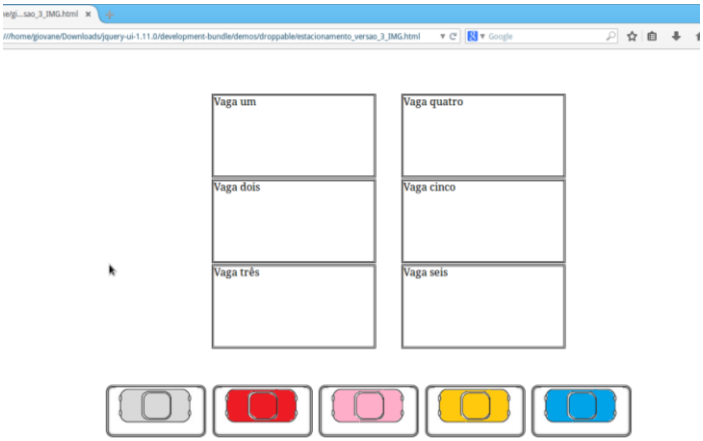


Figura 11- Página Inicial do sistema, versão 1
Fonte: Autoria Própria

Com a ferramenta *Selenium IDE* foi possível gerar as ações dos casos de testes planejados. Esta ferramenta possui comandos que devem ser destacados:

- *"DragAndDrop"* : útil para arrastar um elemento de uma posição e soltá-lo em outra (SELENIUM, 2008). Na sua sintaxe é necessário fornecer o elemento de origem de localização na coluna Target e o deslocamento (x,y) em pixel (localização atual até o local de destino onde você quer deixar cair elemento) na coluna "*Value*" (SELENIUM, 2008).
- *"dragAndDropToObject"* : similar ao comando *"DragAndDrop"*. Para este comando é necessário fornecer um elemento localizador de objeto a ser arrastado na coluna Target e um elemento localizador de destino do objeto na coluna "*Value*" (SELENIUM, 2008).
- *"mouseDown"* : simula a ação de um usuário pressionando o botão esquerdo do mouse no elemento alvo (SELENIUM, 2008). Ele apenas simula o pressionamento do botão esquerdo do mouse, sem liberá-lo (SELENIUM, 2008).
- *"mouseMoveAt"* : simula a ação de um usuário movendo o mouse no elemento alvo (SELENIUM, 2008). Também recebe valores (x,y) para que o mouse se mova nas coordenadas especificadas (SELENIUM, 2008).
- *"mouseUp"* : comporta-se como se um usuário estivesse pressionado o botão esquerdo do mouse no elemento alvo (SELENIUM,2008).

Existem duas maneiras de gerar *scripts* com casos de testes pela ferramenta. É possível gravar as ações que são executadas na interface da aplicação, ou o *tester* pode definir os comandos na ferramenta.

A figura 12 exemplifica a implementação destes *scripts* no *Selenium IDE*. Os campos "Comando", "Alvo" e "Valor" da aplicação são de suma importância para efetuar os testes.

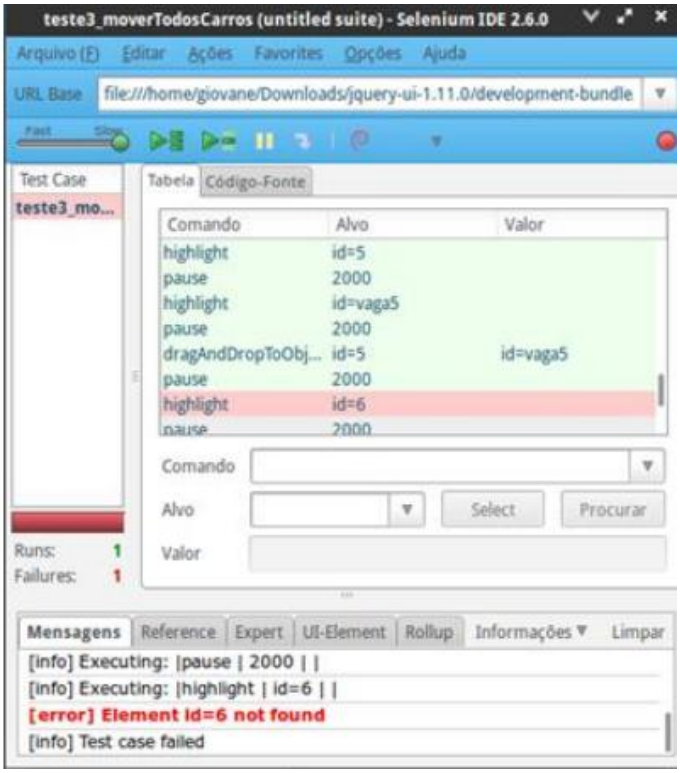


Figura 12 - Implementação de casos de testes com *Selenium IDE*.
Fonte: Autoria Própria

O campo “Comando” serve para digitar a ação que será executada, como por exemplo, o “*mouseUp”* explicado anteriormente. O campo “Alvo” indica o elemento que deve ser afetado pelo comando, normalmente é definido pelo atributo “*id”*. Por fim, o campo “Valor” indica o novo valor que o alvo deverá assumir como resultado final (SELENIUM, 2014). Um valor possível para ser definido é o “*true*” ou “*false*” para testes de ocupação de vaga – seção 3.2.

Também é possível escrever *scripts* para fazer os testes na ferramenta do *Selenium IDE*. Basta implementar os mesmos métodos que são acessados quando o usuário efetua a interação na página. O *script* pode ser interpretado clicando no botão “*Play*” da ferramenta.

É possível gerar um arquivo na própria ferramenta com linguagem Java tendo como base os *scripts* implementados, basta acessar seguir o caminho “Arquivo – Exportar teste como – Java/*JUnit*4/Remote Control” e escolher o local para salvar. Esse arquivo é uma classe *JUnit* – seção 2.3.1, Figura 2 – e roda com o servidor *Selenium RC.*

Para auxiliar na execução dos testes foi utilizado o *Selenium Remote Control (RC).* Ele é uma ferramenta que permite escrever testes automatizados de

aplicações *Web* em qualquer linguagem de programação (SELENIUM, 2014). A figura 13 mostra o comando para iniciar o servidor *Selenium RC*.

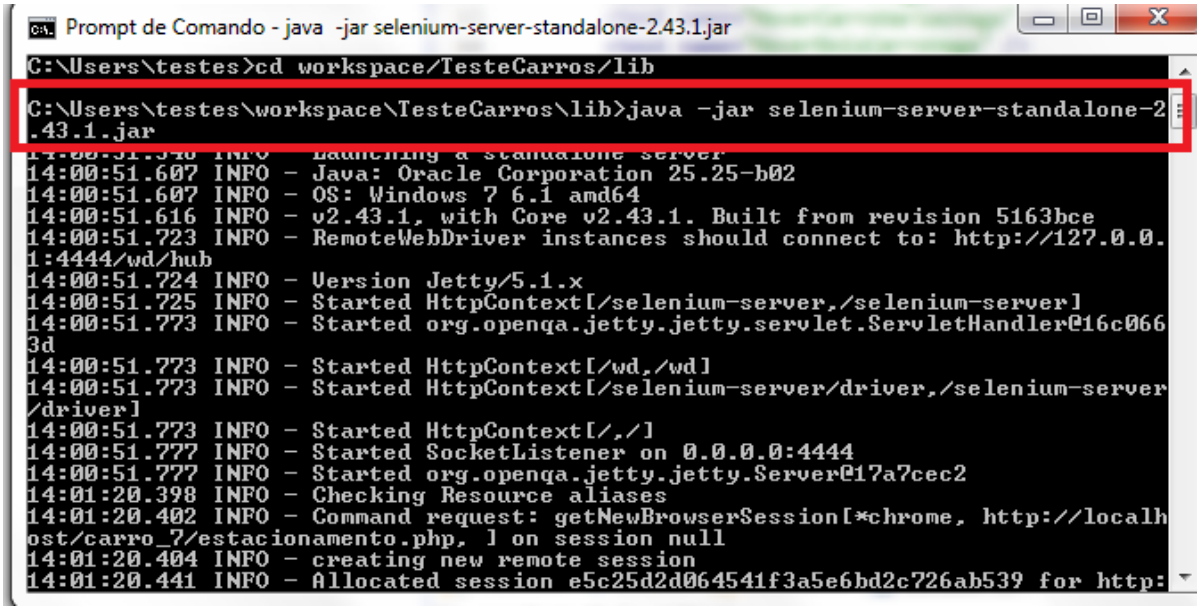


Figura 13 - Iniciando o servidor *Selenium RC*
Fonte: Autoria Própria

Para automatizar o processo de execução dos testes, utilizou-se o arquivo *build.xml*. Este por sua vez é proveniente de uma ferramenta de *build* disponível em Java (CAELUM, 2014). A ideia desse arquivo é gerar uma série de tarefas que são invocadas através de *tags XML*. Isto permite a execução de todas as classes de teste geradas.

A estrutura deste arquivo é composta por diversas *tags*. De acordo com Adams (2011) o *build.xml* é organizado da seguinte forma:

- *project*: representa o projeto desenvolvido;
- *target*: elemento alvo que serão aplicados as tarefas;
- *task*: tarefa que será realizada;

A *target* utilizada foi a que possui o atributo *name "test"*, e para a definição da execução dos testes foi utilizado a *tag* <*test*>. Para inserir as classes de testes que devem rodar na execução do *JUnit* através do *Ant*, há a necessidade de incluí-las na *tag* <*test*> como exemplificado na figura 14:

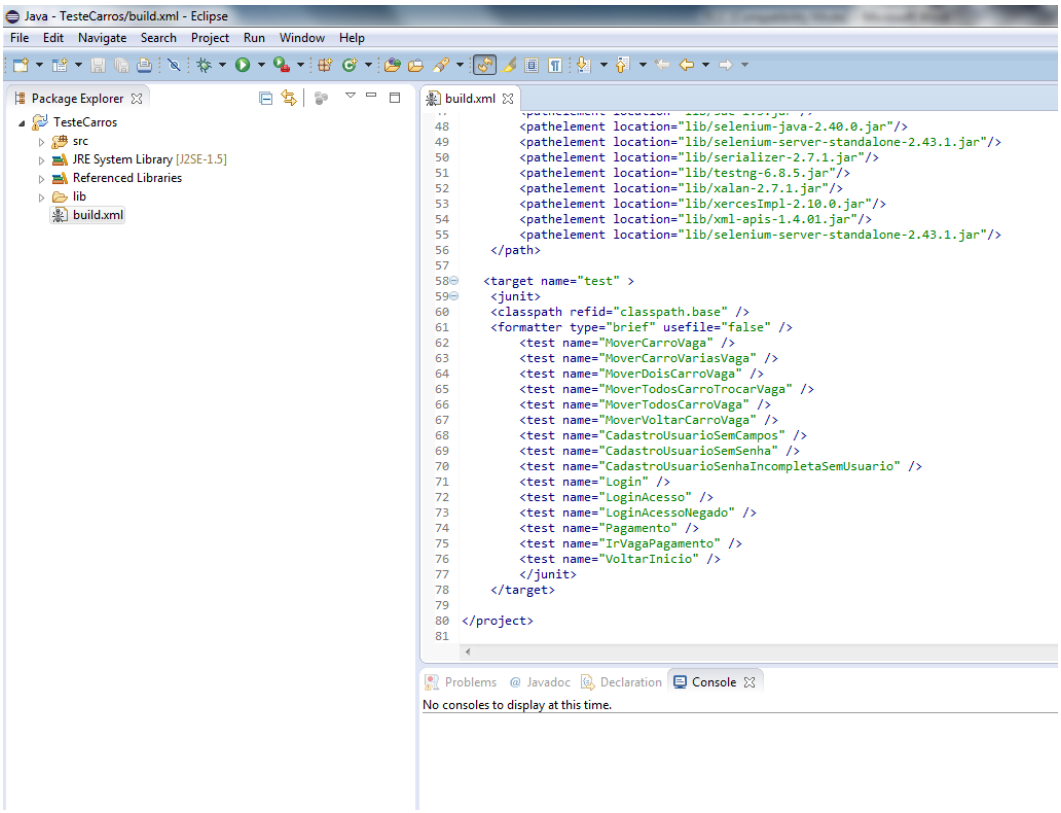


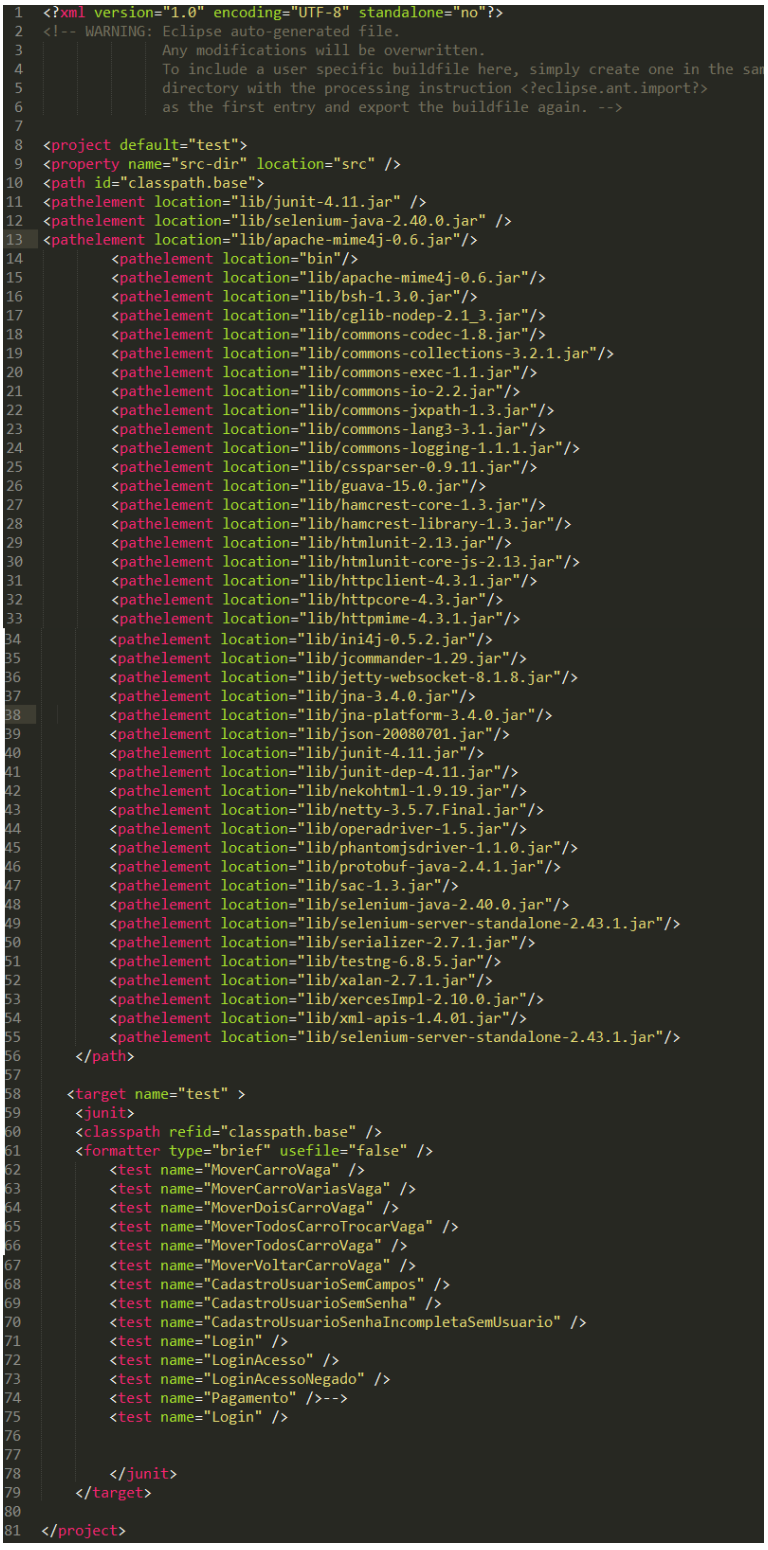
Figura 14 - Projeto de teste no *Eclipse* – *build.xml*
Fonte: Autoria Própria

É preciso definir dentro da *tag project* os arquivos necessários para efetuar a execução das tarefas (ADAMS, 2011). A *tag property* define a localização dos testes – classes *JUnit*. As *tags path* e *pathelement* definem o local dos arquivos das ferramentas que precisam ser iniciadas para rodar os testes. No caso deste trabalho, as classes de teste *JUnit* estão salvas na pasta “src” e os arquivos das ferramentas estão salvos na pasta “lib”. A seguir é demonstrado o trecho do arquivo deste trabalho com a utilização destas *tags.*

```
<property name="src-dir" location="src" />
<path id="classpath.base">
    <pathelement location="lib/JUnit-4.11.jar" />
    <pathelement location="lib/selenium-java-2.40.0.jar" />
    <pathelement location="lib/apache-mime4j-0.6.jar"/>
</path>
```

A ferramenta de desenvolvimento *Eclipse IDE* auxiliou na criação deste arquivo. Com o projeto aberto nesta ferramenta, foi realizada a criação do arquivo e a inserção das *tags* relacionadas anteriormente.

A imagem 15 mostra todo o arquivo *build.xml* deste trabalho.



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- WARNING: Eclipse auto-generated file.
              Any modifications will be overwritten.
              To include a user specific buildfile here, simply create one in the sam
              directory with the processing instruction <?eclipse.ant.import?>
              as the first entry and export the buildfile again. -->

<project default="test">
<property name="src-dir" location="src" />
<path id="classpath.base">
<pathelement location="lib/junit-4.11.jar" />
<pathelement location="lib/selenium-java-2.40.0.jar" />
<pathelement location="lib/apache-mime4j-0.6.jar"/>
        <pathelement location="bin"/>
        <pathelement location="lib/apache-mime4j-0.6.jar"/>
        <pathelement location="lib/bsh-1.3.0.jar"/>
        <pathelement location="lib/cglib-nodep-2.1_3.jar"/>
        <pathelement location="lib/commons-codec-1.8.jar"/>
        <pathelement location="lib/commons-collections-3.2.1.jar"/>
        <pathelement location="lib/commons-exec-1.1.jar"/>
        <pathelement location="lib/commons-io-2.2.jar"/>
        <pathelement location="lib/commons-jxpath-1.3.jar"/>
        <pathelement location="lib/commons-lang3-3.1.jar"/>
        <pathelement location="lib/commons-logging-1.1.1.jar"/>
        <pathelement location="lib/cssparser-0.9.11.jar"/>
        <pathelement location="lib/guava-15.0.jar"/>
        <pathelement location="lib/hamcrest-core-1.3.jar"/>
        <pathelement location="lib/hamcrest-library-1.3.jar"/>
        <pathelement location="lib/htmlunit-2.13.jar"/>
        <pathelement location="lib/htmlunit-core-js-2.13.jar"/>
        <pathelement location="lib/httpclient-4.3.1.jar"/>
        <pathelement location="lib/httpcore-4.3.jar"/>
        <pathelement location="lib/httpmime-4.3.1.jar"/>
        <pathelement location="lib/ini4j-0.5.2.jar"/>
        <pathelement location="lib/jcommander-1.29.jar"/>
        <pathelement location="lib/jetty-websocket-8.1.8.jar"/>
        <pathelement location="lib/jna-3.4.0.jar"/>
        <pathelement location="lib/jna-platform-3.4.0.jar"/>
        <pathelement location="lib/json-20080701.jar"/>
        <pathelement location="lib/junit-4.11.jar"/>
        <pathelement location="lib/junit-dep-4.11.jar"/>
        <pathelement location="lib/nekohtml-1.9.19.jar"/>
        <pathelement location="lib/netty-3.5.7.Final.jar"/>
        <pathelement location="lib/operadriver-1.5.jar"/>
        <pathelement location="lib/phantomjsdriver-1.1.0.jar"/>
        <pathelement location="lib/protobuf-java-2.4.1.jar"/>
        <pathelement location="lib/sac-1.3.jar"/>
        <pathelement location="lib/selenium-java-2.40.0.jar"/>
        <pathelement location="lib/selenium-server-standalone-2.43.1.jar"/>
        <pathelement location="lib/serializer-2.7.1.jar"/>
        <pathelement location="lib/testng-6.8.5.jar"/>
        <pathelement location="lib/xalan-2.7.1.jar"/>
        <pathelement location="lib/xercesImpl-2.10.0.jar"/>
        <pathelement location="lib/xml-apis-1.4.01.jar"/>
        <pathelement location="lib/selenium-server-standalone-2.43.1.jar"/>
    </path>

   <target name="test" >
    <junit>
    <classpath refid="classpath.base" />
    <formatter type="brief" usefile="false" />
        <test name="MoverCarroVaga" />
        <test name="MoverCarroVariasVaga" />
        <test name="MoverDoisCarroVaga" />
        <test name="MoverTodosCarroTrocarVaga" />
        <test name="MoverTodosCarroVaga" />
        <test name="MoverVoltarCarroVaga" />
        <test name="CadastroUsuarioSemCampos" />
        <test name="CadastroUsuarioSemSenha" />
        <test name="CadastroUsuarioSenhaIncompletaSemUsuario" />
        <test name="Login" />
        <test name="LoginAcesso" />
        <test name="LoginAcessoNegado" />
        <test name="Pagamento" />-->
        <test name="Login" />


        </junit>
    </target>

</project>
```

Figura 15 - **Arquivo *build.xml***
Fonte: Autoria Própria

Por fim, após ter construído o arquivo *build.xml* é necessário colocá-lo em execução. Desta forma, todas as classes de teste que foram inseridas neste arquivo

serão iniciadas. Para isso deve-se abrir outra janela no *prompt* de comando, entrar no repositório em que se encontram as classes e o arquivo *build.xml*, e executar o comando "*ant test*". Na figura 16 é ilustrado este comando realizado para rodar o *Ant*, ferramenta que executará a construção e dependências do arquivo *build*, e mostrará o *feedback* do *JUnit* em relação a cada teste iniciado.

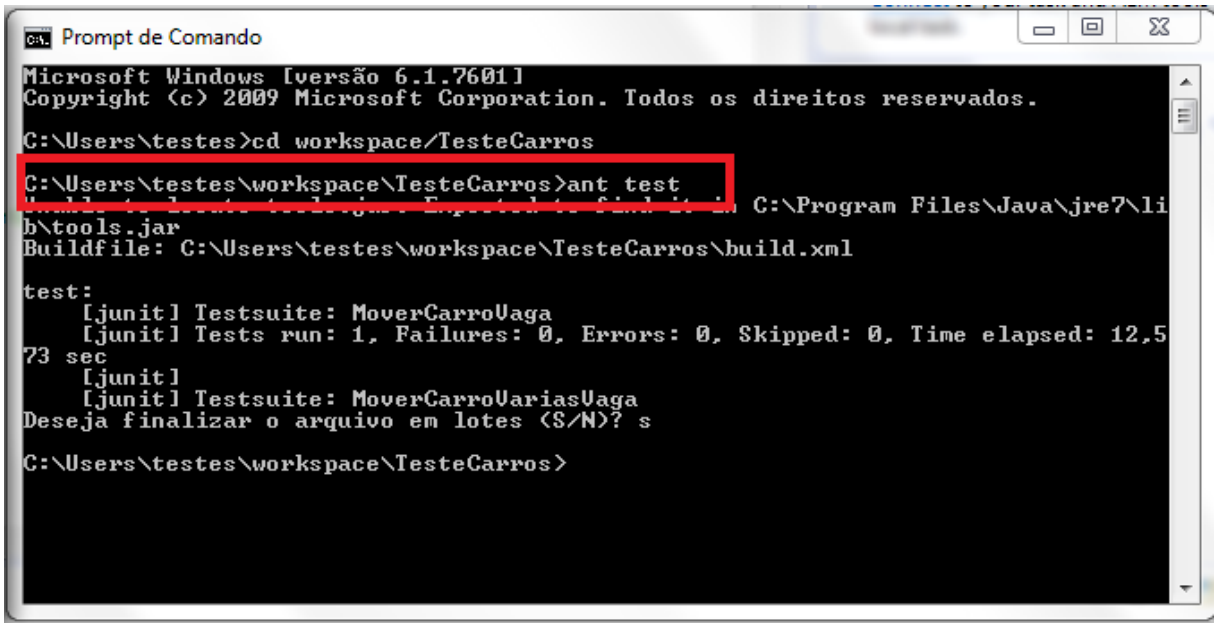


Figura 16 - Rodando os testes implementados
Fonte: Autoria Própria

O *feedback* apresentado indicará quais testes foram executados, e a quantidade de falhas e erros. Para os testes que encontram erros, a ferramenta disponibiliza uma mensagem indicando mais detalhes da falha.

Para facilitar a visualização, foi executado apenas o teste "MoverCarroVaga" no arquivo *build* do exemplo. No entanto, como explicado anteriormente, pode-se executar todos os testes, seguindo a sequência definida do arquivo *buid*. Para cada um dos testes é informado o resumo da execução demonstrada na figura 16.

4 RESULTADOS

A seguir são apresentadas as interfaces e os resultados obtidos após a execução dos testes de aceitação para cada um dos *releases* do sistema desenvolvido para este trabalho.

4.1 TESTES PRIMEIRO *RELEASE*

Nesta seção podemos verificar os testes efetuados na primeira versão do controle de estacionamento.

- Mover um carro para uma vaga: neste teste foi validada a ação de arrastar um carro para uma vaga. Este caso de teste verifica a situação mais simples que o usuário poderia fazer ao acessar a aplicação. Este teste não encontrou falhas, pois através da figura 17 pode-se notar o carro cinza ocupando a vaga número dois do sistema.

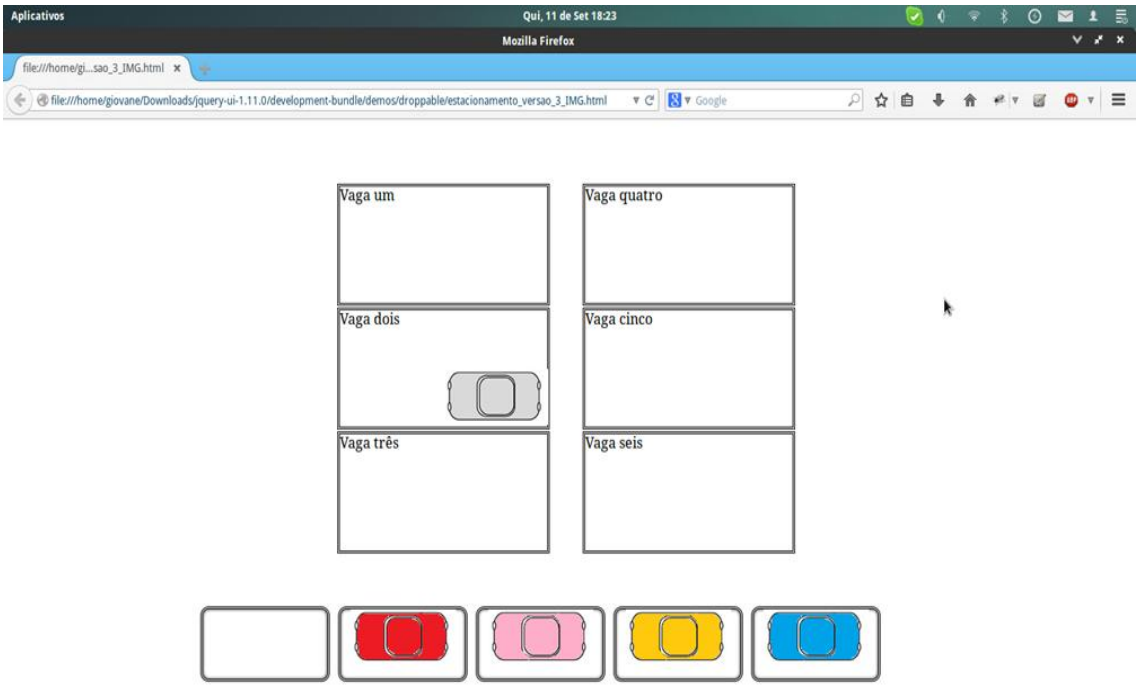


Figura 17 - Resultado do Teste “Mover um carro para uma vaga”
Fonte: Autoria Própria

- Mover dois carros para uma vaga: neste teste foi validada a regra de negócio da aplicação. A verificação se resume na possibilidade de uma vaga ser ocupada por mais de um carro ao mesmo tempo. O sistema não deveria permitir este fato. A figura 18 ilustra o resultado deste teste. Os carros cinza e vermelho estão ocupando a mesma vaga.

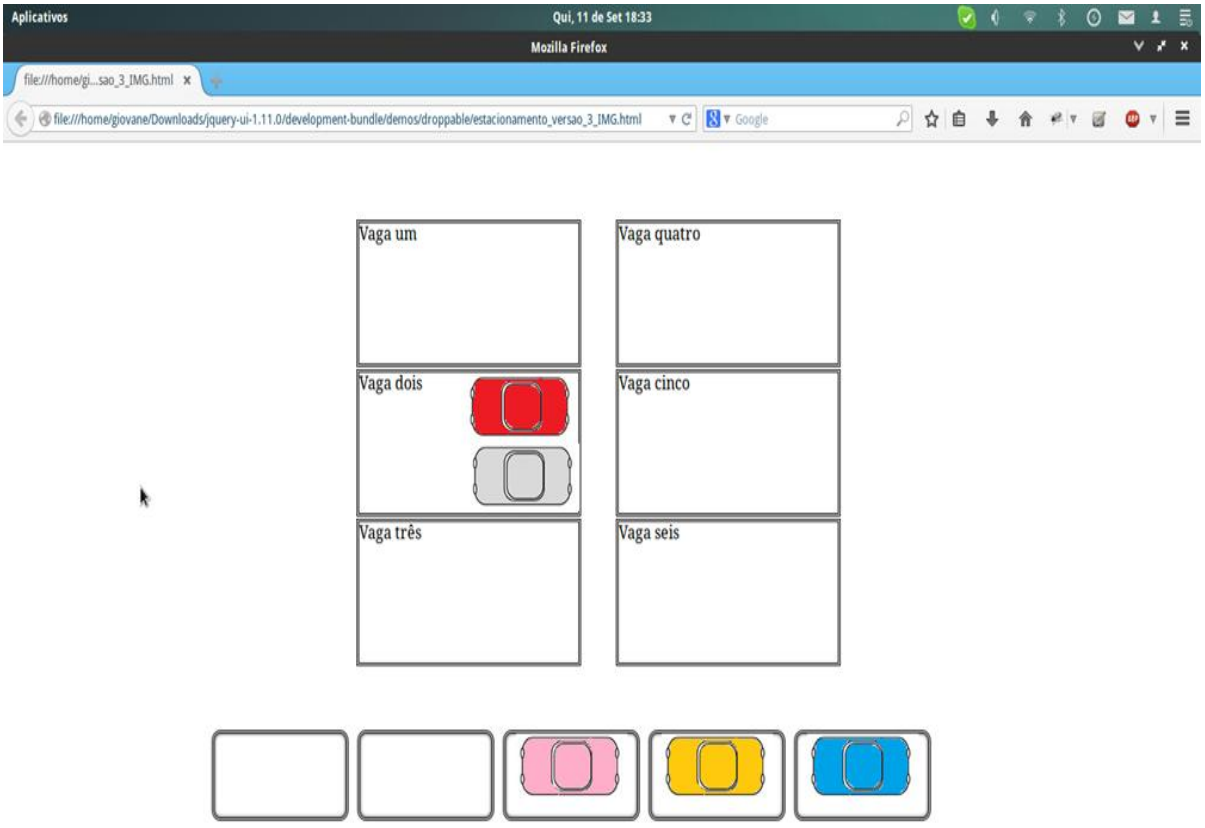


Figura 18 - Resultado do Teste "Mover dois carros para uma vaga"
Fonte: Autoria Própria

Portando foi identificado um erro neste teste. O sistema permitiu inserir dois carros na mesma vaga. Uma possível solução seria a inserção de uma mensagem para o usuário informando quando a vaga já estivesse ocupada.

- Mover todos os carros: outra verificação planejada foi a de mover todos os carros disponíveis nas vagas em aberto. Assim, podemos observar se o número de vagas é igual ou não ao número de carros disponíveis. Como resultado do teste, encontrou-se um erro, devido ao fato de que o número de carros foi diferente do

número de vagas. Com isso, a ferramenta *Selenium IDE* identificou o erro na linha [erro] Element id=6 not found, que pode ser visualizado na figura 19.

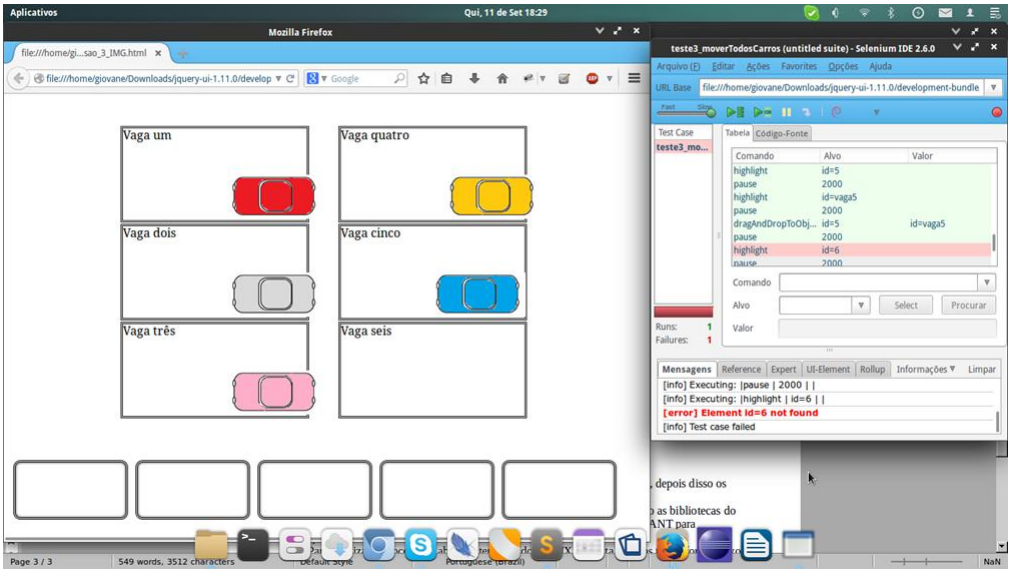


Figura 19 - Resultado do Teste "Mover todos os carros"
Fonte: Autoria Própria

- Colocar um carro e retirar da vaga: neste teste analisamos se o carro após for alocado em uma determinada vaga, poderia sair dela. Deve se observar que nesta versão a implementação da regra de pagamento ainda não havia sido desenvolvida. Através da figura 20, observa-se que neste teste foi encontrado um erro de *script*.

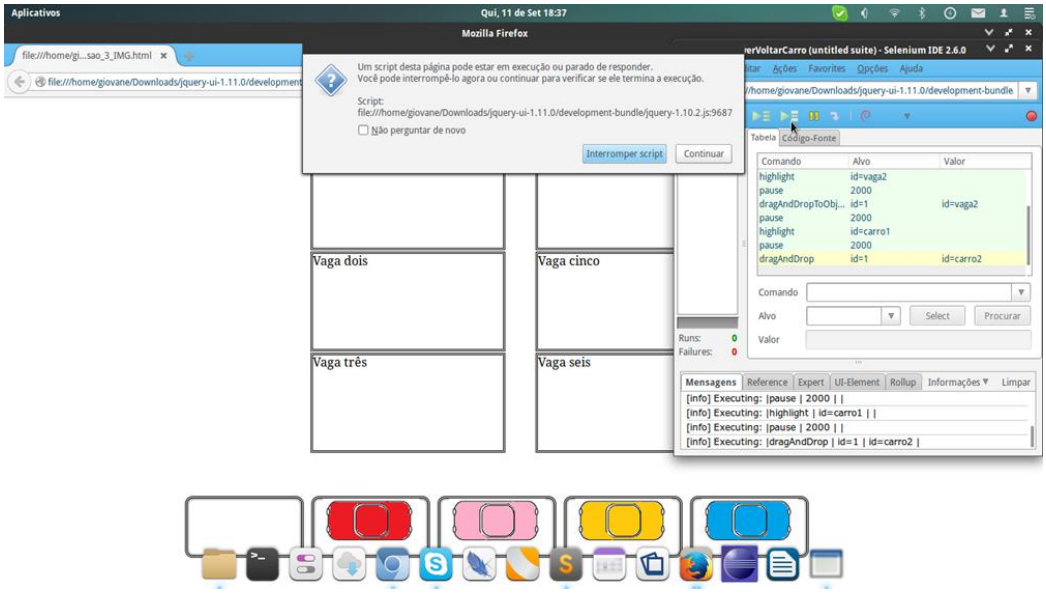


Figura 20 - Resultado do Teste "Colocar um carro e retirar da vaga"
Fonte: Autoria Própria

Ao tentar mover o carrinho para uma vaga e voltá-lo para o local o original, a aplicação travava.

- Mover um carro de uma vaga para outra: neste teste pode-se observar o comportamento da aplicação ao mover um carro que já estava ocupando uma vaga para outra vaga disponível, conforme a figura 21.

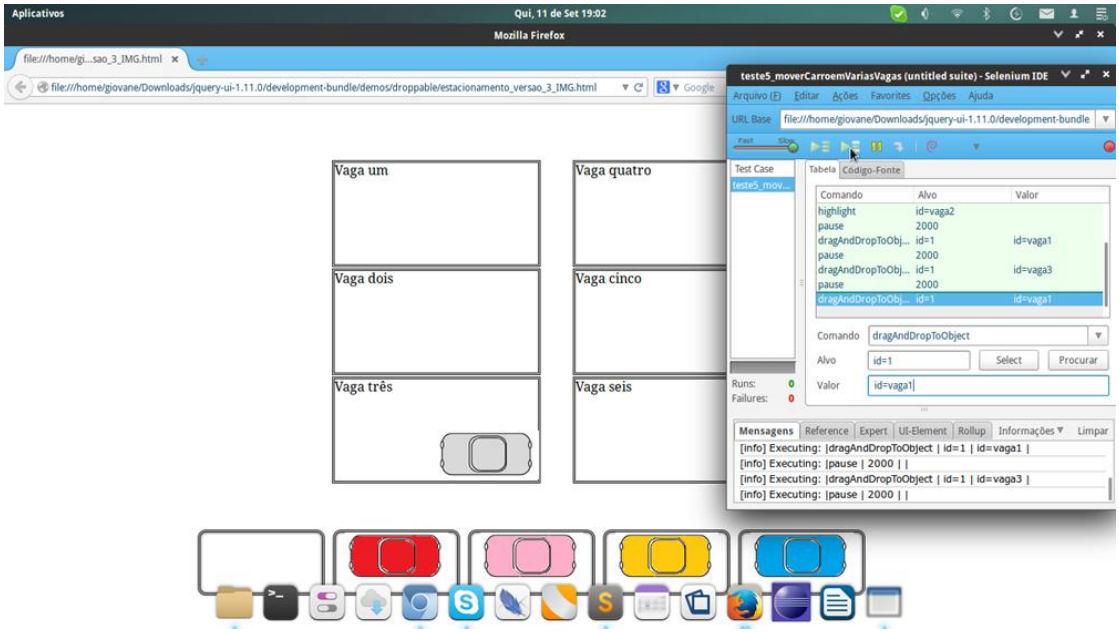


Figura 21 - Resultado do Teste "Mover um carro de uma vaga para outra" – Tela 1
Fonte: Autoria Própria

Na figura 22, pode-se observar a movimentação do carro para outra vaga disponível.

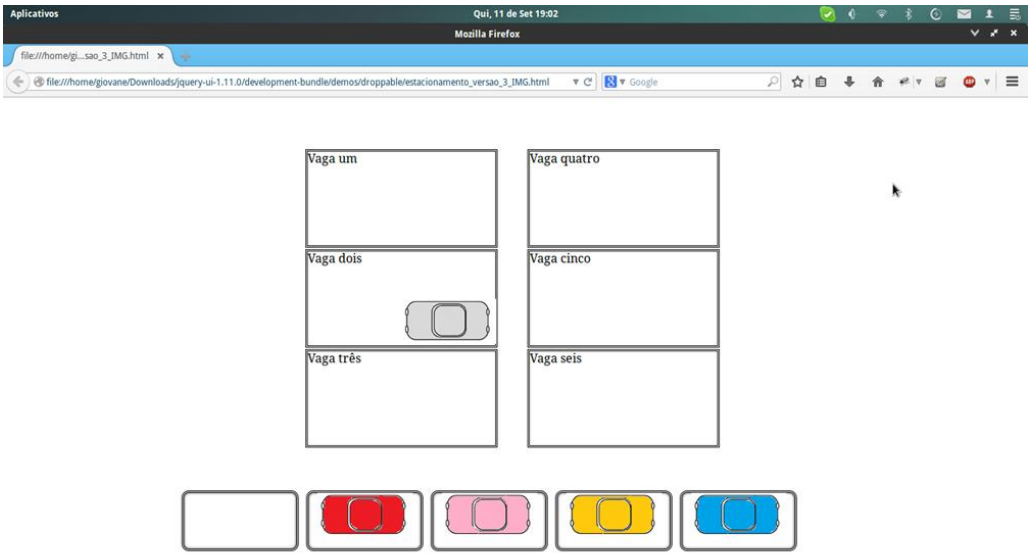


Figura 22 - Resultado do Teste "Mover um carro de uma vaga para outra" – Tela 2
Fonte: Autoria Própria

Portando foi identificado um erro neste teste. O sistema permitiu trocar o carro de vaga sem antes voltar para o seu posicionamento inicial.

- Mover todos os carros em suas respectivas vagas e trocá-los de lugar: outro caso de teste desenvolvido foi a troca de vagas entre os carros. Para isso, um *script* implementa o movimento de todas as imagens dos carros em vagas disponíveis, e depois as troca de lugar. Na figura 23, pode-se observar o movimento de todos os carros para as vagas disponíveis.

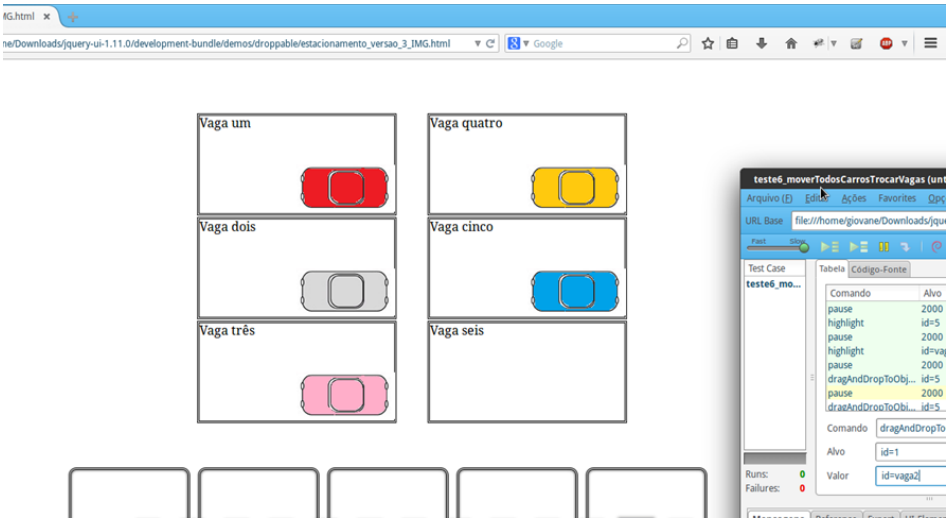


Figura 23 - Resultado do Teste "Mover todos os carros em suas respectivas vagas e trocá-los de lugar"
Fonte: Autoria Própria

A figura 24 representa as trocas de lugares dos carros para novas vagas.

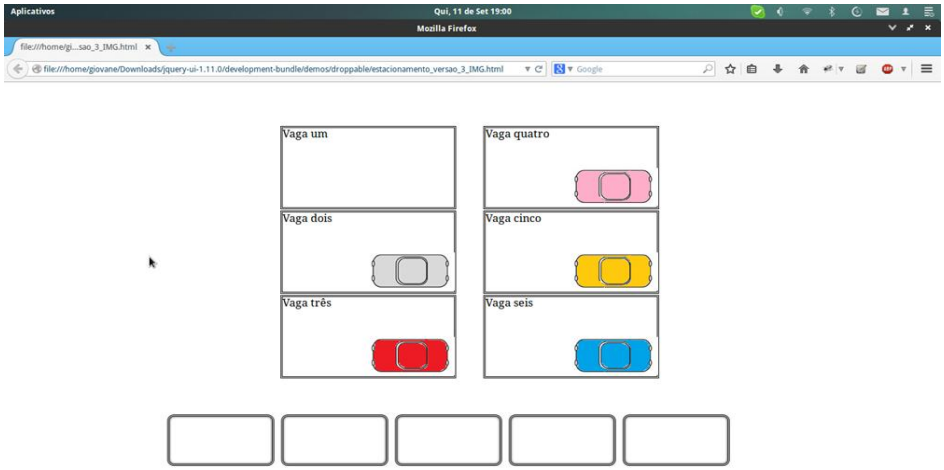


Figura 24 - Resultado do Teste "Mover todos os carros em suas respectivas vagas e trocá-los de lugar" – Tela 2
Fonte: Autoria Própria

Desta forma, um erro foi identificado neste teste. O sistema permitiu trocar os carros de vaga sem antes voltar para o seu posicionamento inicial.

4.2 TESTES SEGUNDO *RELEASE*

Após enviar os testes para o repositório de Integração no *GitHub*, conforme figura 25 – localizado no endereço <https://github.com/giovanegalvao/IntegracaoContinua> – foi necessário realizar o *download* da versão da aplicação corrigida.

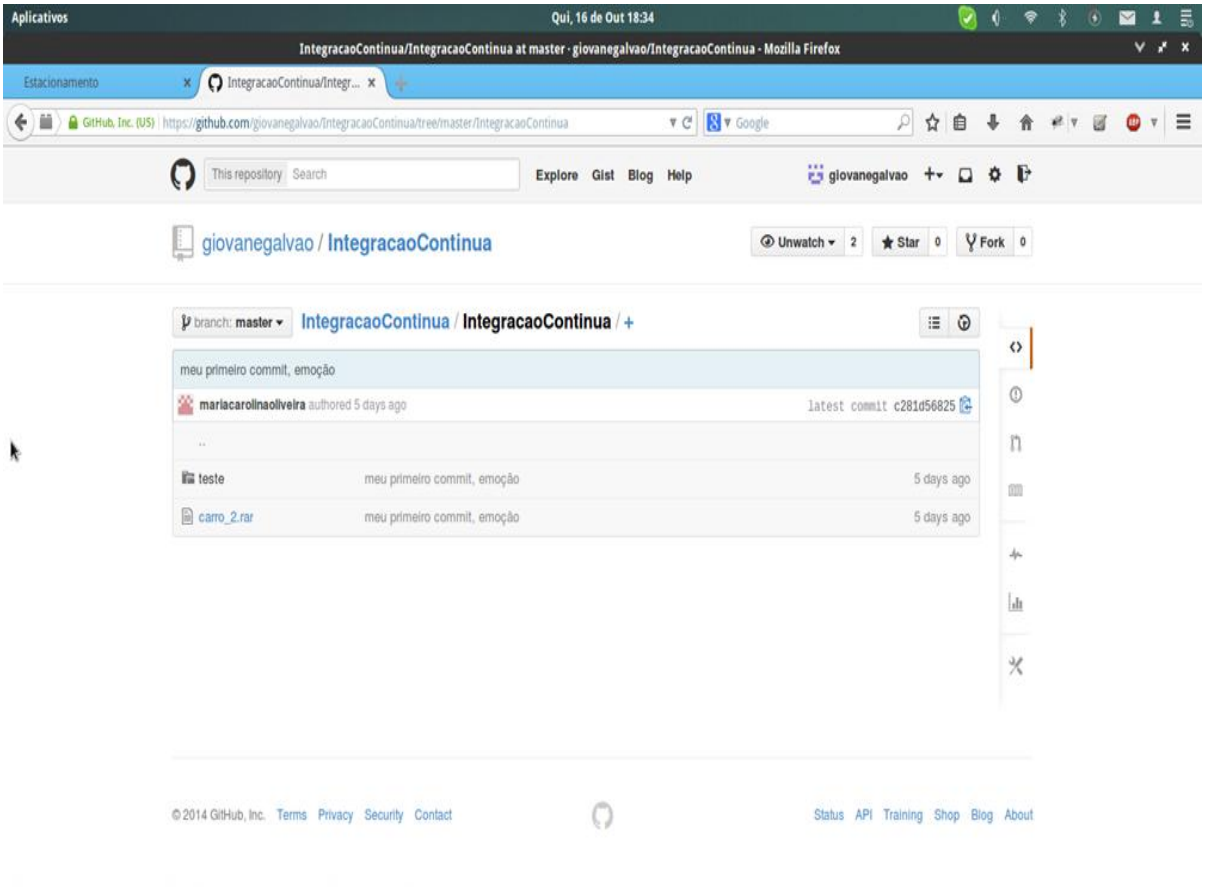


Figura 25 - Repositório *GitHub*
Fonte: Autoria Própria

Em seguida, os mesmos testes de primeiro *release* foram rodados novamente para conferência das correções.

O teste de mover dois carros na mesma vaga que estava com erro na primeira versão, foi corrigido.

O teste de mover todos os carros, um em cada vaga, continua com erro. Além disso, a ferramenta *Selenium* encontrou outro equívoco no código.

O erro aponta que não existe o *id*=carro3 nas *tags* das imagens (figura 26). Após releitura do código, foi verificado que dois carros estavam com o mesmo identificador, onde um deles deveria ser o *id*=carro3.

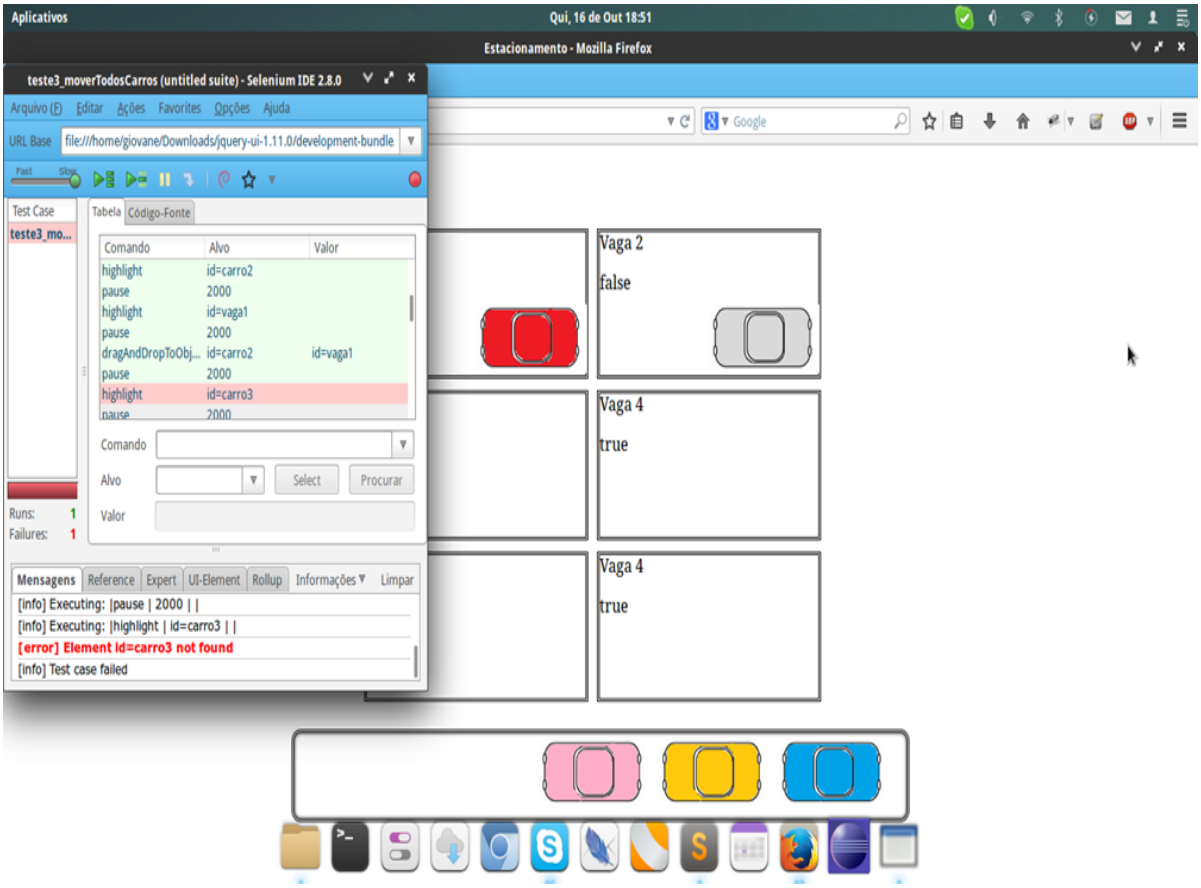


Figura 26 - Resultado Teste "Mover todos os carros" – Segundo *Release*
Fonte: Autoria Própria

No teste de mover o carro para uma vaga e voltar para sua origem, não foi possível utilizar a ferramenta *Selenium*, pois ela retorna erro de *script* do navegador, da mesma forma que a primeira versão.

Com o teste de mover um carro em duas vagas respectivamente, foi encontrado um erro: quando o carro sai de uma vaga para outra, esta última diz estar ocupada, mas na verdade não está.

No último teste implementado na etapa anterior – o teste de mover todos os carros em vagas diferentes e posteriormente trocá-los – a execução no segundo *release* apresentou um erro semelhante (figura 27), o qual verificou a falta do *id*=carro3, indicando que as vagas já estavam ocupadas.

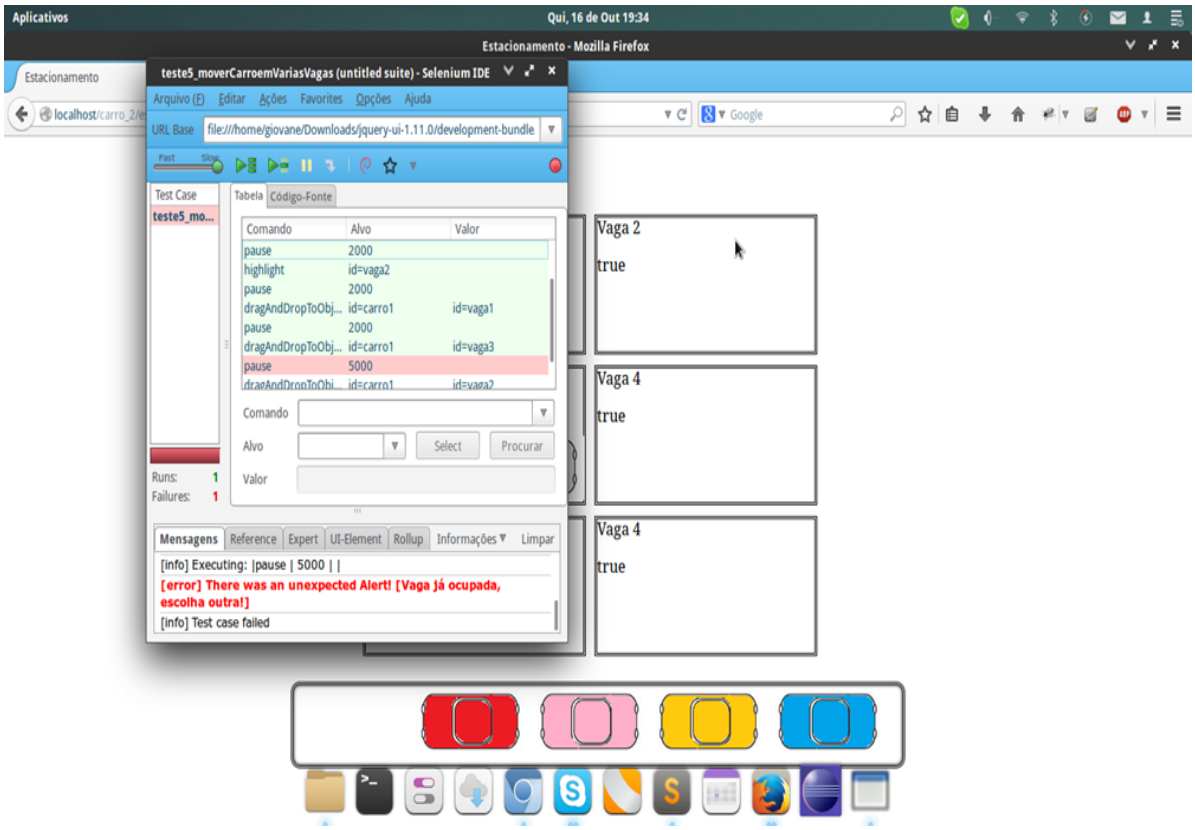


Figura 27 - Resultado do Teste "Mover todos os carros em suas respectivas vagas e trocá-los de lugar" – Segundo Release
Fonte: Autoria Própria

4.3 TESTES TERCEIRO *RELEASE*

Os erros reportados pelo *Ant* nos *releases* anteriores foram corrigidos. Nesta nova versão, as funcionalidade de *login* e de cadastro de usuário foram implementadas. Isso possibilitou a elaboração de novos testes, sendo eles:

- Cadastrar Usuário e realizar *login* no sistema: Neste teste foi verificado o cadastro de um usuário na aplicação. Foi levada em consideração a situação ideal de utilização do sistema, ou seja, o usuário preencheria todos os campos do

formulário de cadastro de forma correta e depois iria realizar o *login* no sistema com estes dados cadastrados. A figura 28 demonstra a situação ideal do preenchimento de formulário de cadastro.

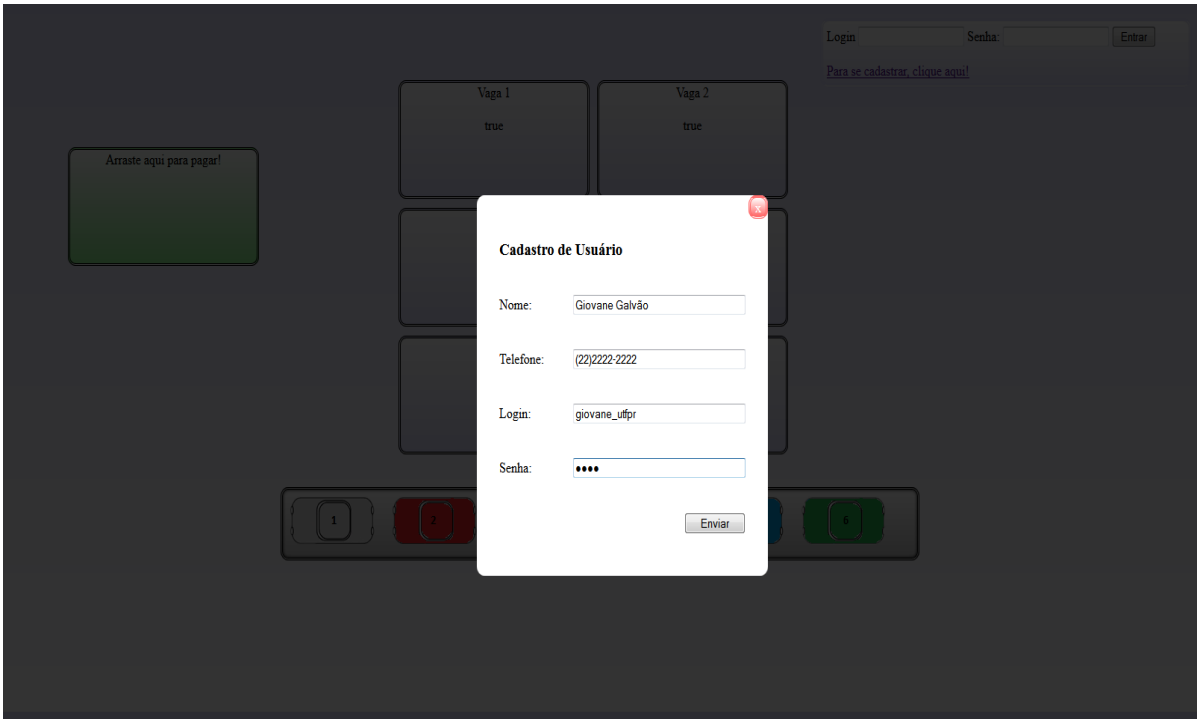


Figura 28 - Resultado do Teste "Cadastrar Usuário e Logar no sistema"
Fonte: Autoria Própria

Na figura 29 pode-se observar o preenchimento do formulário para realizar o *login* no sistema.

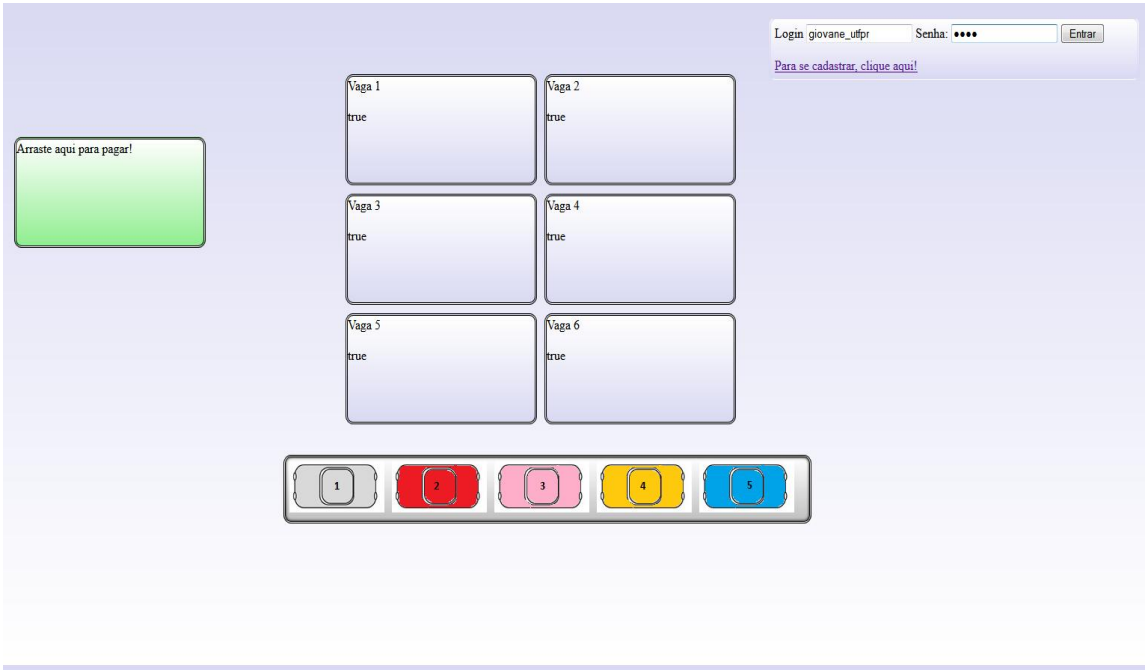


Figura 29 - Resultado do Teste "Cadastrar Usuário e Logar no sistema" - Tela 2
Fonte: Autoria Própria

A figura 30 demonstra um exemplo de um *login* efetuado com sucesso.

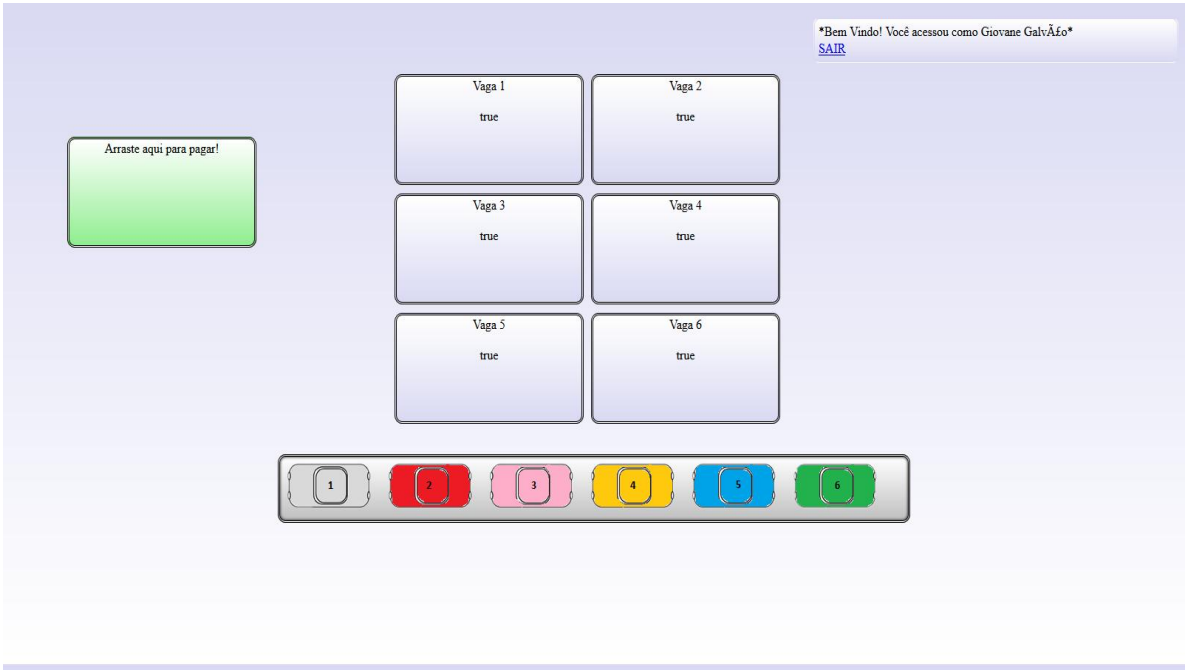


Figura 30 - Resultado do Teste "Cadastrar Usuário e Logar no sistema"- Tela 3
Fonte: Autoria Própria

- Acesso ao site com nenhum usuário: neste teste validamos o acesso do *login*, sem o preenchimento de um usuário e senha, apenas clicando no botão "Entrar". A página informa a mensagem "*Login* não efetuado", conforme figura 31.

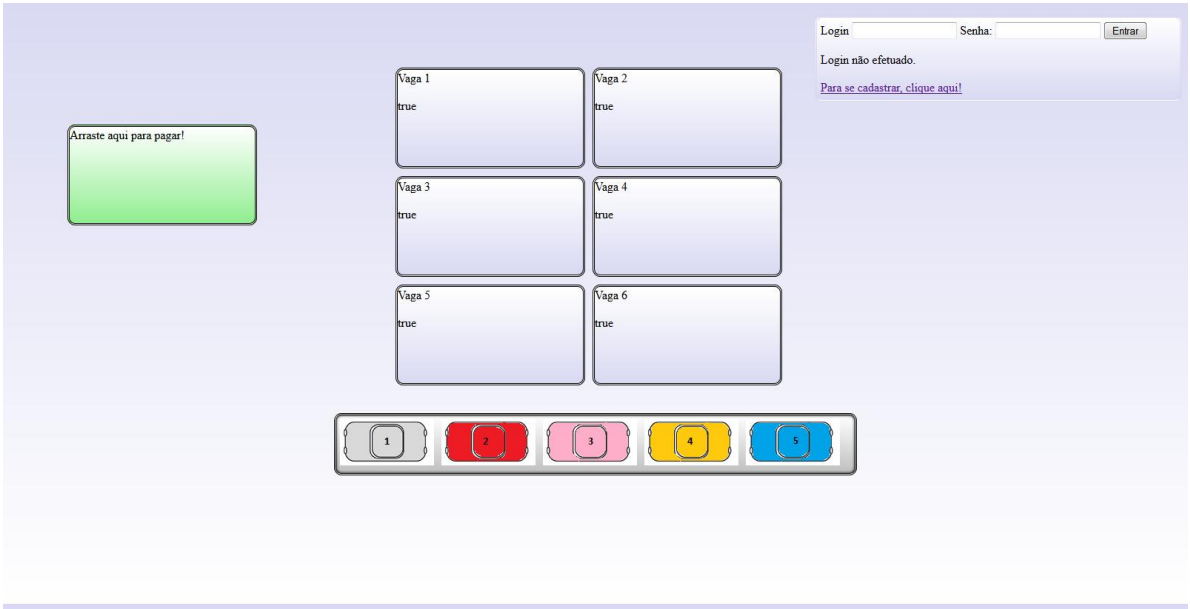


Figura 31 - Resultado do Teste "Acesso ao site com nenhum usuário"
Fonte: Autoria Própria

Outra validação efetuada foi a tentativa de *logar* com um usuário não cadastrado no sistema. O resultado esperado foi uma mensagem alertando ao usuário que seu acesso não foi efetuado, conforme figura 32.

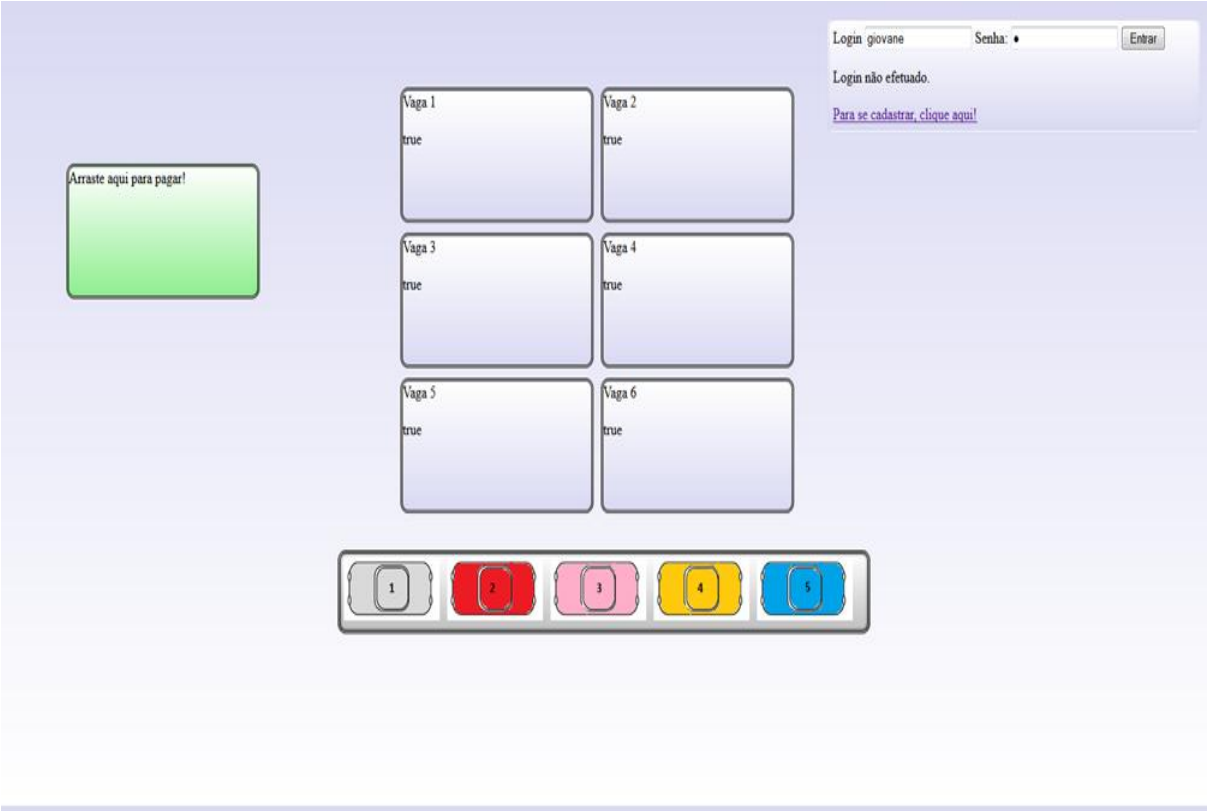


Figura 32 - Acesso ao site com usuário não cadastrado na base de dados
Fonte: Autoria Própria

- Cadastrar usuário sem nenhum campo informado no formulário: Neste teste validamos os campos de preenchimento do formulário de cadastro. Ao deixar os campos obrigatórios sem preenchimento, o resultado esperado e obtido foi mensagens alertando a obrigatoriedade de preencher os campos vazios.

- Cadastrar usuário sem nome e senha com menos de 4 caracteres: Além do teste de campos obrigatórios citados anteriormente, verificou-se a validação dos campos “nome” e “senha”. O campo “nome” possui uma regra que não aceita valores nulos. O campo “senha” possui a regra de que o usuário necessita informar no mínimo 4 caracteres. O resultado apresentado foi a mensagem “A sua senha

deve conter, no mínimo, 4 caracteres.”, pois conforme pode ser observado na figura 33, o usuário não cadastrou uma senha com os requisitos mínimos exigidos.

Figura 33 - Resultado do Teste “Cadastrar Usuário sem nome de usuário e senha com menos de 4 caracteres”
Fonte: Autoria Própria

Não foram encontrados erros nos testes realizados nos formulários de cadastro de usuário e *login*. Pode-se observar esta afirmação na figura 34, a qual demonstra a execução do *JUnit* para rodar os testes através do *Ant*.

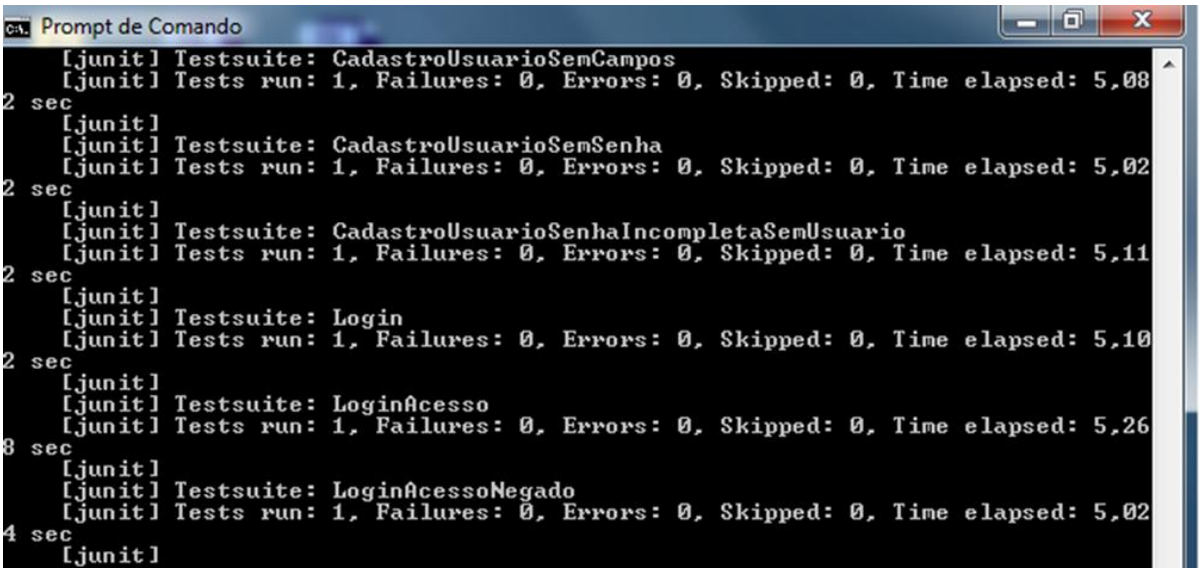


Figura 34 - Resultado dos Testes Terceiro *Release* com *Ant*
Fonte: Autoria Própria

4.4 TESTES QUARTO *RELEASE*

Nesta etapa, foram implementados os testes da funcionalidade de pagamento com base na versão 8 de desenvolvimento. Os testes desenvolvidos foram:

- Efetuar o pagamento após ocupar uma vaga: neste teste, levamos em consideração o caminho correto da aplicação, ou seja, um carro ocupa uma vaga em aberto e depois é movido para a área de pagamento. O resultado esperado é o calculo do tempo e do valor a pagar pelo cliente. O teste obteve sucesso.

- Ir para o pagamento direto sem o carro ocupar uma vaga: neste teste validamos a regra de pagamento, movendo um carro direto para a área de pagamento sem ocupar nenhuma vaga. O resultado esperado pela aplicação era a impossibilidade do usuário completar esta ação.

- Ocupar uma vaga e voltar para o início sem passar na funcionalidade pagamento: neste momento, com a funcionalidade de pagamento implementada, foi possível realizar a validação de ocupação de uma vaga, e fazer o carrinho voltar para a lista inicial, sem realizar o pagamento. No *script* foi implementado a ação de arrastar um carro em uma vaga e tentar voltá-lo para sua posição inicial. O resultado esperado era uma mensagem alertando que o carro não efetuou o pagamento. O teste obteve falha, pois a mensagem não foi apresentada na tela.

Dos testes de pagamento, dois possuíram erros de alerta para indicar que a vaga foi ocupada com sucesso e que o pagamento foi efetuado com sucesso, na última versão de desenvolvimento estas mensagens não foram encontradas.

Na figura 35 pode-se observar os resultados obtidos dos testes efetuados.

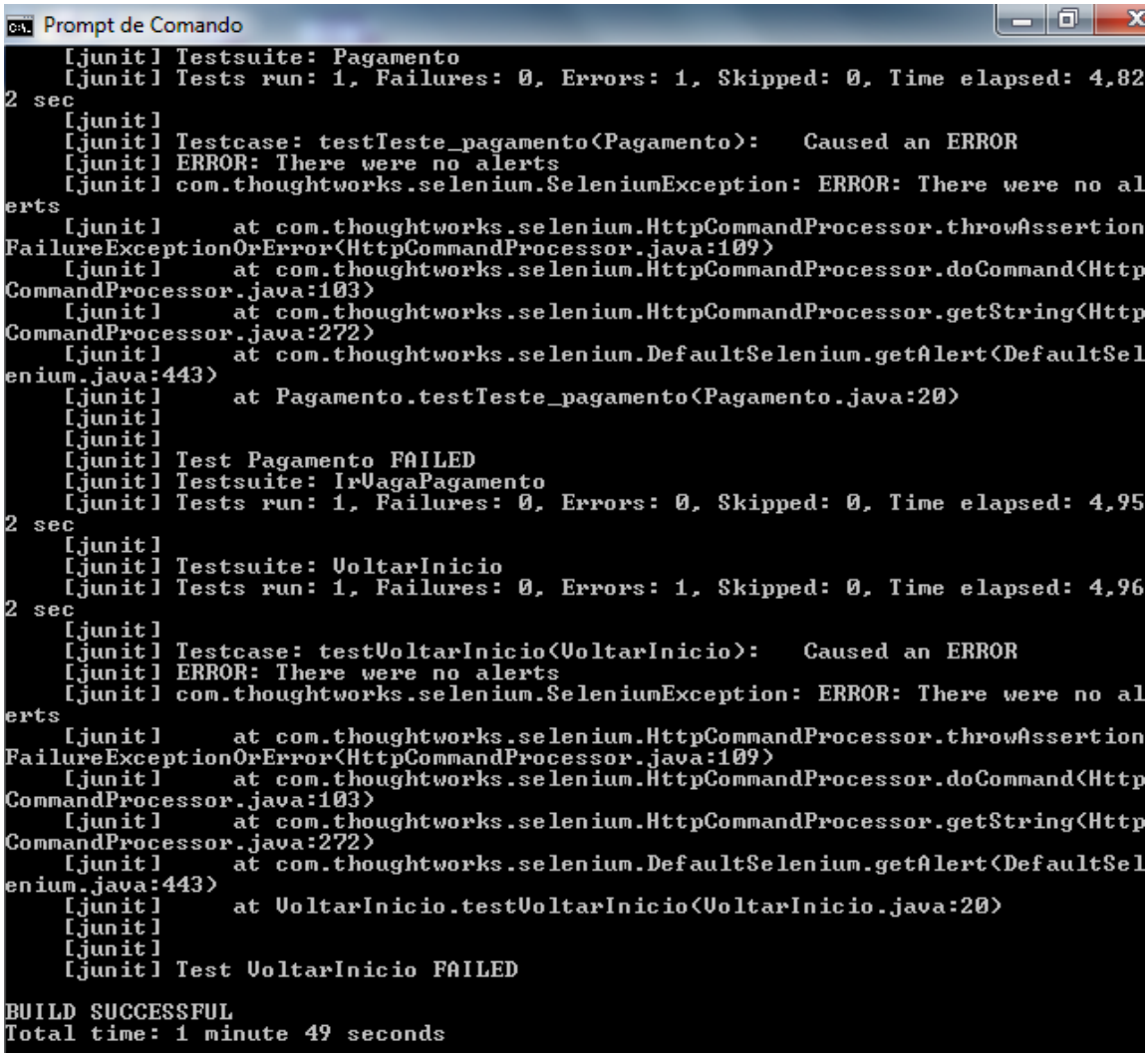


Figura 35 - Resultado dos Testes Quarto *Release* com *Ant*
Fonte: Autoria Própria

Os testes de aceitação implementados para esta aplicação *Web* mostram uma das principais vantagens do Desenvolvimento Orientado a Testes de Aceitação (ATDD), que é a identificação dos erros na aplicação e a correção dos mesmos. Isto faz com que fique mais fácil a manutenção da aplicação, reduzindo a probabilidade de introdução de erros nas próximas fases de desenvolvimento do projeto.

Além disso, os testes também possuem um papel importante no controle do código que foi desenvolvido. Isso pode ser notado caso um teste obtenha sucesso na sua primeira execução, e na sua posterior verificação, apresente falha. Uma das causas pode ser a alteração do código fonte por um dos membros de equipe de desenvolvimento sem repassá-la para os demais integrantes.

5 CONCLUSÃO

Neste trabalho abordamos testes automatizados juntamente com a prática ágil chamada Integração Contínua no desenvolvimento de uma aplicação *web*. A aplicação escolhida foi a de um controle de estacionamento, no qual o usuário pode mover os carros disponíveis na interface para uma vaga disponível no sistema, além disso, foram implementadas as funcionalidades de acesso ao sistema e de pagamento do estacionamento. Esta aplicação utilizou as linguagens *PHP* e *JavaScript* juntamente com *CSS, HTML* e banco de dados.

Na parte de testes foram utilizadas ferramentas como *Selenium IDE* para escrever os casos de testes, *Ant* para executá-los, um servidor do *Selenium* (*Selenium RC*) e um controlador de versão (*Git*) que fez a comunicação com o repositório que continha os *scripts* e códigos desenvolvidos. A cada etapa de desenvolvimento, eram gerados testes automatizados que auxiliavam na busca de erros na aplicação. Este processo visava garantir que o projeto avançasse sem erros para etapa seguinte.

No final do desenvolvimento, obteve-se uma aplicação com uma menor probabilidade de ocorrência de erros e pronta para ser disponibilizada para o usuário final.

Ao aplicar a Integração Contínua pôde-se perceber sua eficiência no desenvolvimento de um sistema por equipes, principalmente pelo fato do *feedback* instantâneo que a prática apresenta para os membros da equipe. A cada modificação do desenvolvedor no repositório há um resultado de sucesso ou insucesso proveniente da ferramenta de integração.

Foi verificado também que é importante ter pelo menos um computador dedicado à Integração Contínua, e que execute uma cópia do ambiente de produção do sistema, pois quanto mais rápido o *feedback* pelas ferramentas de integração em um projeto, melhor.

Integração Contínua resume-se em obter um *feedback* rápido e assegurar que sempre exista um *software* pronto para ser colocado em produção (BECK e ANDRES, 2004).

Tendo em vista os objetivos iniciais traçados para o desenvolvimento do trabalho, obteve-se êxito na execução da maioria deles. A única exceção se refere à

utilização da ferramenta de integração *Jenkins*, que não foi utilizado no projeto, como explicado no capítulo 3.

Com a realização deste trabalho foi possível a verificação prática da Integração Contínua juntamente com os testes automatizados, sendo estes alguns dos conceitos utilizados nas metodologias ágeis. Apesar de um desenvolvimento restrito e uma aplicação com escopo pequeno, os benefícios foram claramente notados.

A realização dos testes juntamente com o ambiente integrado proporciona ao sistema desenvolvido a rápida identificação de erros. Este fato contribui para que a equipe realize a correção imediata destes *bugs*, sempre com foco na melhoria das funcionalidades e na qualidade do *software.*

6 TRABALHOS FUTUROS

Para trabalhos futuros é sugerido utilizar a ferramenta *Jenkins* como servidor de integração, solucionando os problemas de configurações encontrados.

Outro aspecto interessante é a utilização do *Selenium WebDrive* para geração de testes com a ferramenta *Selenium IDE*. No desenvolvimento desse trabalho os testes foram implementados utilizando *Selenium RC* pelo fato de que os comandos *draggable* e *droppable* não estarem disponíveis no *Selenium WebDrive*.

A utilização da prática de desenvolvimento de *software* chamada Integração Contínua pode ser aplicada em sistemas desenvolvidos em várias linguagens. Neste trabalho foi utilizada a linguagem *PHP*. Assim, sugere-se aplicar a prática em outra linguagem de programação.

O sistema de controle de estacionamento pode conter cadastros de vagas e carros. Tais cadastros tornariam a interatividade da aplicação maior, pois o usuário definiria a quantidade de cada um dos elementos para aplicar a sua realidade. Possuir armazenado no banco de dados informações sobre os clientes também é uma sugestão. A aplicação pode aprofundar o uso de ferramentas de desenvolvimento *web* para elaborar gráficos referentes aos lucros do estacionamento.

Por fim, pode-se perceber que existe uma grande quantidade de práticas no desenvolvimento ágil, onde nesta pesquisa foi utilizada uma delas. Logo, um trabalho que envolva a prática Integração Contínua unida com outra prática ágil seria outra sugestão.

REFERÊNCIAS

BASE2. **Selenium *Web* Test.** Disponível em: <http://www.base2.com.br/tecnologias/ferramenta-automacao/selenium-*web*-test/>. Acesso em 23 out. 2014.

ABREU. Carlos B.. **Sistema de visualização gráfica para apoiar a tomada de decisão durante a fase de teste.** Disponível em: <https://www.unimep.br/phpg/bibdig/pdfs/2006/JOJJXXXGLBTX.pdf>. Acesso em: 13 ago. 2014.

ALVAREZ, Miguel Angel. **Introdução ao HTML.** 2004. Disponível em: <http://www.criarweb.com/artigos/10.php>. Acesso em: 29 out. 2014.

ALVAREZ, Miguel Angel. **Introdução à Javascript.** 2004. Disponível em: <http://www.criarweb.com/artigos/156.php>. Acesso em: 23 set. 2014.

ALVAREZ, Miguel Angel. **Introdução à programação em PHP.** 2004. Disponível em: <http://www.criarweb.com/artigos/70.php>. Acesso em: 20 set. 2014.

ALVES, Emmanuel. **O que é HTML?** 2004. Disponível em: <https://www.codigofonte.net/dicas/html/35_o-que-e-html>. Acesso em: 29 out. 2014.

ANTONINI. **Instalação de um servidor Jenkins**. Disponível em: <http://blog.endrigoantonini.com.br/2013/01/23/instalacao-de-um-servidor-jenkins/>. Acesso em: 23 out. 2014

APACHE ANT**. Apache ANT**. Disponível em <http://ant.apache.org/>. Acesso em: 16 fev. 2014.

BARTIÉ, Alexandre. **Garantia de Qualidade de Software**: as melhores práticas de engenharia de software aplicadas à sua empresa. 5.ed. Rio de Janeiro, RJ: Campus: Elsevier, 2002, 291p.

BECK, K.; ANDRES, C. **Extreme Programming Explained**: Embrace Change. 2nd. ed. [S.l.]: Addison-Wesley, 2004.

BEIZER, B.. ***Software* Testing Techniques**. 2ª ed.. Van Nostrand Reinhold, 1990.
BOWYER, J.; HUGHES, J.. **Assessing Undergraduate Experience of Continuous Integration and Test-Driven Development**. ACM New York, NY, USA, 2006.
CAELUM. **O processo de Build: ANT e Maven.** Disponível em: <http://www.caelum.com.br/apostila-java-testes-xml-design-patterns/apendice-o processo-de-build-ant-e-maven/>. Acesso em: 31 out. 2014.

COCKBURN, A. **Agile *software* development**. Boston: Addison Wesley, 2002.

CODEXICO. **Tutorial simples: Como usar o git e o github.** 2010. Disponível em: <http://codexico.com.br/blog/linux/tutorial-simples-como-usar-o-git-e-o-github/>. Acesso em: 11 out. 2014.

CODIGO FONTE. **JQuery: O que é e como usar?,** 2008. Disponível em: <https://www.codigofonte.net/dicas/javascript/310_jquery-o-que-e-e-como-usar>. Acesso em: 28 out. 2014.

COPLIEN, J. O.; HARRISON N. B. **Organizational Patterns of Agile *Software* Development**. 1. ed. Prentice Hall, 2004.

COSTA, P.. **Conceitos: testes de caixa branca e caixa preta**. Disponível em <http://crowdtest.me/teste-caixa-branca-caixa-preta/> Acesso em 23 fev. 2014.

CRAIG,R. D.; JASKIEL, S. P.. **Systematic *Software* Testing**. Artech House Publishers, Boston – London, 2002.

CRIARWEB. **O que é AJAX.** 2008. Disponível em: <http://www.criarweb.com/artigos/o-que-e-ajax.html>. Acesso em: 23 set. 2014.

DEUTSCH, M.. **“Verification and Validation”, in *Software* Engineering**. (R. Jensen e C. Tonies, eds.), Prentice-Hall, 1979.

ESCOBAR, Igor. **JQuery Mask Plugin.** Disponível em: <http://plugins.jquery.com/mask/>. Acesso em: 5 out. 2014.

EIS, Diego. **O básico: O que é HTML?** 2011. Disponível em: <http://tableless.com.br/o-que-html-basico/>. Acesso em: 29 out. 2014.

FERREIRA FILHO, José I.; SILVA, Olissea A.. **Desenvolvimento Orientado a Testes de Aceitação**. Goiânia: Pontifícia Universidade Católica (PUC-GO), 2012. 11 p.

FOWLER, M. **Continuous Integration.** 2006. Disponível em: <http://martinfowler.com/articles/continuousIntegration.html>. Acesso em: 23 fev. 2014.

GARCIA,E.G.; PENTEADO, R.,**Padrões e Métodos Ágeis: agilidade no processo de desenvolvimento de *software***,UFSC,2014.

GITHUB. Disponível em: <https://github.com/>. Acesso em: 23 fev. 2014.

GIT REFERENCE. **Basic snapshotting.** Disponível em: <http://gitref.org/basic/>. Acesso em: 11 out. 2014.

GIT-SCM. **Git Essencial - Obtendo um Repositório Git.** Disponível em: <http://git-scm.com/book/pt-br/v1/Git-Essencial-Obtendo-um-Repositório-Git>. Acesso em: 11 out. 2014.

GONÇALVES, Adriano de Oliveira. **O que é Ajax e por onde começar?** 2006. Disponível em: <http://www.revistaphp.com.br/artigo.php?id=60>. Acesso em: 23 set. 2014.

GRONER, Loiane. **Entendendo o Ext.Ajax.request (Success X Failure).** 2014. Disponível em: <http://www.loiane.com/2014/01/entendendo-o-ext-ajax-request-success-x-failure/>. Acesso em: 5 out. 2014.

GUERRA, Cauê. **Integração Contínua e o processo Agile**. Disponível em: <http://blog.caelum.com.br/integracao-continua/>. Acesso em 23 out. 2014

GUIMARÃES, Célio. **Introdução a Linguagens de Marcação: HTML, XHTML, SGML, XML.** 2005. Disponível em: <http://www.ic.unicamp.br/~celio/inf533/docs/markup.html>. Acesso em: 29 out 2014.

HTMLPROGRESSIVO**. CSS - O que é e para que serve.** Disponível em: <http://www.htmlprogressivo.net/2013/12/CSS-O-que-e-para-que-serve.html>. Acesso em: 17 set. 2014.

JENKINS. Disponível em: <http://jenkins-ci.org/content/about-jenkins-ci>. Acesso em: 13 ago. 2014.

JQUERYVALIDATION. **Documentation.** Disponível em: <http://jqueryvalidation.org/documentation/>. Acesso em: 5 out. 2014.

JQUERYUI. **Draggable.** Disponível em: <http://jqueryui.com/draggable/>. Acesso em: 5 out. 2014

JQUERYUI. **Droppable.** Disponível em: <http://jqueryui.com/droppable/>. Acesso em: 5 out. 2014.

KIOSKEA. **Javascript - Introdução à linguagem Javascript.** Disponível em: <http://pt.kioskea.net/faq/2680-javascript-introducao-a-linguagem-javascript>. Acesso em: 23 set. 2014.

KOSCIANSKI, André; SOARES, Michel dos Santos. **Qualidade de software:** aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. 2. ed. São Paulo: Novatec, 2007.

LEMON, Gez. **Div Mania,** 2011. Disponível em: <http://www.maujor.com/tutorial/divmania.php>. Acesso em: 5 out. 2014.

LEWIS E. W. ***Software*** **Testing and Continuous Quality Improvement** . Au-erbach Publications, 2000.

MAJER, Carlos A.. **Apostila Básica de AJAX.** 2008. Disponível em: <http://pt.scribd.com/doc/13536836/Apostila-Basica-de-AJAX-Carlos-Majer>. Acesso em: 27 set. 2014

MEIRELES, S., Rodrigues,R.. **Instalando o Selenium IDE.** Disponível em: <http://guiadoteste.blogspot.com.br/2012/11/instalando-o-selenium-ide.html>. Acesso em: 28 out. 2014

MENEZES, H. F. . UMA REFLEXÃO SOBRE O HTML5: COMO ESSA TECNOLOGIA TEM POSSIBILITADO A CRIAÇÃO DE PÁGINAS WEB MAIS INTERATIVAS. In: I Encontro Universitário da Universidade Federal do Cariri, 2013, Juazeiro do Norte. **Anais eletrônicos**... Juazeiro do Norte: I Encontro Universitário da Universidade Federal do Cariri, 2013. v. 1. Disponível em: <http://encontros.ufca.edu.br/index.php/eu/eu2013/paper/viewFile/2514/1053%20->. Acesso em: 13 ago. 2014.

MILANI, André. **Construindo Aplicações Web com PHP e MySQL.** 2010. Disponível em: <http://novatec.com.br/livros/phpmysql/capitulo9788575222195.pdf>. Acesso em: 20 set. 2014.

MILANI, André. **MySQL - Guia do Programador.** 2007. Disponível em: <http://novatec.com.br/livros/mysqlcompleto/capitulo8575221035.pdf>. Acesso em: 27 set. 2014.

MOREIRA. Anderson. **Maven 2.** Disponível em: <http://siep.ifpe.edu.br/anderson/blog/?page_id=1055>. Acesso em: 13 ago. 2014.

MOREIRA, Gustavo. **Testes Unitários com JUnit.** Disponível em: <http://www.dclick.com.br/2011/12/14/testes-unitarios-com-junit-de-volta-ao-basico/>. Acesso em: 29 set. 2014.

MYERS, Glenford J. **The Art of *Software* Testing.** New York: J. Wiley,1979.

MYSQL. **Automatic Initialization and Updating for TIMESTAMP.** Disponível em: <http://dev.mysql.com/doc/refman/5.0/en/timestamp-initialization.html>. Acesso em: 5 out. 2014.

NETO. Arilo C. D.. **Introdução a Teste de Software.** Disponível em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035#ixzz3Lhionz4z>. Acesso em: 13 ago. 2014.

NETO, Pedro A. S. **Introdução à Engenharia de Software.** Disponível em: <http://www.ufpi.br/subsiteFiles/pasn/arquivos/files/IntroducaoEngenhariaDeSoftware.pdf>. Acesso em: 15 set. 2014.

NIEDERAUER, Juliano. **Web Interativa com Ajax e PHP.** 2007. Disponível em: <http://www.novateceditora.com.br/livros/ajax/capitulo9788575221266.pdf>. Acesso em: 27 set. 2014.

PEREIRA, Ana Paula. **O que é CSS?** 2009. Disponível em: <http://www.tecmundo.com.br/programacao/2705-o-que-e-css-.htm>. Acesso em: 17 set. 2014.

PEREIRA, Diego V.. **Estudo da Ferramenta Selenium IDE para Testes Automatizados de Aplicações Web.** 2012. Disponível em: <http://www.espweb.uem.br/site/files/tcc/2011/Diego%20Varussa%20Pereira%20-%20Estudo%20da%20ferramenta%20Selenium%20IDE%20para%20testes%20automatizados%20de%20aplicacoes%20web.pdf>. Acesso em: 13 ago. 2014.

PHP. **O que é PHP?** Disponível em: <http://php.net/manual/pt_BR/intro-whatis.php>. Acesso em: 20 set. 2014.

PHP. **Variáveis de sessão.** Disponível em: <http://php.net/manual/pt_BR/reserved.variables.session.php>. Acesso em: 11 out. 2014.

PIMENTEL, M. P. **Junit – Implementando testes unitários em Java – Parte I.** Disponível em: <http://www.devmedia.com.br/junit-implementando-testes-unitarios-em-java-parte-i/1432>. Acesso em: 31 out. 2014.

PISA, Pedro. **O que é e como usar o MySQL?** 2012. Disponível em: <http://www.techtudo.com.br/artigos/noticia/2012/04/o-que-e-e-como-usar-o-mysql.html>. Acesso em: 27 set. 2014.

PRESSMAN, Roger S.. **Engenharia de Software.** São Paulo: Makron Books, 1995. 1056p.

PRESSMAN, R. **Engenharia de *Software*.** 2001, McGraw-Hill, 2001.

PRESSMAN, Roger S. **Engenharia de *Software* – Uma abordagem profissional**. 7ª ed. São Paulo, Bookman, 2011.

ROCHA, Fabio Gomes. **Integração contínua: uma introdução ao assunto.** Disponível em: <http://www.devmedia.com.br/integracao-continua-uma-introducao-ao-assunto/28002>. Acesso em: 23 set. 2014.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. et al..**Qualidade de software – Teoria e prática.** São Paulo: Prentice Hall, 2001.

ROSA, Everton da. **O que é o AJAX e como ele funciona.** 2009. Disponível em: <http://codigofonte.uol.com.br/artigos/o-que-e-o-ajax-e-como-ele-funciona>. Acesso em: 23 set. 2014.

SANTOS. Josiel A.. **Teste unitários em Java com JUnit.** Disponível em: <http://www.vivaolinux.com.br/artigo/Testes-unitarios-em-Java-com-JUnit>. Acesso em: 13 ago. 2014.

SANTOS, Juliana M. C.; SILVA, Edson A.; SILVA, Filipe R. V. **Avaliação de Desempenho do Impacto de Metodologias Ágeis no Desenvolvimento de *Software*.** 2008. Disponível em:
<http://www.dimap.ufrn.br/~sbmac/ermac2008/Anais/Resumos%20Estendidos/Avalia%E7%E3o%20de%20desempenho_Juliana.pdf> Acesso em: 30 set. 2014.

SCOTT,R. **Teste caixa branca. Teste de caixa preta**. Disponível em: <http://pci5.blogspot.com.br/2012/11/vs-teste-caixa-branca-teste-de-caixa.html>. Acesso em: 23 fev. 2014.

SELENIUMHQ. **What is Selenium?.** Disponível em: <http://docs.seleniumhq.org/>. Acesso em: 23 fev. 2014.

SILVA, Maurício Samy. **jQuery: a biblioteca do programador JavaScript.** São Paulo: Novatec, 2009. 430 p.

SILVA, Maurício Samy. **Introdução à jQuery.** Disponível em: <http://www.linhadecodigo.com.br/artigo/2068/introducao-a-jquery.aspx>. Acesso em: 28 out. 2014.

SILVA, Maurício Samy. **O que é PHP.** Disponível em: <http://pt-br.html.net/tutorials/php/lesson1.php>. Acesso em: 20 set. 2014.

SILVA, Maurício Samy. **JavaScript - Guia do Programador.** 2010. Disponível em: <https://www.novatec.com.br/livros/javascriptguia/capitulo9788575222485.pdf>. Acesso em: 23 set. 2014.

SOARES, M. S., **Metodologias Ágeis *Extreme Programming* e *Scrum* para o Desenvolvimento de *Software*,** 2004. Disponível em: <http://webcache.googleusercontent.com/search?q=cache:gf7i1zHJWVAJ:revistas.facecla.com.br/index.php/reinfo/article/download/146/38+&cd=1&hl=pt-BR&ct=clnk&gl=br>. Acesso em: 26 out. 2014.

SOARES M.S., Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de *Software*. **INFOCOMP**–Revista de Ciência da Computação, Lavras (MG), vol.3, n.2, p. 8-13, nov. 2004. Disponível em: <http://www.dcc.ufla.br/infocomp/artigos/v3. 2/art02.pdf>. Acesso em: 10 out. 2014.

SOUZA, E.. **Jenkins**. Disponível em: <http://pt.slideshare.net/emmanuelnerisouza/jenkins-14171428/>. Acesso em 23 fev. 2014

THE STANDISH GROUP. **THE CHAOS report.** 1995. Disponível em: <http://www.csus.edu/indiv/v/velianitis/161/ChaosReport.pdf>. Acesso em: 5 out. 2014.

W3.ORG. **What is CSS?.** Disponível em: <http://www.w3.org/Style/CSS/>. Acesso em: 5 out. 2014.

W3CBRASIL. **CSS CURSO W3C ESCRITÓRIO BRASIL.** Disponível em: <http://www.w3c.br/pub/Cursos/CursoCSS3/css-web.pdf>. Acesso em: 17 set. 2014.

W3SCHOOLS. **HTML id Attribute.** Disponível em: <http://www.w3schools.com/tags/att_global_id.asp>. Acesso em: 5 out. 2014.

WILLIAM, David. **A História Do HTML.** 2012. Disponível em: <http://www.frontendbrasil.com.br/artigos/a-historia-do-html/>. Acesso em: 29 out. 2014.