Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

FELIPE CAMPOS LINS

ESTUDO DE METODOLOGIA DE DESENVOLVIMENTO BASEADO EM TESTES APLICADO A SISTEMAS EMBARCADOS DE TEMPO REAL

Trabalho de Conclusão de Curso

Campina Grande, Paraíba
Outubro de 2016

FELIPE CAMPOS LINS

ESTUDO DE METODOLOGIA DE DESENVOLVIMENTO BASEADO EM TESTES APLICADO A SISTEMAS EMBARCADOS DE TEMPO REAL

Trabalho de Conclusão de Curso, apresentado a Universidade Federal de Campina Grande, como parte das exigências para a obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Gutemberg Gonçalves dos Santos Júnior

Campina Grande, Paraíba
Outubro de 2016

FELIPE CAMPOS LINS

ESTUDO DE METODOLOGIA DE DESENVOLVIMENTO BASEADO EM TESTES APLICADO A SISTEMAS EMBARCADOS DE TEMPO REAL

Trabalho de Conclusão de Curso, apresentado a Universidade Federal de Campina Grande, como parte das exigências para a obtenção do título de Engenheiro Eletricista.

Aprovado em: ____ de _____ de _____.

BANCA EXAMINADORA

_____
Prof. (Gutemberg Gonçalves dos Santos Júnior)
Afiliações

_____
Prof. (Nome do professor avaliador)
Afiliações

_____
Prof. (Nome do professor avaliador)
Afiliações

Campina Grande, Paraíba
Outubro de 2016

Dedico este trabalho à minha família,

com muito amor.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por todas as bênçãos que tem proporcionado na minha vida.

À minha família, por estar sempre ao meu lado e me apoiar nos momentos de maiores dificuldades.

Ao professor Gutemberg Gonçalves, pelo tempo disponibilizado e pela orientação durante o trabalho.

Aos meus amigos e colegas de curso, que me ajudaram durante todos esses anos de universidade.

À empresa Tomus Soluções, por todo aprendizado e por permitir a utilização da aplicação prática neste trabalho.

Aos meus amigos de infância, que me ajudaram a relaxar em momentos de estresse.

À minha amiga Raísa, por toda a ajuda durante a elaboração deste trabalho.

Finalmente, agradeço ao coordenador, aos professores e funcionários do curso de graduação de Engenharia Elétrica, os quais contribuíram com minha formação profissional.

"Não importa o quão devagar você vá,
desde que você não pare."

Confucio

RESUMO

Neste trabalho foi realizado um estudo sobre técnicas de desenvolvimento dirigidos a testes aplicados aos sistemas embarcados de tempo real, onde é necessário o cumprimento de um tempo pré-determinado durante a resposta do sistema aos estímulos externos. Após a abordagem de toda as definições teóricas necessárias para o leitor se familiarizar com o tema, descreveu-se todas as informações necessárias para um entendimento de como esses testes devem ser executados, mostrando os passos do desenvolvimento do código na linguagem em C e por fim mostrando um caso prático, com foco em um equipamento de telemetria, realizado pelo desenvolvedor desse projeto na empresa que o mesmo atua.

Palavras Chave: Desenvolvimento Dirigido por Testes, Sistemas Embarcados, Sistemas de Tempo Real.

ABSTRACT

In this project was studied Test Driven Development technics targeting Real Time Embedded Systems, where the time specification, for a external stimulus response, must be fulfilled. After an approach of all the theoretical definitions needed by the reader to be familiar with the subject, it was described all the information for an understanding of how the tests run, explaining the development steps of code written in C language, and applying it in a practical example, focused on a telemetry equipment.

Key words: Test-driven Development, Embedded systems, Real time systems.

SUMÁRIO

ÍNDICE DE FIGURAS

1. Introdução

A incorporação de sistemas eletrônicos nos mais diversos produtos é um dos maiores motivos para o crescimento da indústria eletrônica [1]. Alguns desses produtos, como automóveis e equipamentos de comunicação pessoal, possuem restrições temporais no processamento de seus dados que, caso não sejam cumpridas, podem representar perdas em termos financeiro, ambiental ou, em alguns casos, humano, tornando-se necessário uma maior preocupação, por parte do desenvolvedor, em entregar códigos com a menor quantidade de erros possíveis [2].

Para evitar ao máximo o comportamento indesejado de códigos foi proposto o desenvolvimento baseado em testes (Test Driven Development - TDD). [3] O qual consiste em criar um teste que validará a funcionalidade desejada antes de qualquer mudança no código. Desta forma, é possível verificar, de maneira segura e independente do hardware, se existe algum erro de implementação no novo código desenvolvido, permitindo um menor escopo para a procura do erro, resultando em uma identificação e correção mais rápida do mesmo, se comparado com casos em que o código é feito por completo e testado no final.

Na realização de testes unitários, cada componente é testada individualmente, e no final da criação do código realiza-se o teste com todas as componentes de forma integrada para certificar-se de que a integração entre elas está funcionando corretamente. Faz-se necessário essa certificação para evitar possíveis erros de montagem.

O desenvolvedor deve estar atento as especificações do sistema para implementar os primeiros testes, por fim é preciso criar um teste que valide todas as componentes, para o TDD fornecer cobertura completa aos testes e funcionar como desejado.

Devido tais fatos, este trabalho pretende estudar uma metodologia para aplicação do TDD em sistemas embarcados em tempo real, afim de permitir uma codificação eficiente e segura, de modo a garantir a funcionalidade do sistema a ser desenvolvido.

2. Objetivos

2.1. Objetivo Geral

O objetivo geral deste trabalho é o estudo de uma metodologia de desenvolvimento dirigido por testes (TDD) para sistemas embarcados de tempo real, de modo a mitigar comportamentos indesejados e permitir maior segurança ao desenvolvedor.

2.2. Objetivo Específico

O objetivo específico deste projeto é desenvolver um módulo para um equipamento de monitoramento remoto, a fim de obter informações de telemetria. Essas informações são passadas via mensagens para a empresa responsável pelo monitoramento, entretanto, para que o módulo gerencie as mensagens de maneira eficaz, foi utilizada a técnica TDD.

3. Desenvolvimento dirigido por testes (TDD)

O desenvolvimento dirigido por testes (TDD) consiste em escrever testes unitários para o código que está sendo implementado antes de qualquer linha de código da aplicação. Portanto, é uma técnica de desenvolvimento incremental que força o desenvolvedor a executar pequenos passos em direção ao que foi especificado para o módulo.

O fluxo básico para o desenvolvimento consiste em escrever um teste unitário que irá falhar, desenvolver o código para passar no teste, repetir com testes que incrementem a funcionalidade, sem falhar os testes anteriores. Os passos principais para este desenvolvimento são citados abaixo:

1. Adicionar um pequeno teste;
2. Realizar todos os testes e ver o novo teste falhar;
3. Modificar o código de forma a passar nos testes;
4. Repetir 3 até todos os testes passarem;
5. Reestruturar para remover código duplicado e melhorar entendimento;

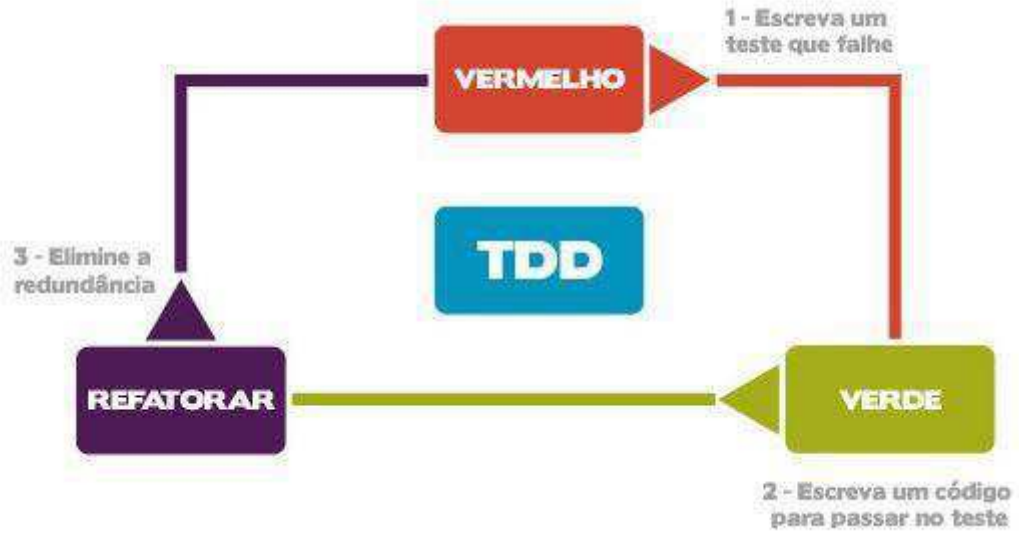

Figura 1-Ciclo de Desenvolvimento com TDD [7]

Para que esse fluxo seja eficiente, é necessário que os testes sejam automáticos e que representem uma funcionalidade nova, ou comportamento desejado. Quando os testes são manuais torna-se desgastante a repetição dos testes para qualquer mudança no código, o que pode ocorrer com relativa frequência, fazendo com que apenas testes considerados essenciais sejam refeitos, evitando verificar se a mudança realizada afetou funcionalidades que estavam corretas.

Quando aplicado corretamente o TDD permite que o desenvolvedor lide com um problema por vez e encontre erros com mais facilidade. Em técnicas tradicionais de desenvolvimento o código é completamente escrito, para depois ser testado, tornando complicado identificar quais partes possuem erro e qual a causa do erro para ser possível corrigi-lo.

Deve-se ter em mente que erros no sistema trazem prejuízos para o usuário final e para a empresa que o desenvolveu, pois ela irá perder a credibilidade com os clientes. Além do mais, o custo para correção de erros ocorre de forma crescente, se encontrado na fase de especificação terá um custo médio X, caso seja encontrado no desenvolvimento custará um valor em média igual a 35X e se encontrado após a implantação da aplicação, esse valor sobe para em média 70X[18]. Pode-se

observar que há uma discrepância gigante entre os valores e é extremamente importante para a empresa identificar o erro antes da implantação da aplicação.

Quando os testes são implementados antes e realizados para cada pequena mudança do código o escopo de procura do erro se reduz para esta mudança, tornando a procura pela causa do erro mais rápida, evitando a propagação do mesmo.

3.1. Vantagens ao utilizar o TDD

- Menor tempo na procura da causa de erros;
- Correção de erro mais rápida diminuindo o custo de propagação do erro;
- Documentação das funcionalidades do módulo pelos testes;
- Noção do que está funcionando e o que falta ser realizado para atingir todas as especificações;
- Design e implementação de código testável;
- Qualidade do produto de software.

3.2. Desvantagens ao utilizar o TDD

- Dificuldade em criar os testes antes do código
- Requer disponibilidade de tempo do programador
- Mais trabalhoso que testar apenas o resultado final
- Possibilidade de erro nas definições de testes

3.3. Benefícios pelo uso de testes automáticos e independentes de plataforma

Códigos embarcados possuem, em geral, a limitação de depender do hardware. Muitas vezes o hardware não está pronto ainda ou a reprodução de um cenário pode quebrar o hardware, portanto testes automáticos e independentes de plataforma possuem alguns benefícios, como:

- Possibilita testes antes de o hardware estar disponível;
- Reduz o número de ciclos onde o código é compilado, linkado e gravado, ciclos em geral demorados;

- Reduz tempo de *debug* no hardware, onde é mais complicado chegar no cenário de teste
- Isola problemas de interação entre hardware e software por modelar as interações com o hardware nos testes
- Melhora o design de forma a pensar em um melhor desacoplamento entre módulos e hardware.

4. Tipos de Testes em Software

Os testes são de suma importância para o desenvolvimento do software, pois eles que irão auxiliar o programador a encontrar e corrigir as falhas do código, e em seguida a validação do projeto após verificar que o sistema está atendendo aos padrões estabelecidos. Na literatura está descrito uma infinidade de tipos de testes para softwares, no entanto, de forma a manter o escopo do trabalho, apenas alguns deles serão abordados. A princípio, iremos classifica-lo em função do objeto de teste e em função do estágio do ciclo de vida do produto.

4.1. Testes em função do objeto de teste

Sistemas complexos são formados por diversos métodos e estruturas. Portanto, é necessário serem submetidos a testes que sejam condizentes com a sua composição. Abaixo pode-se observar os principais exemplos de testes:

4.1.1. Teste Funcional

Esse teste é caracterizado por considerar o comportamento externo do software, derivam as condições de testes e casos de testes de acordo com a funcionalidade e especificação do projeto, devendo ser aplicado a todas as componentes do sistema. Pode também ser chamado de Teste Caixa Preta, por não considerar a parte interna do sistema, trabalhando basicamente a partir da escolha de dados de entrada, aplicando o teste nos mesmos e observando se seus resultados são compatíveis com resultados previamente conhecidos.

4.1.2. Teste Não Funcional

Esse tipo de teste não está ligado a funcionalidade do sistema, servem para verificação de atributos do sistema, alguns exemplos são: a segurança, portabilidade, confiabilidade, usabilidade, entre outros. Podem ser aplicados em todos os níveis de testes.

4.1.3. Teste Estrutural

Nesse tipo de teste, deve-se analisar o comportamento interno do sistema, por isso também é chamado de Teste Caixa Branca. A aplicação do teste consiste em observar o código fonte e construir casos de testes que contemplem todas as possibilidades dos componentes de software. Já que possui acesso ao código fonte, pode-se construir ligações entre as componentes e bibliotecas. Assim como os testes descritos anteriormente, também pode ser aplicado a todos os níveis de testes, porém recomenda-se realizar essa técnica posteriormente as técnicas com base em especificações do projeto.

4.1.4. Testes de Regressão

Esse teste pode ser considerado um reteste, que é realizado com o objetivo de observar se as componentes do sistema, já testadas, não estão falhando nas versões atualizadas do projeto devido a alguma modificação recente. Portanto, toda vez que houver alguma modificação, todo o sistema deve ser testado novamente.

4.2. Testes em função do estágio do ciclo de vida do produto

Durante o desenvolvimento do software, deve-se aplicar testes para observar se a execução está ocorrendo como prevista. É o princípio utilizado pelo Desenvolvimento dirigido por testes (TDD), onde tem-se para cada etapa do desenvolvimento do código, uma aplicação de teste. Esses testes podem ser classificados em cinco tipos, sendo os três primeiros os mais usuais para o TDD, são eles:

I. Teste Unitário
II. Teste de Integração

III. Teste de Interface
IV. Teste de Aceitação
V. Teste de Manutenção

Como os três primeiros são essenciais para o TDD, que é um assunto bastante abordado nesse trabalho, a descrição deles será no próximo tópico, este tópico irá se restringir apenas a definição dos dois últimos testes citados acima.

O Teste de Aceitação não prioriza encontrar falhas no sistema, usualmente é realizado pelo usuário do sistema a fim de obter confiabilidade em uma componente do sistema ou no sistema por completo. É executado no ambiente de homologação e anteriormente a implantação do software.

O Teste de Manutenção deve ser aplicado no sistema após anos de uso do mesmo, para certificar-se de que o sistema continua funcionando corretamente. Por exemplo, em caso de mudanças de ambiente durante esse tempo, observa-se se não houve interferência do novo ambiente no sistema.

4.3. Testes para o TDD

Os testes realizados pelo TDD têm por objetivo evitar e eliminar os erros no código do projeto, sendo classificados basicamente em três tipos: testes unitários, testes de integração e testes de interface. Os testes unitários formam a base para o TDD, porque esse é o primeiro teste que deve ser realizado com pequenos trechos do código para se certificar de que eles estão respondendo de acordo com as expectativas do programador. Após os trechos dos códigos terem funcionado corretamente no teste unitário, deve-se submetê-los aos testes de integração, unindo pelo menos duas partes do sistema e observando seu funcionamento. Depois de observar resultados satisfatórios nos testes de integração, deve-se realizar o teste de interface, que dará segurança ao programador de que todo o sistema está funcionando como esperado. Esses testes serão descritos a seguir:

4.3.1. Teste Unitário (Unit Test)

Esse teste é caracterizado por realizar testes em cada trecho do código, individualmente, analisando o funcionamento do mesmo, identificando e solucionando possíveis falhas de maneira eficaz, já que esse teste é de rápida execução.

4.3.2. Teste de Integração (Integration Test)

Nesse teste é feita a combinação entre trechos do código para observar o funcionamento deles em conjunto, facilitando o reparo de falhas. Tendo em vista que os códigos unitários estão corretos, qualquer falha será devido ao agrupamento desses códigos. Após obter os resultados esperados nesse teste, o código já estará preparado para o teste de interface.

4.3.3. Teste de Interface (Interface Test)

Esse é o teste final que será aplicado ao código por completo, após ter sido submetido aos testes acima, verifica-se se o sistema está funcionando corretamente junto a interface gráfica, para caso precise, fazer ajustes nessa combinação.

5. Sistemas de Tempo Real

Sistemas de Tempo Real são sistemas operacionais que devem responder aos estímulos do ambiente em que estão inseridos, a fim de executarem tarefas em um prazo pré-definido. Portanto, neste tipo de Sistema em especial, além do desenvolvimento correto do código, também é essencial a entrega do resultado no prazo específico para cada tipo de evento, pois caso ultrapasse o tempo estipulado o resultado final pode ser inútil. Esses sistemas computacionais são capazes de realizar múltiplas atividades, que tem ordem de prioridade e sincronismo entre elas. À medida que as atividades prioritárias precisarem ser executadas, elas começarão a controlar o processador e interromperão temporariamente as demais atividades. Esse modo de gerenciamento é chamado Preemptivo. Existe também outro modo de gerenciamento chamado Cooperativo, no qual o programador define o momento em que cada atividade irá assumir o controle do processador, além do momento em que a atividade liberará o controle do processador para que outra atividade seja executada.

As atividades são classificadas, de acordo com o intervalo de tempo de lançamento das suas instâncias, em periódicas, esporádicas e aperiódicas. As atividades periódicas têm um intervalo de tempo regular para o lançamento de cada uma de suas instâncias, enquanto as chamadas esporádicas são aquelas que se conhece apenas um intervalo mínimo de tempo entre os lançamentos de instâncias consecutivas. Por fim as aperiódicas são aquelas atividades que não se conhece a priori nenhuma informação em relação a frequência de lançamento das instâncias.

O Sistema de Tempo Real deve atuar no Sistema a Controlar, no Sistema Computacional de Controle e nas interfaces de entrada e saída. A interface de entrada liga o Sistema a controlar ao Sistema Computacional de controle e a interface de saída liga o Sistema Computacional de controle ao Operador. O Sistema a Controlar e o Operador serão os ambientes do sistema computacional, enquanto o Sistema Computacional de Controle se responsabilizará em responder aos estímulos recebidos dentro do prazo definido.

Para a eficiência desse sistema, deverá existir uma previsão do seu comportamento funcional e temporal, de modo que exista uma exatidão dos resultados, pois caso ocorra alguma falha na execução do código o sistema consequentemente não irá responder em tempo hábil. A previsibilidade do comportamento do sistema é realizada a partir de hipóteses de carga e hipóteses de falha. Essas hipóteses preveem respectivamente, a carga máxima gerada pelo ambiente e os tipos e frequências de falhas que podem ocorrer durante a execução do sistema.

Além desses dois fatores citados acima, existem outros fatores que interferem na previsibilidade do sistema, alguns códigos escritos na linguagem de programação não são previsíveis, cabe ao programador evitar este tipo de linguagem e utilizar laços limitados ao invés de laços ilimitados que não são previsíveis. O hardware também possui algumas fontes que não podem ser previstas e que, se possível, o programador deve substituí-las por outras fontes que possam ser determinadas. Existem casos em que a carga gerada pelo ambiente não pode ser determinada de forma prévia, nesses casos deve-se recorrer a previsibilidade probabilística, pois ela estipulará, a partir de estimativas e simulações, prováveis prazos a serem cumpridos.

Diante do exposto acima, percebe-se que a previsibilidade é uma característica enfatizada do RTS (Real Time System), pois é uma das principais diferenças entre esse sistema e os sistemas convencionais. Pode-se observar também que a divisão da aplicação em várias atividades, que serão gerenciadas pelo núcleo do sistema, otimiza o desenvolvimento do sistema e por conseguinte o tempo de execução.

Os Sistemas de Tempo Real podem ser classificados basicamente em Sistemas Não Críticos de Tempo Real (Soft Real Time System), em que a média do tempo de execução deve ser mantida porém a falha temporal não anula os benefícios do sistema, e Sistemas Críticos de Tempo Real (Hard Real Time System), em que há especificação de temporização e caso haja qualquer falha temporal, ela irá se sobressair em relação aos benefícios do sistema.

Para os Sistemas Críticos de Tempo Real, observa-se duas classes relevantes: (1) os Sistemas de Tempo Real Crítico Seguros em Caso de Falha, que apesar dos limites de tempo para execução, pode atingir um estado seguro em caso de falha no tempo; e (2) os Sistemas de Tempo Real Crítico Operacionais em Caso de Falha, que tem limite temporal para execução e caso ocorra falhas parciais haverá um dano irreparável ao sistema.

As abordagens para problemas em tempo real podem ser classificadas em síncronas e assíncronas. Na abordagem síncrona, existe uma análise cronológica dos eventos, partindo da hipótese que existe simultaneidade entre esses eventos, isto é, que as respostas do sistema ocorrem de forma instantânea. Essa hipótese facilita a modelagem e a verificação formal das propriedades do sistema, porém não leva em consideração o tempo de execução de instruções, cálculos e a possibilidade de interferência do ambiente externo no sistema. Nesse tipo de sistema trabalha-se com sistemas reativos e pressupõem-se que são deterministas, isto é, que a sequência de entradas e saídas serão sempre iguais. Enquanto a abordagem assíncrona, analisa eventos independentes, que não são simultâneos. Essa abordagem possui uma exatidão maior que a abordagem síncrona, por levar em consideração as ferramentas de implementação do sistema. Consequentemente, há uma complexidade maior para o programador, já que ele precisa fazer uma interação do software com o hardware para obter uma visão mais completa do sistema.

6. Exemplos de Sistema de Tempo Real:

6.1. Sistema de radar online que monitora voos:

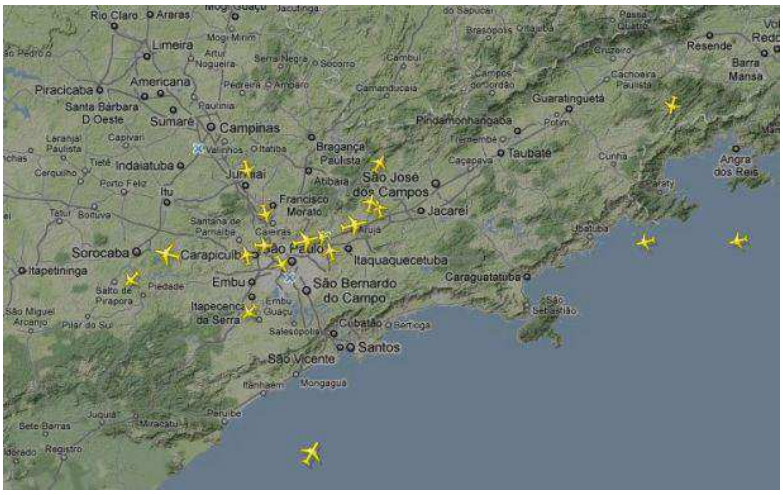

Figura 2-Voos rastreados no Brasil pelo Sistema Flightradar24 [15]

Este sistema de radar é um Sistema de Tempo Real Crítico, pois é necessária a obtenção de resultados para o código em um prazo de tempo pré-estipulado. Caso o avião saia da rota traçada para ele em um voo com piloto automático, é preciso corrigir imediatamente a rota para que não ocorra uma tragédia.

Esse monitoramento utilizando STR, além de ser usado pelas empresas aéreas, que rastreiam os voos para evitar colisões entre as aeronaves, permite também aos usuários desse meio de transporte acompanhar o trajeto do avião pela internet, e tendo acesso inclusive ao horário no qual ele pousará no destino final. No Brasil, a Infraero (Empresa Brasileira de Infraestrutura Aeroportuária), que é responsável pela administração dos principais aeroportos do Brasil, lançou um aplicativo na rede social Facebook, que permite o rastreamento do voo pelo celular.

O Sistema Flightradar24 é um sistema de monitoramento de voos, que é utilizado em mais da metade das aeronaves em âmbito mundial. Na Europa, ele é bastante utilizado chegando a ser responsável pelo monitoramento de cerca de 90% das aeronaves do continente, como se pode observar na imagem abaixo:

Figura 3-Voos rastreados na Europa pelo Sistema Flightradar24 [15]

6.2. Equipamentos Médicos:

Figura 4 - Monitor Cardíaco Hospitalar [16]

Quando na UTI o paciente sofre uma variação nos seus batimentos cardíacos, o monitor cardíaco desse paciente deve alarmar em poucos segundos para alertar a equipe de saúde o ocorrido. Para atender essas expectativas, é necessário a

utilização de um Sistema de Tempo Real Crítico, pois caso haja falha temporal o paciente pode sofrer danos irreparáveis.

6.3. Sistema de Orientação de mísseis:

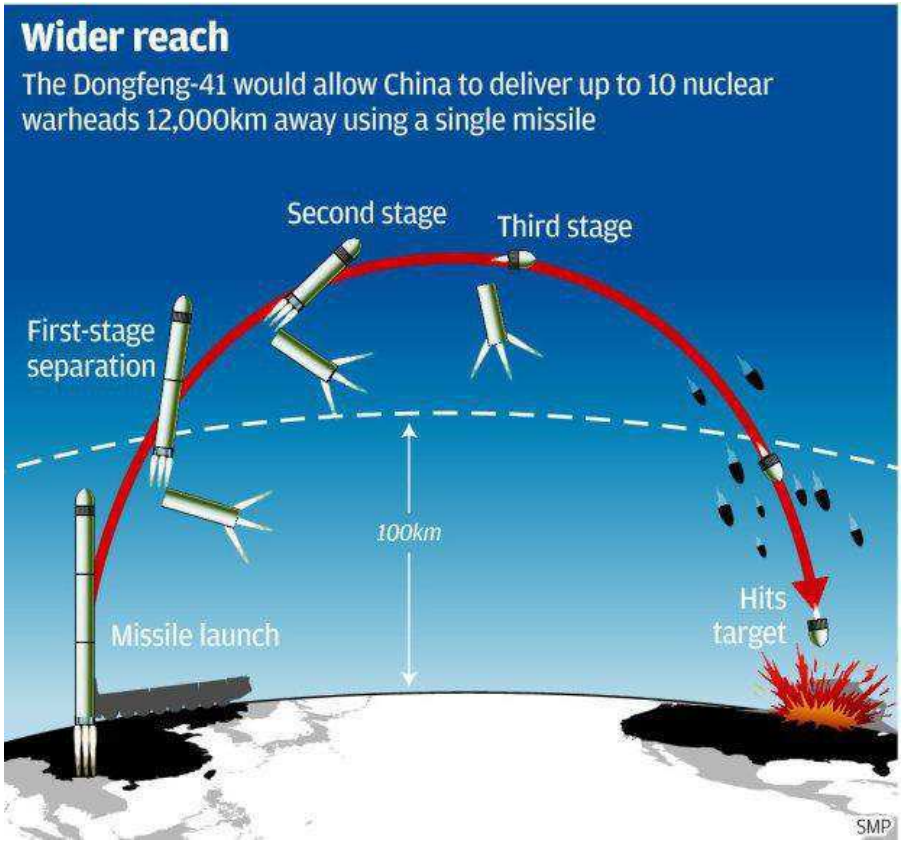

Figura 5-Maior alcance: o sistema Dongfeng-41 permitiria à China entregar até 10 ogivas nucleares a 12.000 quilômetros de distância usando um único míssil. [14]

O Sistema de Orientação e Lançamento de mísseis se caracteriza por ser um Sistema de Tempo Real Crítico, onde é necessária a resposta imediata à ataques de pequena e grande extensão. Países que se enfrentam durante uma guerra, precisam ter equipamentos informatizados que lhe tragam segurança e sejam capazes de realizar o contra-ataque com sucesso. Para as forças armadas a eficiência do STR é essencial para conseguirem pôr em prática os ataques planejados, e em casos de falha temporal do STR, um grande número de pessoas podem morrer.

6.4. Semáforos em Tempo Real:

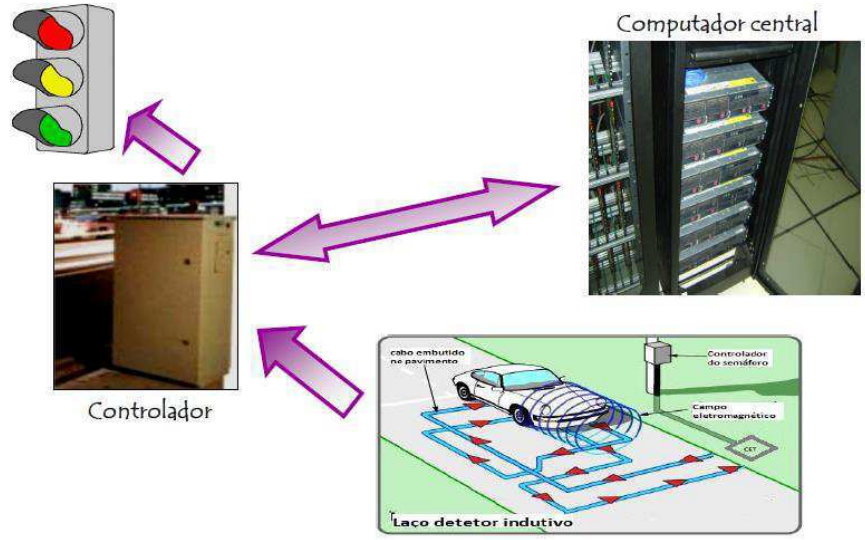

Figura 6 – Esquema da Operação de Semáforos em Tempo Real [17]

Com o avanço da tecnologia, teve-se um aumento considerável no número de veículos transitando pelas estradas e consequentemente um aumento na quantidade de engarrafamentos em vias movimentadas. O uso do semáforo em cruzamentos, facilita o acesso as vias principais e a passagem por essas vias.

Para que o semáforo atenda a demanda diária desejada seria ideal que o mesmo soubesse os momentos críticos do trânsito, não apenas aqueles conhecidos como horários de pico, mas também em casos eventuais que o trânsito esteja congestionado por outro motivo que não seja os horários de pico. O STR é capaz de resolver esse problema, identificando através de câmeras instaladas junto aos semáforos a existência ou não de engarrafamentos. Após receber essa informação, o controlador envia o dado de contagem ao computador, que processa a informação e informa o resultado ao controlador, alterando o tempo do sinal para aquele momento específico, acelerando o processo de descongestionamento da via.

Esse Sistema é considerado um Sistema de Tempo Real Não Crítico, pois apesar de haver exigência de tempo para execução das tarefas pelo sistema, uma falha temporal não irá provocar danos irreversíveis e os benefícios do sistema ainda irão se sobressair sobre a falha ocorrida.

7. Isolando os módulos

Em sistemas muito grandes ou complexos, o código em desenvolvimento pode possuir dependências. Tais dependências possuem a capacidade de limitar o escopo dos testes, pois seria necessário que todas elas estivessem funcionando antes do teste ser realizado. No entanto, algumas delas podem ainda estar sendo desenvolvidas por outra equipe, ou serem dependentes do hardware que não está pronto. Mesmo quando todas as dependências estejam prontas, como identificar se o erro está no código ou em suas dependências?

Outros problemas podem surgir quando é difícil controlar o comportamento de alguma dependência. Para verificar se o código sendo testado (code under test - CUT) consegue lidar com respostas errôneas de alguma dependência, com testes automáticos, é necessário que o comportamento desta seja passível de previsibilidade, ou seja, que a dependência consiga sempre ser colocada em um cenário de falha quando o teste for chamado, no entanto, nem sempre é possível garantir tal comportamento, já que muitas vezes dependências também possuem suas próprias dependências, tornando necessário controla-las por completo.

Isto posto, torna-se necessário isolar o código que está sendo desenvolvido de forma a evitar alguns desses problemas. Para isso decisões de design devem ser tomadas com essa finalidade, tentando possuir um uso mais rigoroso de interfaces, encapsulação, proteção de dados e menor utilização de variáveis globais desprotegidas.

Se a interação entre módulos for feita exclusivamente por interfaces, ou APIs, é possível substituir a implementação real por uma versão de testes, ou falsa, de forma que o CUT não consegue perceber a diferença de um para o outro.

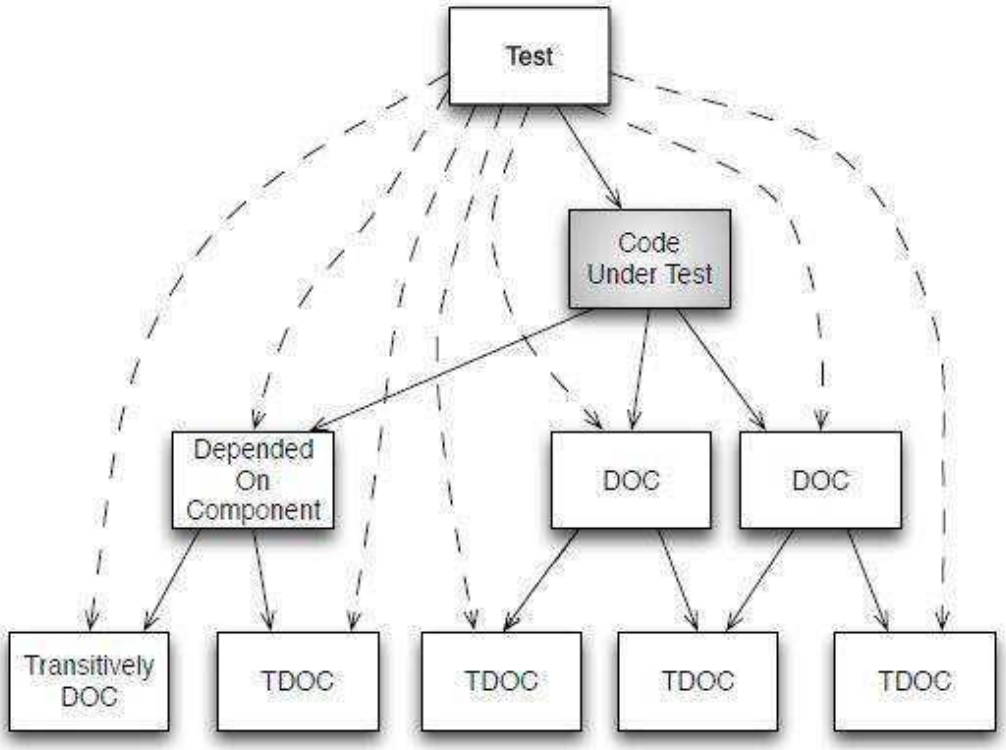

Figura 7- Diagrama sem quebra de dependência [3]

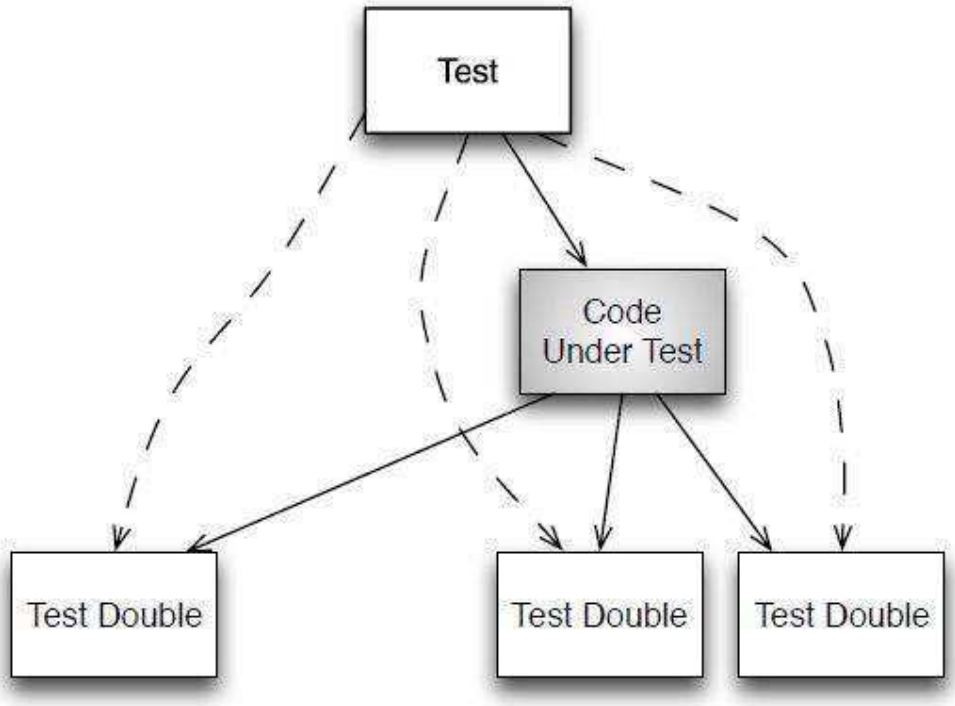

Figura 8-Diagrama com quebra de dependência [3]

Tais versões não são simulações das dependências, apenas versões simplificadas, cujo comportamento pode ser facilmente manipulado. Desta forma, podem ser utilizados para fornecer entradas indiretas ao CUT, por retorno ou ponteiros, direcionando o CUT para um cenário específico, ou para capturar saídas indiretas, observando os parâmetros passados, e verificando o comportamento.

Nem sempre será preciso fornecer uma versão de teste das dependências, por exemplo: se a dependência simplesmente calcula a diferença entre dois horários em segundos, e já foi validado, não há motivos para quebrar tal dependência. Segundo Grenning [3] normalmente deve ser fornecido versões de teste quando:

- Existe dependência com Hardware;
- Há dificuldade em injetar entradas produzidas;
- É necessário acelerar uma dependência lenta;
- Existe dependência de algo volátil;
- Existe dependência de algo ainda em desenvolvimento;
- Existe dependência de algo que é difícil de configurar.

7.1. Estruturas utilizadas:

Meszaros [4] lista as estruturas típicas que podem substituir as dependências de um código. São elas:

7.1.1. Test Stubs:

Um stub é uma implementação de uma interface específica para um cenário de teste. É configurada para responder as chamadas do CUT com valores (ou exceções) que irão exercitá-lo, de acordo com o teste que está sendo feito, no lugar da implementação real. Em geral, stubs são cascas que retornam valores definidos antes do CUT ser chamado.

Tal estrutura pode ser utilizada como ponto de controle permitindo direcionar o comportamento com várias entradas indiretas. Também pode ser utilizada para encaminhar o CUT a um ponto em que chame algum código indisponível ao ambiente de teste.

7.1.2. Test Spy:

Um spy é uma forma simples de verificar o comportamento de um código. Antes de exercitá-lo é feito a implementação de uma interface específica que irá funcionar como um ponto de observação, gravando as chamadas feitas pelo CUT durante sua execução. Na fase de verificação dos testes é possível usar os resultados gravados e compará-los com resultados esperados.

Tal estrutura normalmente é utilizada quando estamos verificando as saídas indiretas do CUT e não é possível prever os valores de todos os parâmetros, quando a forma com o qual os Objetos Mock fazem a verificação de expectativa não é suficientemente explicito, quando testes necessitam de comparações de igualdade que fogem do padrão, quando é preferível ter acesso a todas as chamadas que o CUT faz antes de realizar qualquer assertiva, ou quando assertivas que falharam não conseguem gerar um diagnóstico preciso de onde a falha ocorreu.

7.1.3. Objeto Mock:

Um objeto mock é definido com a mesma interface que um objeto em que o CUT depende. Durante o teste, o mock é configurado com os valores que deverá responder, cada vez que for chamado, e os valores que espera receber quando for chamado. Quando chamado, o Mock compara os argumentos recebidos com os argumentos esperados, utilizando assertivas de igualdade, e falha o teste caso alguma comparação não seja positiva.

O objeto mock pode ser utilizado quando é necessário descrever um comportamento que é esperado para o CUT. Para utilizá-lo, no entanto, é necessário ser possível predizer os valores de todos, ou quase todos, os parâmetros das chamadas de métodos antes de executar o código, que está sendo testado.

Uma das vantagens de utilizar mocks é a capacidade de descrever uma sequência de eventos, podendo simular um cenário dinâmico, que depende de uma sequência bem definida de interações. Outra vantagem é a existência de ferramentas que permitem utilizar a estrutura de forma simples e direta, funcionando

quase como uma lista de chamadas que serão automaticamente verificadas, sem a necessidade de escrever uma grande quantidade de linhas de código para o teste.

Para ter certeza que todas as expectativas foram chamadas é necessário avisar ao objeto mock que o teste terminou, dessa forma ele será capaz de verificar se existe alguma chamada que não foi realizada, esse aviso depende do toolkit utilizado, alguns instalam automaticamente um método no final do teste, outros devem ser colocados manualmente.

7.1.4. Objeto Fake:

Um objeto fake é uma implementação mais simples da funcionalidade que uma dependência realiza. Esta implementação é então utilizada no lugar da dependência real. Tal implementação não precisa ser tão robusta ou ter a mesma capacidade, basta prover um serviço equivalente ao original.

Diferentemente do Test Stub, o Fake não é utilizado como um ponto de controle, mas sim como um meio para manter uma interação autossuficiente e consistente. Outra diferença entre o fake com o stub, e com mocks, é que no primeiro os parâmetros e resultados são modificados em tempo de execução, enquanto os outros possuem valores fixados pelo teste.

Os métodos fakes podem ser configurados indiretamente, utilizando métodos auxiliares que irão manipular variáveis internas. Essa configuração se torna útil em cenários que as dependências precisam ser previamente inicializadas, além de auxiliar no direcionamento do código a ser testado.

Objetos fakes normalmente são utilizados quando o CUT depende de componentes, que não estão disponíveis ou que deixam testes complicados ou lentos, e os testes necessitam de uma sequência de comportamentos mais complexos do que vale a pena em fazer por meio de stubs ou mocks.

7.2. Substituição em C

Existem alguns mecanismos de substituição em C que podem ser empregados para quebrar as dependências.

7.2.1. Em tempo de ligação

Pode ser realizada a troca em tempo de ligação quando se deseja substituir todo o módulo de dependência por uma versão executável de teste. Também, para substituir módulos que o desenvolvedor não tem controle sobre a interface.

Essa técnica se torna útil para testes fora do target e para eliminar dependência com bibliotecas de terceiros, módulos dependentes do hardware, ou sistema operacional.

7.2.2. Ponteiro de função

Pode ser utilizado essa técnica quando é necessário substituir as dependências em apenas alguns casos de teste. É possível realizar a substituição por ponteiro em qualquer lugar que se tenha controle sobre a interface. No entanto, é mais complicado, utiliza mais memória RAM e compromete a clareza de leitura das declarações de função.

Permite grande controle sobre quais funções serão sobrescritas e quais não serão.

7.2.3. Pré-processador

Substituições durante pré-processamento são utilizadas quando as técnicas descritas anteriormente não possuem capacidade suficiente para realizar o trabalho.

É possível quebrar uma cadeia de includes indesejáveis com o pré-processador. Também, para selecionar, ou temporariamente sobrescrever nomes. No entanto, deve ser utilizado em último caso, dado que o código compilado no CUT será diferente, permeando mudanças no código possivelmente indesejadas.

Quando tentado a usar diretivas de pré-processamento, pode-se considerar a alternativa de embalar o código e prover uma nova interface da qual se tem completo controle e permite a utilização de ponteiros de função ou substituição em tempo de ligação.

7.2.4. Combinar substituição em tempo de ligação com ponteiro de funções

É possível fazer a substituição em tempo de ligação trabalhar em conjunto com a utilização de ponteiros de funções. Basta que um stub, substituído em tempo de ligação, contenha um ponteiro de função. Inicialmente o ponteiro será inicializado com NULL.

Neste caso, o stub terá um comportamento padrão de não fazer nada. Mas, algum caso de teste terá a flexibilidade de sobrescrever o ponteiro para realizar o comportamento desejado para o teste. Isto se torna útil quando a flexibilidade dos ponteiros de função é desejada, sem modificar as interfaces das dependências.

8. Framework para teste unitários

Um framework para testes unitários é um pacote de software, que permite ao programador expressar como o código de produção deve se comportar. Este pacote deve fornecer algumas capacidades básicas, que são:

- Uma linguagem comum para expressar casos de teste
- Uma linguagem comum para expressar resultados esperados
- Acesso às características da linguagem de programação do código de produção
- Um lugar para coletar todos os casos de teste para o projeto
- Um mecanismo para rodar os casos de teste, em grupo ou todos de uma vez
- Um relatório conciso de falhas e sucessos
- Um relatório detalhado para qualquer falha de teste

8.1. CppUTest

CppUTest é um framework desenvolvido para suportar múltiplas plataformas de Sistema Operacional com o objetivo de possibilitar o desenvolvimento de códigos embarcados. Embora seja escrito em C++, suas macros permitem uma utilização consideravelmente simples para usuários com conhecimento apenas de C.

Está framework é open souce e pode ser baixado gratuitamente em http://www.cpputest.org[5].

Para criar arquivos de testes deve ser definido um grupo de teste junto com seus casos de teste. Cada grupo de teste possui um método de setup() e outro de teardown(). Desta forma, todos os testes vinculados a um determinado grupo chamará, antes de iniciar, o método de setup, e chamará o teardown quando estiver finalizando.

O método de setup pode ser utilizado para garantir uma inicialização comum para todos os testes de um mesmo grupo, evitando assim a repetição de código para cada um deles.

Da mesma forma, o método de teardown deve ser utilizado para evitar que as modificações realizadas em um módulo durante os testes possam influenciar outros testes.

A definição de um grupo de teste é feito utilizando o macro TEST_GROUP(), que recebe como parâmetro o nome que será utilizado para identificar o grupo.

```
1   extern "c"
2   {
3       /* #include para headers com estrutura em C.
4       *  Necessário devido CppuTest ser escrito em C++
5       */
6   }
7
8   /* Biblioteca estática com macros e estruturas definidas para
9   *  utilizar o CppUTest
10  */
11  #include "CppUTest/TestHarness.h
12
```

```
13  /* Macro para dfinição de um grupo de testes */
14  TEST_GROUP( NomeDoGrupo )
15  {
16      /* Possibilidade de definir variáveis acessíveis
17      *  apenas para testes vinculados ao grupo de testes
18      */
19
20      void setup()
21      {
22          /* Definição do método de inicialização que será chamada
23          *  antes de qualquer teste do grupo
24          */
25      }
26
27      void teardown()
28      {
29          /* Definição do método de limpeza que será chamada
30          *  ao final de qualquer teste do grupo
31          */
32      }
33  }
```

Podem ser criados vários casos de teste utilizando a macro TEST() que recebe como parâmetro o nome do grupo a qual este caso de teste está vinculado e o nome do caso de teste específico. Tal nome pode ser utilizado como uma forma de documentação do teste.

```
35  /* Macro com definição de teste */
36  TEST( NomeDoGrupo, NomeDoTeste )
37  {
38      /* Caso de teste a ser definido pelo desenvolvedor com
39      *  inicializações específicas para o teste, chamada para
40      *  código em teste e assertivas de verificação específicas
41      *  do caso de teste.
42      */
43  }
44
```

Para que os testes possam ser executados é necessário ter a função main que será responsável por chamar todos os testes.

```
#include "CppUTest/CommandLineTestRunner.h"

int main(int ac, const char** av)
{
    return RUN_ALL_TESTS(ac, av);
}
```

Para gerar o executável do código de testes é necessário, no entanto, compilar a biblioteca estática do código fonte fornecido pelo CppUTest. No windows, pode ser realizado utilizando a ferramenta de GNU Cygwin, um ambiente de linha de comando Unix. Basta baixar e instalar os pacotes default e os pacotes Devel.

Com todas as ferramentas instaladas, navegue até a pasta cpputest_build, dentro da pasta do CppUTest, utilizando o Cygwin. Onde deverá ser utilizado os seguintes comando.

autoreconf .. -i

../configure

make

Assim, teremos a biblioteca estática compilada na pasta build, dentro da pasta do CppUTest, podendo ser utilizada nos testes sem erros de ligação.

O Cygwin pode ser utilizado também para gerar o executável dos testes escritos utilizando um Makefile, que é fornecido como exemplo, realizando algumas modificações.

COMPONENT_NAME - Recebe o nome do executável que será criado.

CPPUTEST_HOME - Recebe o caminho para a pasta onde o CppuTest foi salvo.

SRC_DIRS - Recebe todos os caminhos onde está implementado o código de produção, C, que será compilado.

TEST_SRC_DIRS - Recebe todos os caminhos onde está implementado os códigos de teste, incluindo os métodos que serão substituidos em tempo de ligação.

INCLUDE_DIRS - Recebe todos os caminhos onde estão implementados todos os arquivos de cabeçalho que serão utilizados, tanto de produção quanto de teste.

```
#---------
#
# CppUTest Examples Makefile
#
#----------
#Set this to @ to keep the makefile quiet
ifndef SILENCE
    SILENCE = @
endif
#--- Inputs ----#
COMPONENT_NAME = CppUTestExamples
CPPUTEST_HOME = ..

CPPUTEST_USE_EXTENSIONS = Y
CPP_PLATFORM = Gcc

# This line is overriding the default new macros.  This is helpful
# when using std library includes like <list> and other containers
# so that memory leak detection does not conflict with stl.
CPPUTEST_MEMLEAK_DETECTOR_NEW_MACRO_FILE = -include ApplicationLib/ExamplesNewOverrides.h
SRC_DIRS = \
    ApplicationLib

TEST_SRC_DIRS = \
    AllTests

INCLUDE_DIRS =\
  .\
  ApplicationLib\
  $(CPPUTEST_HOME)/include\

include $(CPPUTEST_HOME)/build/MakefileWorker.mk
```

Da mesma forma que foi compilada a biblioteca estática do CppUTest, o código de teste é compilado, ligado e executado, utilizando o comando make na pasta contendo o Makefile modificado.

Algumas assertivas são fornecidas pelo CppUTeste, são elas:

- CHECK(lógica booleana) - Verifica o resultado da lógica booleana passada, falha se for falsa
- CHECK_TEXT(lógica booleana, texto) - Idêntica a anterior, mas escreve o texto passado em caso de falha
- CHECK_FALSE(condição) - Verifica se a condição passada é falsa
- CHECK_FALSE_TEXT(condição, texto) - Verifica a condição e escreve o texto em caso de falha
- CHECK_EQUAL(expectativa, valor) - Verifica se o valor passado é igual a expectativa, utilizando o operador ==, falha em caso de diferença

- STRCMP_EQUAL(expectativa, valor) - Verifica se a const char* string passada como valor é igual à expectativa
- STRNCMP_EQUAL(expectativa, valor, tamanho) - Verifica igualdade entre a string const char* passada em valor com a expectativa, apenas no tamanho fornecido
- STRCMP_CONTAINS(expectativa, valor) - Verifica se a string const char* valor contém a string const char* expectativa
- LONGS_EQUAL(expectativa, valor) - Verifica igualdade entre dois números inteiros
- UNSIGNED_LONGS_EQUAL - Verifica igualdade entre dois números sem sinal
- BYTES_EQUAL(expectativa, valor) - Verifica igualdade entre dois bytes
- POINTERS_EQUAL(expectativa, valor) - Verifica igualdade entre dois ponteiros
- DOUBLES_EQUAL(expectativa, valor, tolerância) - Compara dois valores em ponto flutuante, dentro da tolerância passada
- FUNCTIONPOINTERS_EQUAL_TEXT(expectativa, valor, texto) - Compara dois ponteiros de função, escreve texto em caso de desigualdade
- MEMCMP_EQUAL(expectativa, valor, tamanho) - Verifica igualdade entre duas regiões de memória
- BITS_EQUAL(expectativa, valor, máscara) - Realiza comparação bit a bit, aplicando mascaramento
- FAIL(texto) - Sempre falha e escreve texto

9. Estratégia TDD para Embarcados

Como comentado anteriormente, a realização de testes diretamente no hardware possui alguns transtornos, que podem atrasar o desenvolvimento e diminuir a repetição de testes. Para evitar tais problemas, o desenvolvimento utilizando alvo, ou target, duplo se torna útil.

Target duplo é realizado quando o código é pensado para ser executado em pelo menos duas plataformas diferentes: a plataforma de hardware final e no seu sistema de desenvolvimento. Desta forma, o código pode ser testado antes do

hardware estar pronto, além de evitar longas esperas de carregamento do código no processador que será utilizado e permitir gerar situações de testes mais facilmente.

Outro benefício que o target duplo proporciona está no design. Ao prestar mais atenção nos limites entre software e hardware, o desenvolvedor tende a apresentar um design mais modular e mais independente do hardware. Permitindo, assim, uma portabilidade maior, ou uma mudança mais rápida de periféricos.

9.1. Riscos do target duplo

Embora exista grande benefício em desenvolver com target duplo, existem alguns riscos inerentes a tal abordagem. A maioria dos riscos são devido às diferenças entre a plataforma de desenvolvimento e a plataforma de hardware aplicada. Esses riscos incluem:

- Compilador pode suportar linguagens diferentes
- O compilador para o hardware alvo pode possuir um conjunto de bugs, enquanto o compilador da plataforma de desenvolvimento possui outro conjunto
- As bibliotecas de tempo de execução podem ser diferentes
- Os nomes de arquivos incluídos e suas características podem ser diferentes
- Dados primitivos podem ter tamanhos diferentes
- Ordem de bytes e alinhamento de estruturas de dado podem ser diferentes

Por causa destes erros, códigos que são executados sem nenhuma falha em um ambiente podem apresentar falhas em outro. Mesmo que tais riscos sejam contornáveis, faz-se necessário leva-los em consideração.

9.2. Ciclo do TDD para embarcados

O ciclo do TDD para embarcados é uma extensão do ciclo do TDD descrito anteriormente.

A efetividade do TDD se torna maior quando a compilação e os ciclos de teste demoram apenas alguns segundos. Uma demora de compilação, escrita na placa e

execução dos testes, tende a fazer o desenvolvedor realizar mudanças maiores por ciclo. Com mudanças maiores, uma maior quantidade de código pode falhar, aumentando o escopo de procura da causa do erro, e o tempo de debug. Dessa forma, a realização de testes com feedbacks mais rápidos torna necessária a execução do ciclo básico do TDD na plataforma de desenvolvimento.

Para mitigar os riscos inerentes do target duplo, no entanto, deve-se realizar alguns estágios de testes a mais. Os estágios de teste necessários são:

Estágio 1: Ciclo básico do TDD

Como expresso anteriormente, o primeiro estágio é executado com mais frequência. Durante este estágio, é escrito a maior parte do código e compilado para ser executado na plataforma de desenvolvimento. Testes na plataforma de desenvolvimento permitem feedback mais rápido, além de constituir um ambiente de execução estável e validado. Muitas vezes possui uma ferramenta para debug mais completa, do que a desenvolvida para o hardware específico.

Este estágio permite uma visualização melhor entre os limites do software para com o hardware, o desenvolvedor deve procurar por oportunidades para desacoplá-los, e deixar o código independente da plataforma.

Estágio 2: Verificação de compatibilidade com compilador

Deve-se, periodicamente, compilar para a plataforma alvo utilizando o compilador que se espera empregar em produção. Este estágio permite uma verificação preliminar de incompatibilidades entre compiladores, denunciando problemas de portabilidade, como arquivos de cabeçalho indisponíveis, suporte à linguagem incompatível, e funcionalidade de linguagem faltando. Forçando o desenvolvedor a utilizar apenas as funcionalidades disponíveis em ambas as plataformas.

Não é necessário realizar o estágio 2 para cada mudança de código. É preferível que seja feito quando for adicionado alguma funcionalidade nova da linguagem, incluindo um novo arquivo de cabeçalho ou uma nova chamada de biblioteca. Com as mudanças de mercado, pode utilizar a biblioteca de testes

desenvolvidos para avaliar novas versões de compiladores, ou compiladores de outras marcas.

Estágio 3: Executar testes unitários em uma placa de avaliação

Existe o risco de o código compilado executar de forma diferente nas plataformas alvo. Para diminuir tal risco, é recomendado executar os testes em uma placa de avaliação. Assim, é possível verificar em que situações o comportamento do código difere entre as plataformas.

Idealmente, seria melhor utilizar o hardware alvo, sem a necessidade de utilizar uma placa de avaliação. No entanto, nem sempre isto é possível, além de que, mesmo quando o hardware estiver disponível, a execução na placa de avaliação auxilia a identificar se algum erro está presente no hardware, ou se é erro de código.

Estágio 4: Executar testes no hardware alvo

Este estágio possui os mesmos objetivos do estágio 3. Com o aspecto adicional de poder adicionar testes específicos para o hardware alvo. Tais testes permitem caracterizar, ou até mesmo aprender, como o hardware se comporta.

Um desafio adicional a este estágio está na capacidade de memória em geral ser limitada. Fazendo com que nem todos os testes caibam no hardware alvo, dado a quantidade de código extra que deve ser adicionado para criar e caracterizar os casos de teste. Neste caso, é possível separar os testes em grupos de tamanho suficiente para serem usados.

Estágio 5: Executar testes de aceitação no Hardware alvo

Por fim, para se ter certeza que as funcionalidades do produto estão de acordo com todas as especificações, é executado testes automáticos e manuais no hardware alvo. Todo código que não puder ser automaticamente testado deve ser testado manualmente.

Durante o ciclo de vida do projeto, alguns estágios podem ser impossíveis ou não críticos. Por exemplo, quando não existe hardware pronto, os estágios 4 e 5 não

podem ser feitos. De forma similar, se o hardware estiver disponível e tiver sido validado, ou for estável, o estágio 3 pode ser deixado de lado sem grandes perdas.

A frequência de execução de cada estágio desse leva em consideração o tempo que cada teste leva para fornecer informações úteis ao desenvolvedor. Portanto, quanto mais perto do estágio 1 mais frequente deve ser sua execução, enquanto que, quanto mais perto do estágio 5 menos frequente são as execuções.

10. Estratégia TDD para Sistemas de Tempo Real

Lidar com problemas de tempo real se torna um desafio devido ao fato de lidar com estímulos, muitas vezes, imprevisíveis, lidar com interações complexas e com tempo de resposta limitado. Por isso, é importante tomar o máximo de cuidado durante seu desenvolvimento, de modo a evitar ao máximo possíveis bugs, principalmente em sistemas cuja criticidade é muito alta.

Algumas estratégias podem ser levadas em considerações quanto a utilização do TDD em sistemas desta natureza. Entre elas, pode-se citar:

I. Tempo

Duas situações básicas devem ser levadas em consideração quando for avaliar o atraso de um módulo.

Primeiramente, o atraso pode estar relacionado com a demora em conseguir algum recurso. Por exemplo, receber dados da memória. Tal situação foge do controle do desenvolvedor, logo, testes relacionados a tal problema devem verificar qual deve ser a resposta se o recurso demorar demais para ser adquirido e se conseguir o recurso em tempo, verificar se o código está trabalhando de forma correta.

Os dois tipos de teste, relacionados com o atraso de recurso, podem ser testados no estágio 1 do ciclo TDD para embarcados. É possível utilizar no código estruturas de TimeOut que garantem um tempo máximo de espera, e nos testes será

utilizado Fakes destas estruturas para direcionar os casos de teste para uma situação ou outra.

No entanto, o atraso pode ser gerado pelo próprio tempo de processamento dos recursos que foram captados. O tempo de processamento de um código é dependente da plataforma em que está sendo executado, logo, testes que levem em consideração tal situação devem ser realizados no estágio 3 ou 4 do ciclo TDD para embarcados.

Testes automáticos podem ser realizados utilizando o clock do próprio hardware alvo para verificar quantos ciclos são utilizados, na execução do código em um caso de teste específico, e se ele está condizendo com as especificações exigidas.

II. Interações

Interações com o mundo real podem se tornar muito complexas, onde cada sequência de eventos demanda um tratamento diferenciado. Portanto, para lidar com tal situação se torna imperativo a utilização de modelos, ou casos de uso, de forma a possuir um entendimento claro de como o sistema deve se comportar.

Um modelo de interação amplamente utilizado é a Máquina de Estados Finita. Neste, são definidos estados em que o sistema pode se encontrar, quais transições entre os estados são possíveis e os sinais que impulsionam tais transições.

O desenvolvedor pode então, utilizar o estágio 1 do ciclo do TDD para embarcados de forma a realizar testes na funcionalidade de cada estado, verificando em quais possíveis situações o estado é inicializado e se todas as transições de cada estado estão sendo realizadas corretamente.

Após realizado testes em cada estado individualmente, testes de integração devem ser realizados para verificar o comportamento da máquina como um todo, e todos os possíveis caminhos de interações entre os estados.

Algumas situações, no entanto, se tornam muito complicadas de serem verificadas. Por exemplo, a interação entre maquinas de estados distintas pode tornar a quantidade de caminhos possível tão grande que torna o desenvolvimento de casos de teste uma tarefa exaustiva. Portanto, é recomendado testar tais problemas no estágio 5 do ciclo TDD para embarcados, e em geral, focar em

situações mais críticas, que possam gerar Deadlock ou situações de Inanição, por exemplo.

11. Aplicação Prática

De modo a validar as técnicas explicitadas neste relatório, elas foram aplicadas no desenvolvimento de um módulo para um equipamento comercial da empresa Tomus Soluções. Trata-se de um equipamento de monitoramento remoto, capaz de trocar mensagens de telemetria com um servidor base via um protocolo baseado em pacotes de dados. Para evitar perdas de informações, as mensagens são armazenadas em memória não-volátil até que possam ser enviadas de forma efetiva.

O módulo responsável por gerenciar as mensagens na memória, no entanto, estava apresentando problemas recorrentes, o módulo não estava conseguindo encontrar as mensagens de forma satisfatória e muitas mensagens eram perdidas, apesar de estarem salvas, ou apagadas de forma errônea.

Trata-se de um módulo cuja responsabilidade é central para o monitoramento correto dos equipamentos, uma vez que ao perder as mensagens todos os dados coletados também se perdem, não sendo possível levantar o histórico corretamente.

Para solucionar o problema foi utilizado técnicas TDD. Inicialmente, foi feito uma modelagem de como deveria se comportar o módulo. Foi decidido que o módulo operaria em duas regiões de memória distintas, uma região seria composta de cabeçalhos com metadados sobre as mensagens, uma medida de segurança para ter certeza se as mensagens estariam corrompidas, qual a posição que estariam guardadas, e se já teriam sido enviadas ou não para o servidor, enquanto a outra região possuiria as mensagens em si.

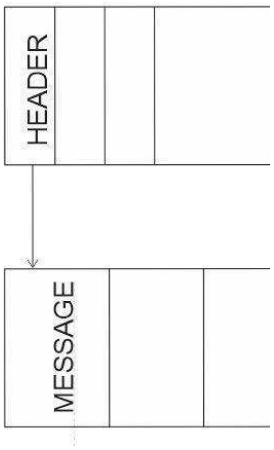

Figura 9 - Separação das Regiões de Memória

Cada região de memória foi dividida em páginas, onde cada página de cabeçalhos endereçaria os dados da mensagem, e o modo como as mensagens e cabeçalhos seriam salvos estaria estruturado como uma fila circular e o endereçamento abstraído para dois valores, um representando o índice na página e outra a própria página.

Address = {Index, Page}

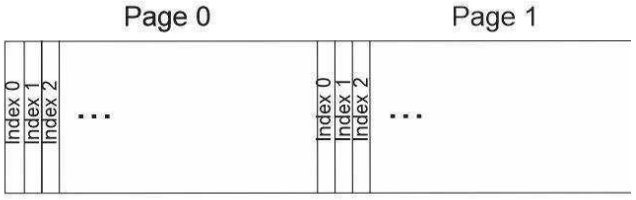

Figura 10 - Estrutura de Índices e Páginas

No contexto do módulo serão salvas três estruturas, uma indicando o próximo endereço, onde será salvo uma nova mensagem, considerado a cabeça da fila circular, outra indicando a próxima mensagem a ser lida para compor o pacote de mensagens, e uma última indicando a última mensagem enviada, que será realmente o fim da fila.

Uma vez que a mensagem foi lida para compor o pacote a ser enviado, ele não pode ser automaticamente descartado, se ocorrer algum problema de envio do pacote este deverá ser formado de novo, no entanto, quando for enviado esta

mensagem pode ser considerada descartável pelo módulo e apagado quando necessário.

Ao escrever uma nova mensagem a posição da cabeça da fila deve pular para o próximo índice vazio, se estiver no último índice da página, deve pular para o primeiro índice vazio da próxima página, e se estiver na última página deve pular para o próximo índice. Caso o apontador de escrita estiver uma posição antes do indicador de mensagens enviadas, então a fila está cheia.

O comportamento do indicador de mensagem lida segue o mesmo comportamento do de escrita, no entanto, quando estiver apontando para a mesma posição do indicador de escrita, todas as mensagens já foram lidas. De forma semelhante, quando o indicador de mensagem enviada for igual ao de leitura, todas as mensagens lidas foram enviadas.

Devido a limitações do driver de memória utilizado, e levando em consideração que apagar informação da memória flash é uma operação demorada, as mensagens que foram enviadas com sucesso não são apagadas imediatamente, deve ser esperado que uma página inteira possa ser apagada. Portanto, só se serão apagadas mensagens quando todos os apontadores já tiverem passado para uma página seguinte. Liberando espaço para quando a fila circular estiver passando por aquele endereço novamente.

Com essas considerações, a região de mensagens pode possuir endereços vazios, que podem receber uma nova mensagem, endereços lidos, no entanto, não enviados, endereços com mensagens enviadas, que ainda não foram apagadas, e por último, endereço com regiões corrompidas, seja por erro de escrita do driver, seja por desgaste físico do componente, ou por qualquer outro motivo desconhecido.

Dado que os apontadores estão no contexto do módulo, eles não serão salvos na memória não-volátil, portanto, toda vez que for reinicializado o módulo deve retomar o contexto de forma a não perder mensagens que não foram enviadas ainda, e ainda evitar que o apontador de escrita aponte para regiões que já foram enviadas, mas não foram apagadas, forçando a procura de regiões vazias e possibilitando a perda de mensagens.

A inicialização se mostrou, portanto, uma das operações mais críticas do módulo e é essa que levaremos em consideração neste exemplo.

11.1. Operação de Inicialização

Como mostrado, a inicialização do módulo constitui uma das operações mais críticas, se não a mais críticas, e deve ser tratada com cuidado. Devido à escolha do modelo, alguns casos podem ser observados para a inicialização.

O caso mais simples, será quando a memória estiver completamente vazia. Nesta situação foi escolhido que todos os apontadores devem estar apontando para o primeiro índice da primeira página.

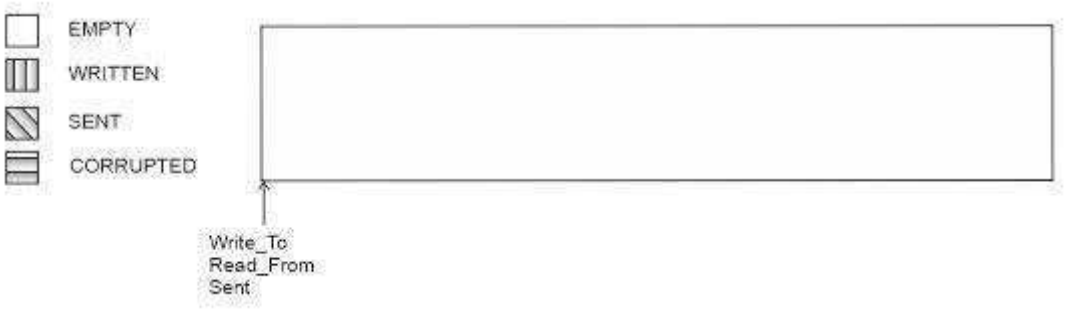

Figura 11 - Memória Vazia

O segundo caso seria o da memória estar parcialmente escrita, sem nenhuma delas ter sido enviada. Neste caso, o ponteiro de escrita deve estar um endereço após a última região escrita, e os ponteiros de leitura e envio devem estar ambos apontando para a primeira região escrita.

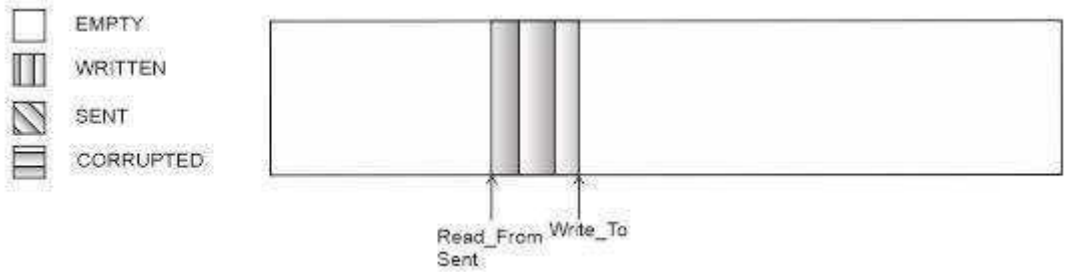

Figura 12 - Memória com Região Escrita

O terceiro caso pensado seria de parte da memória possuir mensagens enviadas, e nenhuma mensagem escrita. Nesta situação, o ponteiro de escrita deve

indicar o endereço seguinte à última mensagem enviada. O ponteiro de leitura e o de envio devem apontar para o mesmo ponto em que o de escrita.

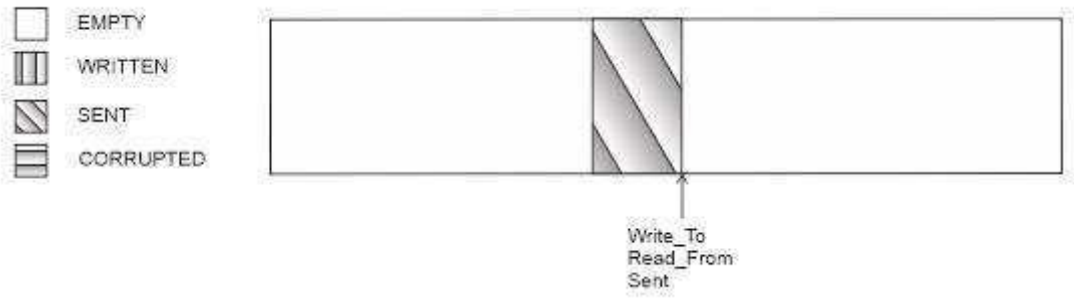

Figura 13 - Memória com Região Enviada

A quarta situação é encontrada quando parte da memória está escrita e parte foi enviada. Mesmo que não estejam coladas uma na outra, a região que está escrita terá prioridade, logo, o ponteiro de escrita deve apontar para o próximo índice vazio após a última escrita, e de forma semelhante ao segundo caso, os outros dois apontadores devem estar apontando para a primeira mensagem escrita.

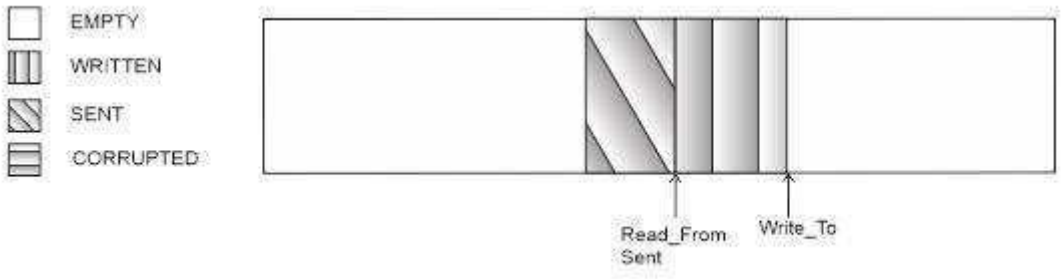

Figura 14 - Memória com Região Escrita e Região Enviada

A quinta situação pode ser encontrada quando a memória está completamente escrita e nenhuma mensagem foi enviada. Por tanto, a fila está cheia e os ponteiro de leitura envio devem ser colocados no primeiro índice da primeira página, enquanto o ponteiro de leitura deve ser colocado no último índice da última página. Vale salientar que essa situação é altamente prejudicial ao sistema, uma vez que a ordem cronológica das mensagens pode ser completamente perdida e nenhuma mensagem nova poderá ser salva.

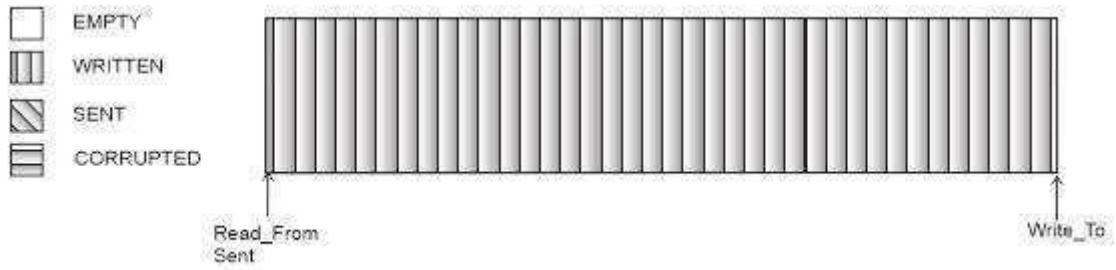

Figura 15 - Memória Completamente Escrita

Na penúltima situação, a memória possui apenas regiões de memória corrompida, neste caso o módulo deve inicializar todos os ponteiros com uma posição seguinte à região corrompida mais perto do fim da região de memória. Tal escolha é feita para que a parte corrompida possa ser apagada o mais rápido possível, além de permitir que o ponteiro de escrita demore ao máximo para chegar nessa região.

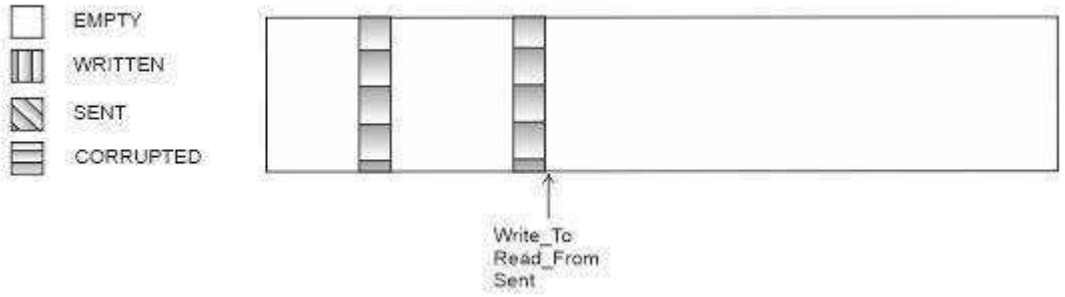

Figura 16 - Memória com Regiões Corrompidas

Por último, se a memória por algum motivo estiver escrita completamente com mensagens corrompidas. O módulo deverá apagar toda a região e posicionar todos os apontadores para o primeiro índice da primeira página.

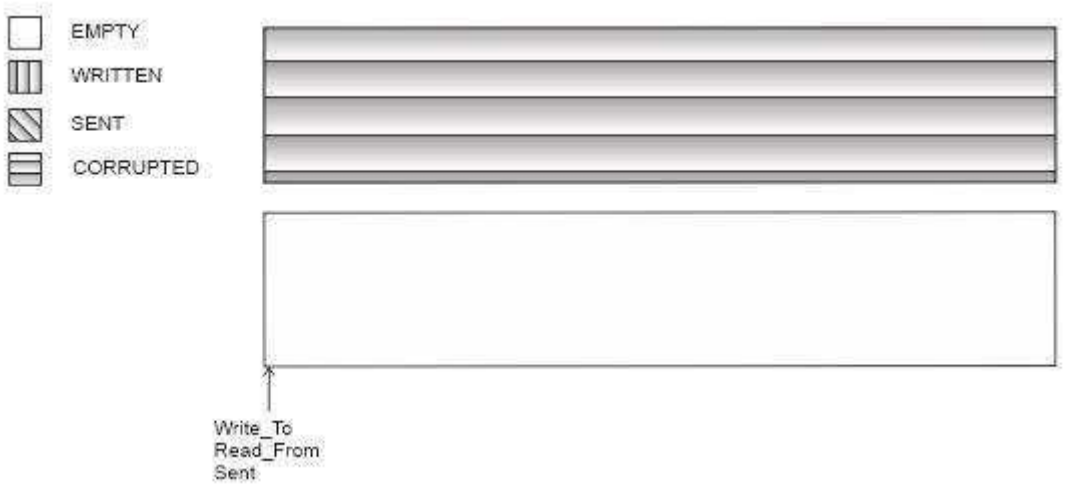

Figura 17 - Memória Corrompida

11.2. Testes realizados

Testes foram amplamente realizados, para garantir o comportamento previsto na seção anterior, utilizando a metodologia TDD para realizar mudanças e manter o funcionamento de estruturas previamente feitas. Alguns deles serão mostrados a seguir.

Todos os testes envolvidos na inicialização do módulo estão no mesmo Grupo de Testes e são sempre inicializados com a memória completamente vazia. Para isso, foi feito um módulo Fake da memória, com funções cujo o módulo de produção tem acesso, e funções que apenas o módulo de teste possui acesso.

```
20  /* Tests Initialization */
21  TEST_GROUP(MyMsgStorageTests_Init)
22  {
23      TEST_SETUP()
24      {
25          Fake_mem_mgr_eraseAll();
26      }
27
28      TEST_TEARDOWN()
29      {
30      }
31
32  };
```

As funções que o módulo de produção pode utilizar são implementações diferentes das implementações do módulo original, mas que permitem capturar informações do funcionamento módulo sobre teste. Por exemplo:

```
242  NVM_ResultCode Memory_erase(uint32_t dest, uint8_t size)
243  {
244      uint32_t erase_size = size*1024u;
245      if(erase_size + dest < TOTAL_MEMORY_SPACE)
246      {
247          Fake_Erased++;
248          memset(&Fake_memory[dest], 0xFF, erase_size);
249          return NVM_Success;
250      }
251
252      return NVM_EraseError;
253  }
```

A implementação Fake da função responsável por apagar a memória conta numa variável interna, do módulo Fake, quantas vezes foi chamada para apagar uma página e apaga um buffer interno que está sendo utilizado para simular a região de memória do equipamento.

A função utilizada no setup do grupo de testes tem um propósito um pouco diferente, ela não é acessível para o CUT, apenas para os casos de teste. Esse tipo de função pode ser utilizado preparar um ambiente de teste, ou para capturar informações sobre o andamento do teste.

```
263  void Fake_mem_mgr_eraseAll(void)
264  {
265      memset(Fake_memory, 0xFF, TOTAL_MEMORY_SPACE);
266  }
267
```

No caso, esta função está sendo utilizada para que todos os testes sejam inicializados com o buffer vazio, ou seja, que nenhuma modificação no buffer realizado por um teste possa ser sentida por outros casos de teste.

Com esses conceitos em mente, podemos então realizar o primeiro teste.

```
34  TEST(MyMsgStorageTests_Init, initTest_AllCorrupted)
35  {
36
37      void *params = NULL;
38
39      Fake_mem_mgr_SetErased(0);
40      Fake_mem_mgr_corruptAll();
41
42      __MsgStorage_InitTask(params);
43
44
45      CHECK_EQUAL(1728, Fake_mem_mgr_GetErased())
46
47  }
48
```

Esse primeiro teste é bem simples, basicamente ele é feito para verificar se quando a região de memória está completamente corrompida ela será toda apagada. Primeiramente, foi condicionado o ambiente do caso de teste, a função Fake_mem_mgr_SetErased(), na linha 39, é utilizada para zerar o contador interno do módulo Fake que indica quantas vezes a função para apagar a memória foi chamada. A função Fake_mem_mgr_corruptAll(), na linha 40, também é uma função auxiliar, que está sendo utilizado para corromper a memória toda. Em seguida o método responsável pela inicialização é executado, e na etapa de verificação podemos comparar os resultados obtidos com o esperado. Então, fake_mem_mgr_GetErased(), na linha 45, pode ser utilizado para pegar o contador interno mencionado anteriormente. Esse resultado é então comparado com a quantidade de páginas que era esperado ser apagado. Qualquer diferença entre os valores indica que a região de memória não está sendo completamente apagado, se o número for menor, ou está apagando uma região de memória que não pertence ao gerenciador de mensagens, caso o número resultante seja maior.

```
49  TEST(MyMsgStorageTests_Init, initTest_AllEmpty)
50  {
51      MsgStorage_Index_t currentReadFrom;
52      MsgStorage_Index_t nextReadFrom;
53      MsgStorage_Index_t currentWriteTo;
54
55      void *params = NULL;
56
57
58
59      __MsgStorage_InitTask(params);
60
61      currentReadFrom = MsgStorage_GetCurrentReadFrom();
62      nextReadFrom = MsgStorage_GetNextReadFrom();
63      currentWriteTo = MsgStorage_GetCurrentWriteTo();
64
65      CHECK_EQUAL(0, currentWriteTo.sector_index)
66      CHECK_EQUAL(0, currentWriteTo.header_index)
67
68      CHECK_EQUAL(currentWriteTo.sector_index, currentReadFrom.sector_index)
69      CHECK_EQUAL(currentWriteTo.header_index, currentReadFrom.header_index)
70
71      CHECK_EQUAL(currentReadFrom.sector_index, nextReadFrom.sector_index)
72      CHECK_EQUAL(currentReadFrom.header_index, nextReadFrom.header_index)
73
74  }
```

O segundo teste implementado é utilizado para verificar o comportamento quando a memória está completamente vazia, lembrando que esta é a situação inicial de todos os testos, portanto, não é necessária nenhuma ação a mais no intuito de preparar o ambiente para o caso de teste. Na etapa de verificação é utilizado funções do próprio módulo que informam as posições atuais de cada ponteiro, e é verificado o valor de cada índice na página e cada página, todos devem ser iguais a zero.

Outro exemplo de teste implementado pode ser visto pelo código abaixo. Nele é testado o funcionamento quando os 10 primeiros cabeçalhos estão corrompidos. Nesta situação a função Fake_mem_mgr_Write(), na linha 91, é responsável por escrever no buffer, utilizado para simular a memória, valores que farão o módulo considerar a região como corrompida. Ao ser exercitado, todos os ponteiros devem ser inicializados com o mesmo valor, para o índice 10 da primeira página, dado que dos índices 0 a 9 todos estão corrompidos. Durante a verificação os ponteiros são buscados e comparados com os valores esperados, utilizando as assertivas do CppUTest.

```
77  TEST(MyMsgStorageTests_Init, initTest_TenFirstHeadersCorrupted)
78  {
79      MsgStorage_Index_t currentReadFrom;
80      MsgStorage_Index_t nextReadFrom;
81      MsgStorage_Index_t currentWriteTo;
82
83      void *params = NULL;
84
85      void *src;
86      uint32_t messageSize = 10 * MESSAGE_STORAGE_HEADER_SIZE;
87
88      src = malloc(messageSize);
89      memset(src, 0xAA , messageSize);
90
91      Fake_mem_mgr_Write((uint32_t)0, src, messageSize);
92
93      free(src);
94
95      __MsgStorage_InitTask(params);
96
97      currentReadFrom = MsgStorage_GetCurrentReadFrom();
98      nextReadFrom = MsgStorage_GetNextReadFrom();
99      currentWriteTo = MsgStorage_GetCurrentWriteTo();
100
101     CHECK_EQUAL(0, currentWriteTo.sector_index)
102     CHECK_EQUAL(10, currentWriteTo.header_index)
103
104     CHECK_EQUAL(currentWriteTo.sector_index, currentReadFrom.sector_index)
105     CHECK_EQUAL(currentWriteTo.header_index, currentReadFrom.header_index)
106
107     CHECK_EQUAL(currentReadFrom.sector_index, nextReadFrom.sector_index)
108     CHECK_EQUAL(currentReadFrom.header_index, nextReadFrom.header_index)
109
110 }
```

Com o passar do tempo os casos de teste vão ficando mais complexos e se tornando cada vez mais complicado montar a situação desejada.

Uma situação de parte da memória possui mensagens enviadas e, em outra página, possui mensagens escritas, que deve ser testada, embora, uma vez que o módulo esteja em funcionamento, esse tipo de situação não vai ocorrer, é necessário realizar o teste, pois os equipamentos que já estejam em funcionamento com a versão anterior do módulo podem apresentar situações como esta.

Um código que permite o teste do funcionamento é apresentado a seguir. Inicialmente é escrito nos 40 primeiros índices da terceira página mensagens, que já

foram marcadas como enviadas. Boa parte do tamanho do código é para garantir que a região está corretamente preenchida.

```
631  TEST(MyMsgStorageTests_Init, initTest_TenFirstOfFourthSectorCorruptedAndTenFirstOfSecondSectorValidToRead)
632  {
633      void *params = NULL;
634      void *src;
635
636      MsgStorage_PktHdr_t header_pkt;
637
638      MsgStorage_Index_t currentReadFrom;
639      MsgStorage_Index_t nextReadFrom;
640      MsgStorage_Index_t currentWriteTo;
641
642      uint32_t dataSector = 3;
643      uint32_t dataIndex = 0;
644
645      uint32_t dataAddress = 0;
646      uint32_t headerAddress = 0;
647
648      uint16_t messageSize = 1 * MESSAGE_STORAGE_DATA_BLOCK_SIZE;
649
649
650      /* Set sent region */
651      for (uint32_t i = 0; i < 10; i++)
652      {
653          dataIndex = i;
654
655          dataAddress = Mem_Mgr_GetStorageStartAddress(MEM_MGR_StorageMessageData) + //Data Base Address
656                  ((dataSector * MESSAGE_STORAGE_HEADERS_PER_SECTOR) + dataIndex ) * //Pointers placement
657                  MESSAGE_STORAGE_DATA_BLOCK_SIZE; // Data size
658
659          src = malloc(messageSize);
660          memset(src, 0xAA , messageSize);
661
662          Fake_mem_mgr_Write( dataAddress, src, messageSize);
663
664          free(src);
665
666          headerAddress = Mem_Mgr_GetStorageStartAddress(MEM_MGR_StorageMessageHeaders) + //Header Base Address
667                  (dataSector * MESSAGE_STORAGE_HEADERS_PER_SECTOR + dataIndex) * //Pointers placement
668                  MESSAGE_STORAGE_HEADER_SIZE; //Header size
669
670          /* Creating confirmed header */
671          header_pkt = Test_BuildSentHeader();
672
673          Fake_mem_mgr_Write(headerAddress , &header_pkt , MESSAGE_STORAGE_HEADER_SIZE );
674
675      }
```

```
677     /* Set writen region */
678     dataSector = 1;
679     for (uint32_t i = 0; i < 10; i++)
680     {
681         dataIndex = i;
682 
683         dataAddress = Mem_Mgr_GetStorageStartAddress(MEM_MGR_StorageMessageData) + //Data Base Address
684                 ((dataSector * MESSAGE_STORAGE_HEADERS_PER_SECTOR) + dataIndex ) * //Pointers placement
685                 MESSAGE_STORAGE_DATA_BLOCK_SIZE; // Data size
686 
687         src = malloc(messageSize);
688         memset(src, 0xAA , messageSize);
689 
690         Fake_mem_mgr_Write( dataAddress, src, messageSize);
691 
692         free(src);
693 
694         headerAddress = Mem_Mgr_GetStorageStartAddress(MEM_MGR_StorageMessageHeaders) + //Header Base Address
695                 (dataSector * MESSAGE_STORAGE_HEADERS_PER_SECTOR + dataIndex) * //Pointers placement
696                 MESSAGE_STORAGE_HEADER_SIZE; //Header size
697 
698         /* Creating writen header */
699         header_pkt = Test_BuildWritenHeader();
700 
701         Fake_mem_mgr_Write(headerAddress , &header_pkt , MESSAGE_STORAGE_HEADER_SIZE );
702 
703     }
704 
705     __MsgStorage_InitTask(params);
706 
707     currentReadFrom = MsgStorage_GetCurrentReadFrom();
708     nextReadFrom = MsgStorage_GetNextReadFrom();
709     currentWriteTo = MsgStorage_GetCurrentWriteTo();
710 
711 
712     CHECK_EQUAL(1 , currentWriteTo.sector_index)
713     CHECK_EQUAL(10 , currentWriteTo.header_index)
714 
715     CHECK_EQUAL(1, currentReadFrom.sector_index)
716     CHECK_EQUAL(0, currentReadFrom.header_index)
717 
718     CHECK_EQUAL(currentReadFrom.sector_index, nextReadFrom.sector_index)
719     CHECK_EQUAL(currentReadFrom.header_index, nextReadFrom.header_index)
720 
721 }
```

A segunda parte do código acima é feita para escrever nos 10 primeiros índices da segunda página mensagens válidas que ainda não foram enviadas. Embora as mensagens não sejam informações úteis, o CUT deve perceber tais mensagens como válidas, por meio dos metadados corretos.

Nesta situação o ponteiro de escrita deve ser inicializado apontando para o índice 10 da segunda página, pois esta é o primeiro cabeçalho vazio após a região escrita, e os ponteiros de leitura e de mensagens enviadas devem estar apontando para o índice zero da mesma página.

Como último exemplo temos um caso de teste utilizado para medir um pouco da resposta temporal da inicialização. Lembrando que este teste deve ser realizado no estágio 3 ou 4 do ciclo TDD para embarcados, e para ter uma resposta mais precisa, utilizando o driver de memória original, podendo ser utilizadas funções do

mesmo para montar os casos de teste, do mesmo modo que a funções do módulo Fake foram utilizadas.

```
76  TEST(MyMsgStorageTests_Init, initTest_TimePerformanceAllEmpty)
77  {
78      MsgStorage_Index_t currentReadFrom;
79      MsgStorage_Index_t nextReadFrom;
80      MsgStorage_Index_t currentWriteTo;
81
82      uint32_t startTime;
83      uint32_t finishTime;
84      uint32_t MaxTimeAllowed = 1000 * 60 * 1; // 60000ms = 1 min
85      void *params = NULL;
86
87      startTime = SysClock_GetTickMilliseconds();
88      __MsgStorage_InitTask(params);
89      finishTime = SysClock_GetTickMilliseconds();
90
91      currentReadFrom = MsgStorage_GetCurrentReadFrom();
92      nextReadFrom = MsgStorage_GetNextReadFrom();
93      currentWriteTo = MsgStorage_GetCurrentWriteTo();
94
95      CHECK_EQUAL(0, currentWriteTo.sector_index)
96      CHECK_EQUAL(0, currentWriteTo.header_index)
97
98      CHECK_EQUAL(currentWriteTo.sector_index, currentReadFrom.sector_index)
99      CHECK_EQUAL(currentWriteTo.header_index, currentReadFrom.header_index)
100
101     CHECK_EQUAL(currentReadFrom.sector_index, nextReadFrom.sector_index)
102     CHECK_EQUAL(currentReadFrom.header_index, nextReadFrom.header_index)
103
104     CHECK( ( finishTime - startTime) < MaxTimeAllowed)
105 }
```

Neste teste a memória foi inicializada completamente vazia, e é testado se o tempo de inicialização nesta situação ultrapassa um minuto. Embora essa situação seja simples, é uma das mais demoradas, uma vez que para ter certeza que a memória está completamente apagada é necessário percorrer todas a páginas e índices.

Para o teste foi utilizado um módulo responsável pelo funcionamento do real time clock (RTC) do processador, a função utilizada retorna, em milisegundos, o tempo passado entre a última vez que o processador foi resetado até o momento em que a função foi chamada. Utilizando o RTC como referência, é possível ter uma ideia segura de como o sistema se comporta.

Na etapa de verificação foram utilizados assertivas para garantir que o módulo inicializou os ponteiros nas posições corretas, e se a diferença de tempo entre o momento que a função foi chamada e quando ela terminou não foi superior ao tempo máximo permitido, de um minuto.

11.3. Resultados Finais

O código desenvolvido utilizando as técnicas e testes mencionados fazem parte de um trabalho realizado para a empresa Tomus Soluções, portanto, considerado como propriedade intelectual da empresa, e não será reproduzido neste trabalho. No entanto, alguns resultados podem ser apresentados.

A criação de testes constituiu um trabalho que aparenta ser contra produtivo, no entanto, foi percebido que com o passar do tempo, o desenvolvimento de um teste novo passa a ser feito com maior rapidez e facilidade, dado que todas as dependências já foram quebradas e não há necessidade de criar novos módulos Fakes.

Após testes realizados, que levaram em consideração o tempo, foi percebido que o tempo de inicialização superava, e muito, o tempo máximo permitido, forçando uma mudança na abordagem de inicialização. Tal fato explicitou a importância de realizar testes no hardware alvo, levando em consideração a performance. Se o módulo continuasse inicializando com esse tempo, algumas mensagens poderiam estar prontas antes do gerenciador estar inicializado, e não seriam salvas.

Por último, desde a implementação do código até a data em que foi escrito este relatório, nenhuma reclamação foi feita em relação ao módulo desenvolvido. Logo, o módulo está estável e se comportando de forma desejada. Não é possível afirmar que não exista nenhum bug no módulo, no entanto, pode-se garantir que não existem bugs críticos que permitam a perda de informações.

12. Conclusão

Neste trabalho foi possível avaliar que o uso do desenvolvimento dirigido por testes, traz uma quantidade de vantagens significantemente maior que a quantidade de desvantagens. As desvantagens dessa técnica estão basicamente ligadas ao tempo gasto pelo programador para executar os testes, entretanto, o custo benefício é favorável, pois durante esse tempo ele gastará menos com correções de erros e ao encerrar os testes terá um projeto de alta qualidade.

Observou-se também que os sistemas de tempo real estão bastante presentes no nosso dia-a-dia e a eficácia para esse tipo de sistema é essencial, principalmente em sistemas críticos, que só são válidos se reagirem de imediato aos estímulos do ambiente e possuem custos elevados para falha do sistema.

Dependências no código são difíceis de controlar e muitas vezes limitam o escopo dos testes, já que precisam estar funcionando corretamente antes da realização dos testes, para os testes serem bem-sucedidos. Portanto, deve-se isolar o código e utilizar estruturas que substituam as estruturas reais durante os testes, isto é, utilizará estruturas que irão simular o comportamento de estruturas reais de maneira controlada, sem que o CUT perceba tal substituição. Existem também alguns mecanismos de substituição em C que são capazes de quebrar essas dependências, para facilitar a aplicação dos testes.

O framework para testes unitários consiste em um pacote de software, capaz de mostrar como o código de produção deverá se comportar, emitindo relatórios de falha ou de sucesso do código. Existem vários tipos de framework, cabe ao programador decidir qual o que melhor se encaixa com seu teste, podendo também utilizar mais de um framework em um teste unitário de classe.

O Desenvolvimento dirigido por testes aplicado a sistemas embarcados, deve seguir algumas etapas para minimizar os riscos causados pelo target duplo, que consiste no fato do código ser projetado a fim de executá-lo em pelo menos duas plataformas diferentes. A aplicação do TDD a esses sistemas traz também outros benefícios fora os convencionais, como a remoção do longo processo de debug no

target e o isolamento do hardware e do firmware, que é o conjunto de instruções operacionais programadas diretamente no hardware de um equipamento eletrônico.

Para Sistemas de Tempo Real, deve-se utilizar o tempo e interações como estratégias para aplicar o TDD, onde o tempo será utilizado como estratégia nos estágios 1,3 ou 4 do ciclo do TDD em sistemas embarcados, já as interações serão consideradas estratégias nos estágios 1 e 5 desse ciclo.

Na aplicação prática, o código foi desenvolvido na empresa Tomus Soluções, a partir da técnica de Desenvolvimento Dirigido por Testes em Sistemas Embarcados de Tempo Real, com a finalidade de monitoramento. Esse equipamento de monitoramento remoto se comunica com o servidor base a partir de pacotes de mensagens, que serão gerenciadas pelo módulo desenvolvido. Para que esse módulo funcionasse como previsto, usou-se técnicas TDD. Após a realização dos testes, observou-se um resultado final satisfatório, com isso foi possível confirmar que esta técnica realmente propicia maior confiança e segurança ao código.

13. Referências

[1] BARROS, E. CAVALCANTE, S., "**Introdução a Sistemas Embarcados**". CIn/UFPE

[2] FARINES, J-M., FRAGA, J. S., OLIVEIRA,R. S. de, "**Sistemas em Tempo Real**". Florianópoles, Julho de 2000

[3] GRENNING, J., W. "**Test-Driven Development for Embedded C**". The Pragmatic Bookshelf, Maio de 2011

[4] MEZAROS, G., "**xUnit Test Patterns - Refactoring Test Code**". Pearson Education, Inc, Maio de 2007

[5] "**CppUTest unit testing and mocking framework for C/C++**". Disponivel em: <https://cpputest.github.io> Acesso em: 01/10/2016

[6] Karlesky, M., Williams, G., Bereza,W., Flatcher, M., "**Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns**", San Jose, California, Estados Unidos, Abril de 2007

[7] "**TDD: fundamentos do desenvolvimento orientado a testes**".

Disponível em: <http://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151> Acesso em: 10/10/2016

[8] "**Seleção de Sistemas Operacionais de Tempo Real para Sistemas Embarcados**".
http://www.aedb.br/seget/arquivos/artigos08/361_SELECAO_DE_SISTEMAS_OPERACIONAIS-SEGeT.pdf> Acesso em 04/10/2016

[9] LAVRATTI, Felipe. "**Menos bugs, maior qualidade: TDD - Test-Driven Development"**, 2013. Disponível em: <http://www.embarcados.com.br/tdd/>
Acesso em: 11/10/2016

[10] "**Artigo Engenharia de Software - Introdução a Teste de Software**". Disponível em: <http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-teste-de-software/8035> Acesso em: 11/10/2016

[11] BERNAL, Volnys Borges, HIRA, Adilson. “**Tipos de teste de software”**. Disponível em: <https://disciplinas.stoa.usp.br/pluginfile.php/384739/mod_resource/content/1/Aula%205_2014_Tipos-de-teste-software-v2.pdf> Acesso em: 09/10/2016

[12] DETI, Luis Almeida. “**Restrições Temporais: Origem e Caracterização”**, 2007. Disponível em: <http://www.inf.ufpr.br/lmperes/transp1.pdf> Acesso em: 01/10/2016

[13] NAZARIO, Renan Leme, RUFINO, Ricardo. “**O conceito de TDD no desenvolvimento de software”**, 2015. Disponível em: <http://web.unipar.br/~seinpar/2015/_include/artigos/Renan_Leme_Naz%C3%A1rio.pdf> Acesso em: 06/10/2016

[14] “**A estratégia militar de uma potência: China**.”, 2015. Disponível em: <https://dinamicaglobal.wordpress.com/2015/06/17/a-estrategia-militar-de-uma-potencia-china/> Acesso em: 04/10/2016

[15] Evandro, “**Sistema de radar online monitora aviões no mundo todo”**, 2011. Disponível em: <http://www.postmania.org/sistema-de-radar-online-monitora-avioes-no-mundo-todo/> Acesso em: 04/10/2016

[16] PARVANEH, Alyka. “**Monitor Cardíaco”**, 2016. Disponível em: < http://cantosaudcia.blogspot.com.br/2016/03/monitor-cardiaco.html> Acesso em: 04/10/2016

[17] “**Semáforo: Ser ou não ser inteligente?**”, 2015. Disponível em: < http://docplayer.com.br/8374478-Semaforo-ser-ou-nao-ser-inteligente.html> Acesso em: 04/10/2016

[18] REZENDE, B. A. C., "**Utilização de TDD em Projetos de Software: Estudo de Caso Acadêmico**", Belo Horizonte, 2011