Universidade Estadual de Londrina

SAMUEL HEITOR DE CAMARGO LOURENÇO

INSERÇÃO DAS MELHORES PRÁTICAS DE TESTE DE SOFTWARE NO CICLO DE DESENVOLVIMENTO DE SISTEMAS EMBARCADOS AUTOMOTIVOS

LONDRINA - PR
2013

SAMUEL HEITOR DE CAMARGO LOURENÇO

INSERÇÃO DAS MELHORES PRÁTICAS DE TESTE DE SOFTWARE NO CICLO DE DESENVOLVIMENTO DE SISTEMAS EMBARCADOS AUTOMOTIVOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Estadual de Londrina, como requisito parcial à obtenção do grau de Bacharel.

Orientadora: Dra. Jandira Guenka Palma

LONDRINA - PR
2013

SAMUEL HEITOR DE CAMARGO LOURENÇO

INSERÇÃO DAS MELHORES PRÁTICAS DE TESTE DE SOFTWARE NO CICLO DE DESENVOLVIMENTO DE SISTEMAS EMBARCADOS AUTOMOTIVOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Estadual de Londrina, como requisito parcial à obtenção do grau de Bacharel.

COMISSÃO EXAMINADORA

_____
Prof. Dra. Jandira Guenka Palma
Universidade Estadual de Londrina

_____
Prof. Componente da Banca
Universidade Estadual de Londrina

_____
Prof. Componente da Banca
Universidade Estadual de Londrina

Londrina, _____de _____de _____.

Agradecimentos

Agradecimentos.

dedicatória, (...)

LOURENÇO, Samuel Heitor De Camargo. **Inserção das melhores práticas de teste de software no ciclo de desenvolvimento de sistemas embarcados automotivos**. 2013. <Número total de folhas>. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2013.

Resumo

Palavras-Chaves: Teste de software, software embarcado

LOURENÇO, Samuel Heitor De Camargo. **Insertion of software testing best practices in the development cycle of automotive embedded systems.** 2013. <Número total de folhas.> Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2013.

Abstract

Keywords: Software Test, Embedded Systems

Lista de Figuras

Sumário

1. INTRODUÇÃO

1.1. Justificativa do Trabalho

1.2. Objetivo Principal

O objetivo deste trabalho de conclusão de curso é identificar e aplicar as técnicas de teste de software que são mais adequadas no ciclo de desenvolvimento de software embarcado automotivo.

1.3. Objetivos Complementares e Organização do Trabalho

Para o detalhamento mais preciso dos objetivos deste trabalho, elencamos os seguintes objetivos específicos:

- Estudar as diversas técnicas de teste de software existentes e as estratégias de inserção dessas técnicas no ciclo de desenvolvimento de software.
- Aplicar as técnicas de teste que melhor se enquadram no desenvolvimento de software embarcado automotivo. Esta aplicação se dará levando-se em conta a especificidade do sistema, as características peculiares do sistema, tanto de desenvolvimento quanto de atuação do sistema, o que leva à incapacidade ou não de aplicação de uma determinada técnica de teste em questão.
- Realizar a aplicação das técnicas através de um estudo de caso cujo objeto de estudo será uma Unidade de Controle Eletrônica (Electronic Control Unit, ECU) utilizada em um automóvel, como o painel de instrumentos (também chamado de cluster) ou uma unidade de bloqueio e rastreamento veicular.

- Propor aplicações de ferramentas mais adequadas para cada característica do sistema, levando-se em conta a possibilidade e viabilidade de automação dos testes.

2. Fundamentação Teórico-Metodológica

Neste capítulo, serão abordados os conceitos científicos utilizados na realização deste trabalho. Primeiramente, é necessário compreender os fundamentos básicos do teste de software, bem como os termos utilizados e definições básicas.

A seguir, são apresentadas as técnicas utilizadas para se desenvolver eficientemente os casos de teste do software. Com estas técnicas definidas, serão abordadas as estratégias básicas de teste utilizadas no ciclo de desenvolvimento do software. Para este fim, será tomado como base o trabalho de Roger S. Pressman [1], pois se trata de um texto considerado clássico sobre Engenharia de Software e aborda de maneira detalhada as técnicas e estratégias de teste de software.

Neste ponto, também será apresentado o desenvolvimento voltado à teste, também conhecido como *Test Driven Development* (TDD), baseado nos trabalhos de Ian Sommerville [2] e Kent Beck [4].

Após os conceitos de teste de software, será definido o que é um sistema embarcado e quais são suas características principais que tornam um desafio a elaboração de casos de teste realmente eficientes. Serão discutidas também características relacionadas aos sistemas operacionais embarcados e os protocolos de comunicação mais utilizados.

2.1. Teste de Software

O processo de teste do software é extremamente importante e um elemento crítico para garantir a qualidade do software. [1] Apesar de parecer um tanto quanto exagerado, esta implicação deve ser enfatizada em demasia.

Atualmente, é crescente a presença de software nos mais diversos aspectos da vida humana. Desde um simples sistema embarcado que controla a mudança de canais em um televisor até os mais complexos

algoritmos que auxiliam controladores de voo, o software já está tão arraigado em nossas vidas que é capaz de mudar nossa cultura e nossos hábitos.

Por outro lado, tamanha inserção apresenta um grande risco financeiro quando uma falha de software ocorre, gerando custos altíssimos e até mesmo colocando vidas humanas em perigo, se a falha ocorrer em um sistema crítico.

A atividade de teste também representa uma revisão de especificação, projeto e codificação. [1] A revisão de especificação tem como objetivo validar se as funcionalidades desejadas estão presentes no software. Uma revisão de projeto verifica se o software foi construído da maneira como foi planejado. Uma revisão de codificação pode melhorar a legibilidade do código, aumentando assim a manutenibilidade do software.

Todos estes fatos devem servir como incentivo para maior ênfase, minuciosidade e planejamento para as atividades de teste de software a serem executadas em um projeto de desenvolvimento de software.[1]

2.1.1. Fundamentos de Teste de Software

Antes de abordar as técnicas e estratégias de teste de software, é necessário a definição dos termos básicos utilizados, bem como nomear os artefatos envolvidos no processo de teste.

Também é importante definir claramente os objetivos de uma atividade de teste, além de prover uma visão geral a respeito de alguns aspectos do teste de software.

Quando se testa um software, o objetivo deve ser agregar algum valor à ele, por aumentar a sua qualidade e confiabilidade. [6]

Glenford Myers em [6] define precisamente o que é teste de software. Assumindo-se que o software contém erros, podemos definir o teste de software como um processo de executar o software com a intenção de encontrar erros.

Neste ponto é importante salientar precisamente a definição de erro, defeito e falha. Estas definições se encontram em *Standards Glossary for Software Engineering Terminology*, desenvolvido pelo IEEE. Este é um

documento que define os termos utilizados no campo da Engenharia de Software. [3]

Defeito (*fault*) é um ato inconsistente, um equívoco ou um engano cometido pelo indivíduo que desenvolve o software ao tentar resolver um problema ou ao utilizar uma ferramenta ou método. É uma ação humana que produz um resultado incorreto. Um exemplo de defeito pode ser quando o programador acidentalmente confunde uma variável e digita o seu nome incorretamente ou digita uma instrução ou comando incorretamente. [3] [7] [8]

Erro (*error*) é a manifestação concreta de um defeito. É uma anomalia em um artefato de software causada por um defeito, que resulta em um comportamento diferente daquele especificado. Um exemplo de erro é quando se obtém um resultado diferente daquele que é o resultado esperado na execução de um artefato de software. [3] [7]

Falha (*failure*) é a incapacidade do artefato de software em executar suas funções requeridas dentro dos requisitos de performance especificados. Comumente, o termo *bug* também é usado para expressar o significado de falha e, por consequência, o termo *debug* é usado para se referir ao processo de correção de um *bug*, que pode envolver profundas análises de valores de memória e cuidadosa análise de fluxos de controle em busca da origem da falha. Uma falha pode ser causada por mais de um erro, mas alguns erros podem nunca causar uma falha. [3] [7] [8]

Durante o ciclo de desenvolvimento de software, as falhas geralmente são detectadas pelos testadores no decorrer da atividade de teste. Estas falhas são então comunicadas aos desenvolvedores para que possam corrigir os erros causadores da falha encontrada. [8]

O esquemático abaixo nos dá uma visão mais clara sobre as relações entre defeito, erro e falha. Um defeito, que se encontra no universo físico que é o software em si, origina-se de um ato humano equivocado. Ao executar o artefato de software, no universo da informação, ocorre um erro, um desvio da especificação produzido pelo defeito. Um erro então, pode se manifestar através de uma falha, que é um comportamento inconsistente no universo do usuário do referido artefato de software.

Figura 1 - Relação entre defeito, erro e falha [7]

A documentação de uma atividade de teste de software deve ser precisa e bem definida. Para tanto, é necessário uma definição para o termo *caso de teste*.

O caso de teste é a abordagem mais comum para a detecção erros em um software. [8] Em [3] também encontramos uma definição para tal termo.

Caso de teste (*test case*) é um conjunto de entradas de teste, condições de execução e resultados esperados que são desenvolvidos com o objetivo de estimular certa parte do código de um programa para verificar a conformidade com a especificação [3].

Com todos estes conceitos definidos, inicia-se agora a apresentação das técnicas fundamentais para elaboração de casos de testes eficientes.

2.1.2. Técnicas de Teste de Software

Em [9], os autores Kaner, Falk e Nguyen traçam quatro atributos fundamentais para um teste ser considerado um bom teste.

Primeiro, um bom teste tem alta probabilidade de encontrar um erro. Com o objetivo de desenvolver casos de teste que atendam este quesito, um testador deve entender a lógica do software que está testando e ter uma certa compreensão mental de como essa lógica poderia falhar.

Segundo, um bom teste não é redundante. Devido às limitações de tempo e outros recursos destinados à atividade de teste, não há razão para se executar um caso de teste que tenha o mesmo objetivo de outro caso de teste. Todo caso de teste deve ter um propósito diferente. No entanto, as diferenças entre alguns casos de teste podem ser sutis, o que deve ser considerado pelo testador.

Terceiro, um bom teste deve ser o melhor de sua classe. Quando há um grupo ou uma classe de testes a ser executada onde todos os testes possuem um objetivo similar, podem ocorrer durante a atividade de teste algumas limitações de tempo e recurso e, então, somente um sub-grupo desses testes será de fato executado. Nesses casos, os melhores testes devem ser usados, que são aqueles que possuem alta probabilidade de encontrar a maioria dos erros.

Quarto, um bom teste não deve ser nem muito simples e nem muito complexo. Este aspecto é encontrado quando uma série de testes é combinada em apenas um caso de teste. Apesar de isto ser possível, esta abordagem pode mascarar erros, portanto, geralmente, cada caso de teste deve ser executado separadamente.

Visando alcançar estes objetivos, existem inúmeras técnicas de desenvolvimento de casos de teste. Neste trabalho, baseando-se no trabalho de Pressman em [1], serão apresentadas as principais técnicas de teste.

Todo software pode ser testado utilizando-se de duas abordagens principais, que englobam todas as técnicas existentes:

- **Teste de Caixa Branca:** em inglês *white-box testing*, também chamado de teste estrutural, é uma abordagem onde o teste é conduzido sabendo-se o funcionamento interno do software, isto é, examinando-se de perto o código do software, para garantir que todas as operações internas são executadas conforme as especificações. [1]
- **Teste de Caixa Preta:** em inglês *black-box testing*, também chamado de teste funcional, é a segunda abordagem de teste, onde o teste é conduzido conhecendo-se a funcionalidade especificada para o software, a qual ele deve executar. O teste então é conduzido de forma a demonstrar que todas as funções especificadas para o software estão completamente

> operacionais enquanto simultaneamente busca-se por erros nestas funções. Diferentemente do teste de caixa branca, o teste de caixa preta nos remete à ideia de um teste conduzido na interface do software com o usuário, sem o conhecimento do funcionamento interno do software, ou seja, sem a observação do código da lógica interna. [1]

Apesar de parecer que utilizando somente a abordagem de teste de caixa branca pode-se garantir software 100% livre de erros, é fácil demonstrar que isto é impraticável. Um teste de caixa branca exercita a estrutura de controle de um programa através da definição de todos os caminhos lógicos possíveis do programa, que passam pelos condicionais e pelos laços. [1] Sendo assim, ambas estas técnicas devem ser aplicadas em uma atividade de teste de software, uma como complemento da outra, e encontrar um equilíbrio entre as abordagens para uma cobertura adequada de todo o software é o desafio a ser enfrentado pelo testador.

2.1.2.1. Técnicas de teste de caixa branca

O teste de caixa branca, também chamado de teste estrutural, é uma técnica onde os casos de teste são derivados com base na estrutura de controle do programa, ou seja, diretamente do código fonte do software. Os testes de caixa branca englobam as seguintes técnicas:

- **Teste de caminho básico:** os casos de teste são derivados com o objetivo de executar cada instrução do programa pelo menos uma vez;
- **Teste de condição:** esta técnica põe à prova as condições lógicas contidas no código fonte do programa;
- **Teste de fluxo de dados:** seleciona os caminhos de teste de acordo com o uso de variáveis;
- **Teste de laços:** esta técnica concentra-se exclusivamente na validade de construções de laços.

2.1.2.2. Técnicas de teste de caixa preta

Os testes de caixa preta se concentram nos requisitos funcionais do software, portanto, são derivados diretamente dos requisitos. Nos testes de caixa preta não se considera a estrutura interna, ou seja, não é baseado no código fonte do software. Os testes de caixa preta são executados considerando-se as saídas produzidas para determinado conjunto de entradas. Os testes de caixa preta englobam as seguintes técnicas:

- **Particionamento de equivalência:** técnica que divide as entradas de um programa em classes de dados e os casos de teste são derivados a partir dessas classes, que podem ser um valor numérico, um intervalo de valores ou um conjunto de valores relacionados.
- **Análise de valor limite:** os erros tendem a ocorrer com maior frequência nas fronteiras do domínio de entrada. Esta técnica concentra-se no teste destes valores que estão nas fronteiras;
- **Técnicas de grafo causa-efeito:** nesta técnica são levantadas as possíveis condições de entrada (que são as causas) e as possíveis ações do programa (que são os efeitos), as causas e os efeitos são relacionados através de um grafo, que é convertido em uma tabela de decisão, de onde são derivados os casos de teste;
- **Testes de comparação:** técnica utilizada quando há redundância de hardware ou software, isto é, dois ou mais sistemas semelhantes que trabalham simultaneamente. É feita a comparação entre as saídas geradas em ambos os sistemas para um mesmo conjunto de entrada.

2.1.3. Estratégias de Teste de Software

As estratégias para aplicação dessas técnicas de teste em um projeto de desenvolvimento de software também serão estudadas neste trabalho. A abordagem clássica, descrita por Pressman [1], descreve como cada etapa de desenvolvimento serve como base para o planejamento de uma correspondente etapa de teste. Molinari [11] apresenta essa abordagem como é mais conhecida: o modelo V. As características deste modelo são:

- A fase de levantamento de requisitos serve como base para os testes de validação, que são testes funcionais de alto nível (caixa preta) onde será avaliado se o software funciona da maneira especificada pelos requisitos;
- A fase de projeto de sistema serve como base para os testes de integração, que são testes executados após a integração dos módulos do sistema, testando-se interfaces de comunicação entre os módulos;
- A fase de projeto de módulo serve como base para os testes unitários, que são executados nos módulos individuais do sistema (unidades) e são testes de caixa branca. [12]

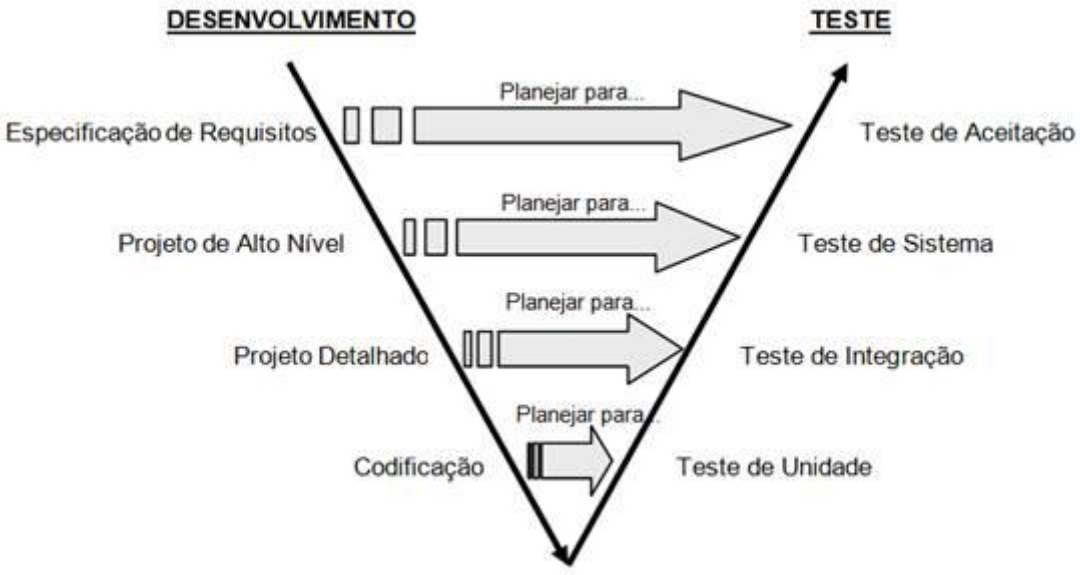

Figura 2 - O modelo V

2.2. Desenvolvimento Focado em Teste

O desenvolvimento focado em teste ou, em inglês, Test Driven Development (TDD), tem como ideia principal intercalar desenvolvimento de código e atividades de teste. [2]

O desenvolvimento se dá de forma incremental e os testes para um determinado incremento são criados antes da criação do código que implementa este incremento. Desta forma, os programadores desenvolvem o incremento com o objetivo de obter aprovação do código pelos casos de teste previamente criados. [2]

O desenvolvimento não continua para o próximo incremento até que todos os testes do incremento atual sejam executados e aprovados. [2]

O fluxograma abaixo dá uma visão aprimorada de como é o processo de desenvolvimento focado em testes.

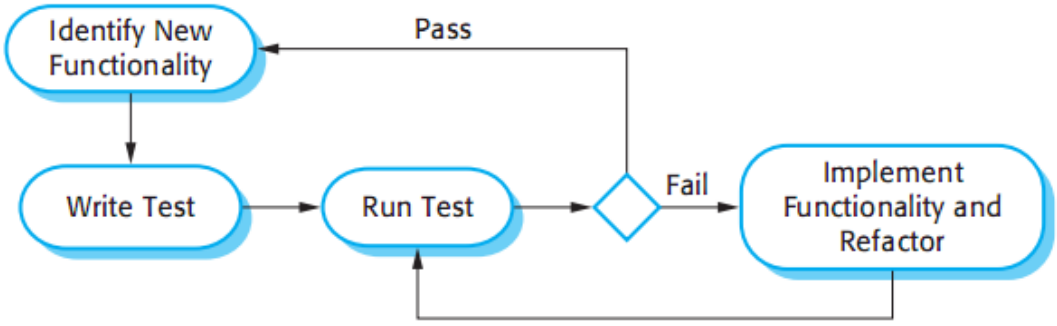

Figura 3 - Processo Fundamental do *Test Driven Development* [2]

Podemos observar neste fluxograma quatro atividades que compõem o processo do TDD.

- **Identificar nova funcionalidade (*Identify New Functionality*):** é o ponto de partida, onde se identifica uma nova funcionalidade a ser implementada no software. Geralmente, esta funcionalidade deve ser pequena e implementável em apenas algumas linhas de código, como, por exemplo, um método. [2]

- **Escrever o teste (*Write Test*):** nesta etapa, os testes para a funcionalidade são escritos. Para isso, é imprescindível o uso de uma ferramenta de automação de testes unitários. Assim os testes podem ser executados facilmente e produzir o resultado (“passou” ou “falhou”) rapidamente. [2]
- **Executar o teste (*Run Test*):** após a automação dos casos de teste, toda a bateria de testes é executada. Inicialmente, todos os testes irão falhar, pois a funcionalidade ainda não foi implementada. [2]
- **Implementar a funcionalidade e refatorar (*Implement Functionality and Refactor*):** por fim, a nova funcionalidade é implementada e os testes são executados novamente. O código é desenvolvido visando a aprovação nos testes. Isso envolve refatoração e otimização do código e frequentes execuções dos testes durante o desenvolvimento. [2]

O desenvolvimento só é continuado para a próxima funcionalidade somente quando se obtem o resultado “passou” em todos os testes da funcionalidade atual.

Devido ao rápido desenvolvimento de pequenos incrementos e a necessidade de se executar todos os testes a cada incremento ou refatoração do software, é essencial uma ferramenta de automação de testes que possa prover agilidade na execução de testes unitários, como por exemplo as ferramentas de automação de testes unitários chamadas *xUnit*.

xUnit é um nome genérico dado à qualquer ferramenta de automação de testes unitários, onde a letra “x” é substituída pela primeira letra do nome da linguagem para a qual se desenvolveu a ferramenta. Seu conceito foi inicialmente concebido por Kent Beck, que desenvolveu *SUnit* para a linguagem *SmallTalk*. Rapidamente esta ideia se difundiu para outras linguagens como o *CPPUnit*, para C++, *JUnit* para Java, *PHPUnit*, para PHP e *PyUnit* para *Python*. Em todas estas ferramentas, os testes são codificados como um módulo do programa seguindo um modelo pré-definido. O módulo de testes invoca os métodos a serem testados, compara as saídas obtidas com as saídas esperadas e, então, exibe o resultado. [10]

Ian Sommerville elenca em [2] alguns benefícios do desenvolvimento focado em teste. O mais notável destes argumentos é de que o desenvolvimento focado em teste ajuda os programadores a obter uma ideia mais clara sobre o quê um pequeno trecho de código, que pode ser um único método ou apenas um trecho de um método, deve realmente fazer quando executado. Para escrever o teste é necessário saber qual é o resultado esperado da execução do trecho de código em questão, e isto faz com que seja mais fácil escrever este trecho de código. [2] Ainda outros benefícios são:

- **Cobertura de código:** partindo do princípio de que todo trecho de código possui no mínimo um teste associado e que todos estes testes sejam executados, preferencialmente pela ferramenta de automação, podemos garantir que todo o código escrito foi executado ao menos uma vez. Também, devido ao fato de que o código é testado à medida em que vai sendo escrito, os defeitos são revelados mais cedo. [2]
- **Teste de regressão:** toda a bateria de testes de um software é construída incrementalmente junto com o desenvolvimento do programa. Este aspecto, aliado à automação oferecida pelas ferramentas *xUnit*, reduz dramaticamente o tempo e o custo de execução dos testes de regressão do sistema. Ao se fazer alguma alteração no sistema, todos os testes anteriores podem ser executados rapidamente, e somente após o software obter aprovação de todos os testes é que uma nova funcionalidade é acrescentada, garantindo assim que esta nova funcionalidade não causou e também não revelou nenhum erro do código que já existia antes. [2]
- ***Debugging* simplificado:** se um teste falhar, a localização do defeito é praticamente óbvia, pois o defeito está no trecho de código associado ao teste que falhou, e no desenvolvimento focado em teste este trecho de código é exatamente o trecho que está sendo implementado no momento. Isso dispensa o uso de ferramentas de localização de *bugs* ou de ferramentas de depuração (*debugging*). [2]
- **Documentação do código:** os testes agem como uma forma de documentação do código, pois eles descrevem o que o código ao qual estão associados deve fazer exatamente. [2]

2.3. Sistemas Embarcados

A definição de um sistema embarcado, segundo Heath [6], é de um sistema desenvolvido para controlar uma função ou um conjunto de funções, baseado em um microprocessador, e que não é projetado para ser programado como os computadores pessoais, que possuem grande interação com o usuário.

Nos sistemas embarcados automotivos, o desenvolvimento é feito utilizando-se o sistema operacional OSEK. O objetivo do OSEK é prover uma arquitetura aberta para as unidades de controle distribuídas do veículo. Ele provê uma interface entre o microcontrolador e a aplicação, agindo com uma plataforma e trazendo uma certa medida de reaproveitamento de código para os sistemas embarcados automotivos. [13]

A comunicação interna entre as unidades de controle do veículo se dá principalmente através do barramento CAN (*Controller Area Network*). O CAN é um protocolo que implementa basicamente as duas camadas mais baixas do modelo OSI (*Open Systems Interconnection*). [15] É um protocolo baseado em mensagens, onde todos os nós da rede recebem todas as mensagens. Cabe a cada nó da rede decidir se a mensagem será descartada, ou será mantida para ser processada. [14]

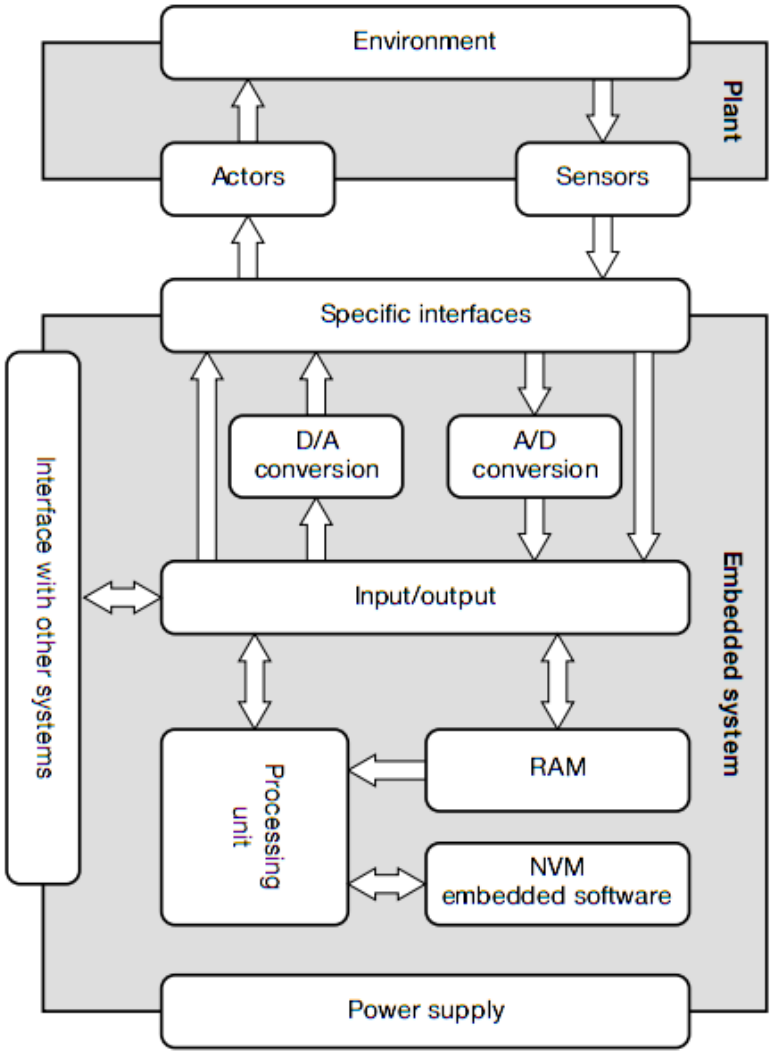

Figura 4 - Diagrama Esquemático Genérico de um Sistema Embarcado [5]

2.3.1. Características e Tipos de Sistemas Embarcados

2.3.2. Barramentos de Comunicação e Sistemas Operacionais

2.3.3. Diferenças entre Sistemas Embarcados e Sistemas Desktop

3. Desenvolvimento do Trabalho

3.1. Características dos Sistemas Embarcados Automotivos

3.1.1. Sistema Operacional *OSEK*

3.1.2. Comunicação: *Controller Area Network* (CAN)

3.1.3. Estrutura do Sistema

3.1.4. Finalidade do Sistema

3.1.5. Normas Técnicas Para o Setor Automotivo

3.2. Desafios para Teste, Validação e Qualidade

3.2.1. Criticidade do Sistema

3.2.2. Alta Especificidade do Sistema

3.2.3. Mudanças de Plataforma

3.2.4. Dependência de Normas Técnicas Rígidas

3.3. Alternativas e Soluções

3.3.1. Melhores Práticas

3.3.2. Viabilidade de Automação

4. Conclusão

5. Referências Bibliográficas

[1] – PRESSMAN, Roger; “Software Engineering: A Practitioner's Approach”; Fifth Edition; New York, 2001.

[2] – SOMMERVILLE, Ian; “Software Engineering”; Ninth Edition; Pearson, Boston, 2011.

[3] – The Institute of Electrical and Electronics Engineers; “IEEE Glossary of Software Engineering Terminology (IEEE Std. 610.12-1990)”; New York, 1990.

[4] – BECK, Kent; “Test Driven Development by Example”; Addison-Wesley, 2002.

[5] – BROEKMAN, Bart; NOTEMBOOM, Edwin; “Testing Embedded Software”; Addison-Wesley, Great Britain, 2003.

[6] – MYERS, Glenford J.; “The Art Of Software Testing”; Second Edition; New Jersey, 2004.

[7] – NETO, Arilo Cláudio Dias; “Introdução ao Teste de Software”; Engenharia de Software Magazine; Ano 1 – 1ª Edição; DevMedia; 2007.

[8] – CAMPOS, Renan Barbosa; “Estudo da implantação do teste de software no ciclo de desenvolvimento visando equipes pequenas”, Trabalho de Conclusão de Curso, UEL, 2008.

[9] – KANER, FALK, NGUYEN.

[10] – BRANDÃO, H.; CAMPOS, J.; FREITAS, T.; GUERREIRO, J.; OLIVEIRA, V.; PINTO, J.; “xUnit – Testes Unitários Automatizados”. 2005. Disponível em: <http://paginas.fe.up.pt/~aaguiar/es/artigos%20finais/es_final_6.pdf>

[11] – MOLINARI, Leonardo. (2008) “Testes funcionais de software”, Visual Books.

[12] – The Institute of Electrical and Electronics Engineers (1986) “IEEE Standard for Software Unit Testing”, New York.

[13] – The Osek Group, (2005) “OSEK/VDX Operating System Specification”, Version 2.2.3.

[14] – PAZUL, Keith. (1999) “Controller Area Network (CAN) Basics”, Microchip Technology.

[15] – LOPES, Chris Andrew Confortini (2009) “CAN – Controller Area Network”, Trabalho de Conclusão de Curso, UEL