

Teste de Software

Prof. Dr. Giovanni Gracioli
giovani@lisha.ufsc.br

ROTA2030
FUNDEP

- Linux com suporte a C e C++
 - Máquina virtual disponibilizada
- Cmocka disponibilizado com o curso
 - Testado em Linux
- Unity test disponibilizado com o curso
 - Testado em Linux
- Google test disponibilizado com o curso
 - Testado em Linux

- Introdução a metodologias de teste de software
- Introdução ao framework de testes cmocka
 - Exemplos
 - Exercícios
- Introdução ao framework de testes Unity
 - Integração com valgrind para detecção de vazamentos de memória
 - Integração com nm para análise de variáveis globais
 - Exemplos
- Introdução ao framework de testes google test para C++
 - Exemplos

- Introdução a metodologias de teste de software
- Introdução ao framework de testes cmocka
 - Exemplos
 - Exercícios
- Introdução ao framework de testes Unity
 - Integração com valgrind para detecção de vazamentos de memória
 - Integração com nm para análise de variáveis globais
 - Exemplos
 - Exercícios
- Introdução ao framework de testes google test para C++
 - Exemplos
 - Exercícios

- Realizar testes é uma tentativa de encontrar bugs (erros) no código
 - As razões para encontrar bugs variam
 - Encontrar todos os bugs é impossível

- Existem vários tipos de testes para diferentes situações
 - Teste exploratório: guiado pela experiência
 - Teste white box: guiado pela estrutura do software
 - Teste black box: guiado pelas especificações funcionais

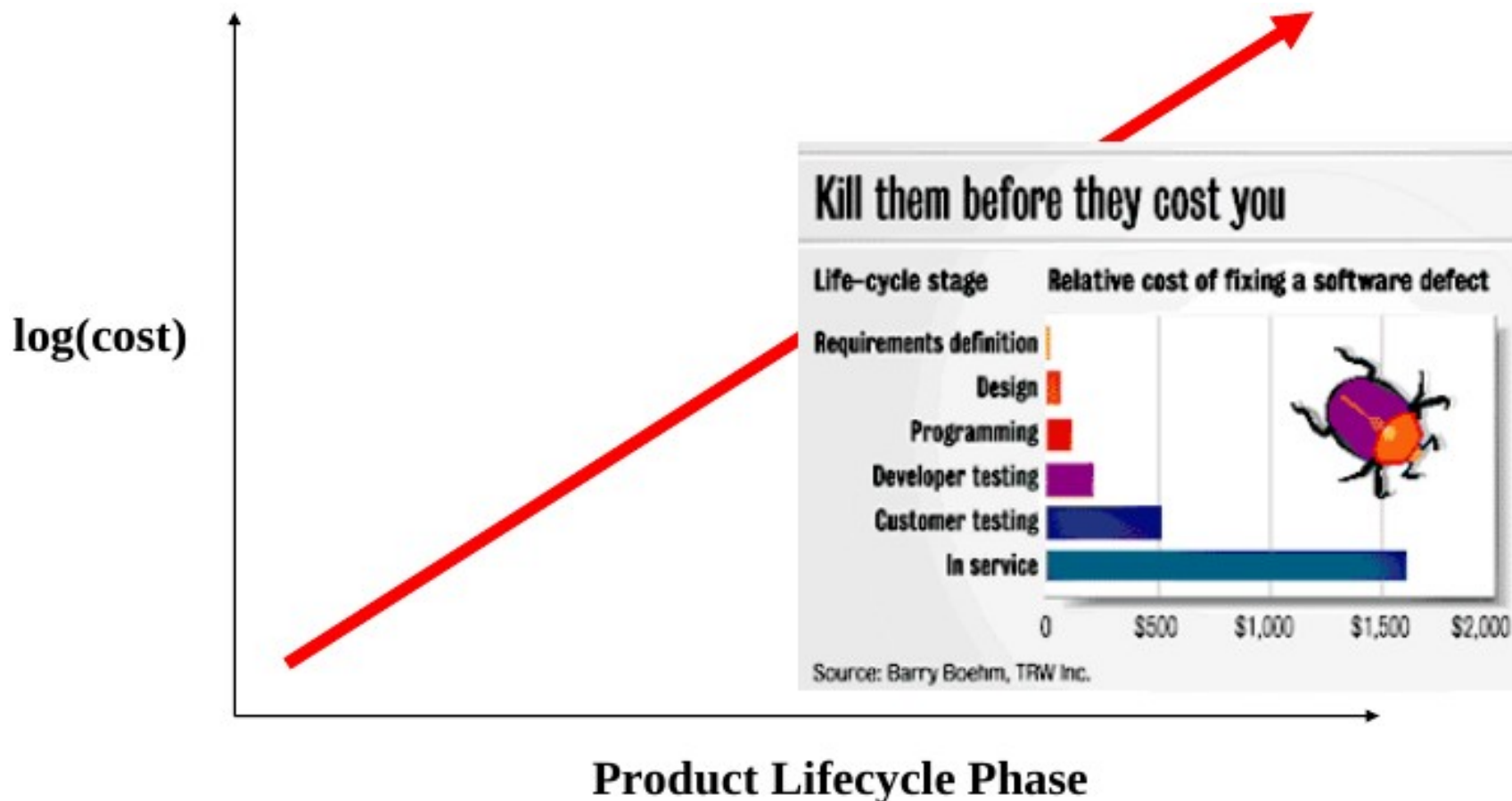
- Vários tipos de testes durante o ciclo de desenvolvimento
 - Teste unitário (desenvolvimento)
 - Teste de subsistemas (integração modular)
 - Testes de integração do sistema
 - Teste de regressão (produto)
 - Teste de aceitação (produto)
 - Teste beta (seleção de alguns clientes para teste)

Porque testamos?

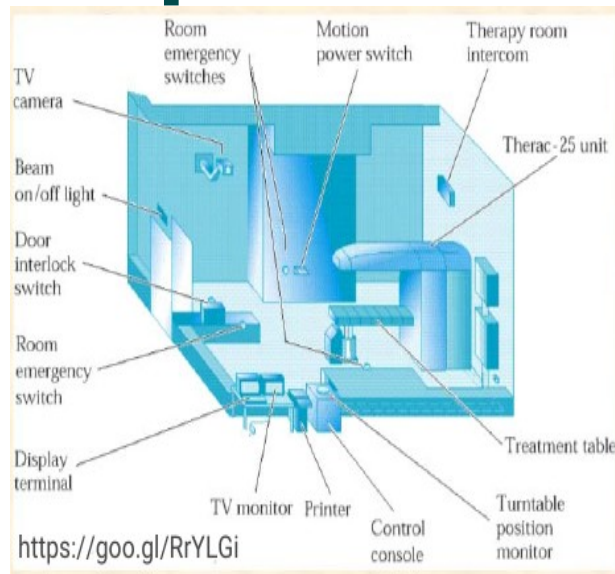
- **Porque alguém nos faz testar**
 - Bom, somente isso não é uma razão satisfatória
 - Não é produtivo e é uma obrigação burocrática imposta
 - Certificação (ex: deve ter 100% de cobertura de código – branch coverage)
- **Queremos encontrar bugs no programa para removê-los**
 - Testes são projetados para garantir que importantes operações funcionem corretamente
 - Usualmente a abordagem é testar até que não se encontre mais bugs
 - Quando bugs “importantes” são corrigidos, produto é enviado
 - Abordagem usada em **desktops**
- **Queremos testar a hipótese que não existem bugs importantes**
 - Testes não devem encontrar erros
 - Continue testando até que seja improvável ter algum bug
 - Se um erro é encontrado, isso indica que houve uma falha no processo de desenvolvimento do software
 - Tipicamente a abordagem usada em **safety-critical systems**

Qual o custo de encontrar e corrigir um erro?

- Custos incluem dinheiro e perda do tempo de mercado
- Custos maiores mais tarde no ciclo de desenvolvimento são quando os lucros são menores

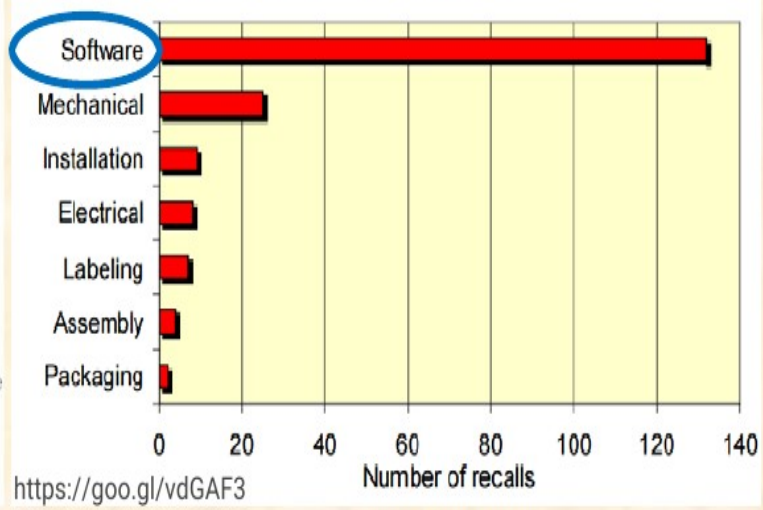


Código ruim existe por toda a parte



Therac 25: 1985-1987
6+ radiation overdoses

Figure 17: Causes of Linear Accelerator Recalls



FDA Recall Data 2002-2012



(1991) Patriot Missile Software Defect / 28 Americans Dead
Workaround: frequent reboots

To keep a Boeing Dreamliner flying, reboot once every 248 days

by Edgar Alvarez | @abcdedgar | May 1st 2015 At 6:34pm



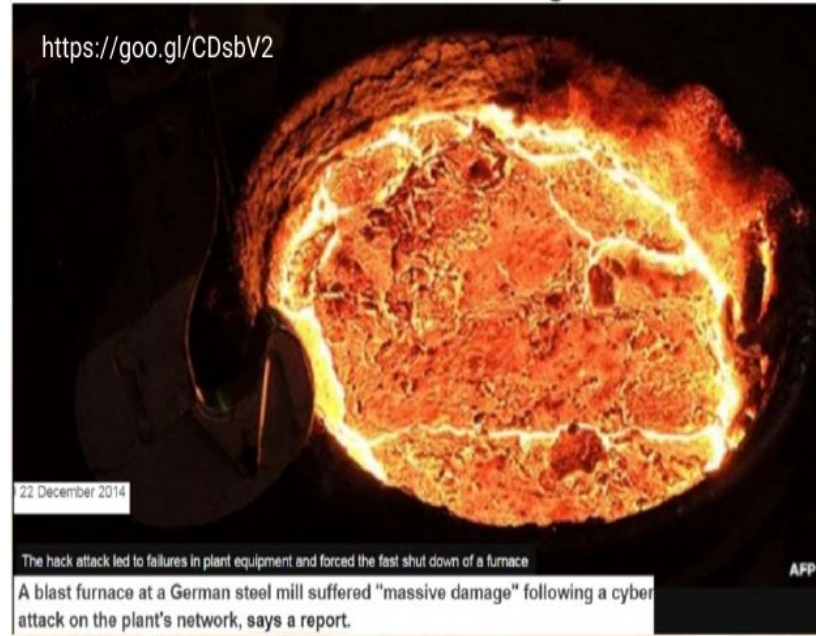
Knight Capital Says Trading Glitch Cost It \$440 Million
AUGUST 2, 2012 9:07 AM
Runaway Trades Spread Turmoil Across Wall St.



Repurposed Bit activates testing mode

Código ruim também leva a problemas de segurança

Hack attack causes 'massive damage' at steel works



Hackers caused power cut in western Ukraine



A power cut in western Ukraine last month was caused by a type of hacking known as "spear-phishing", says the US Department of Homeland Security (DHS).

■ Attacks can affect the physical world

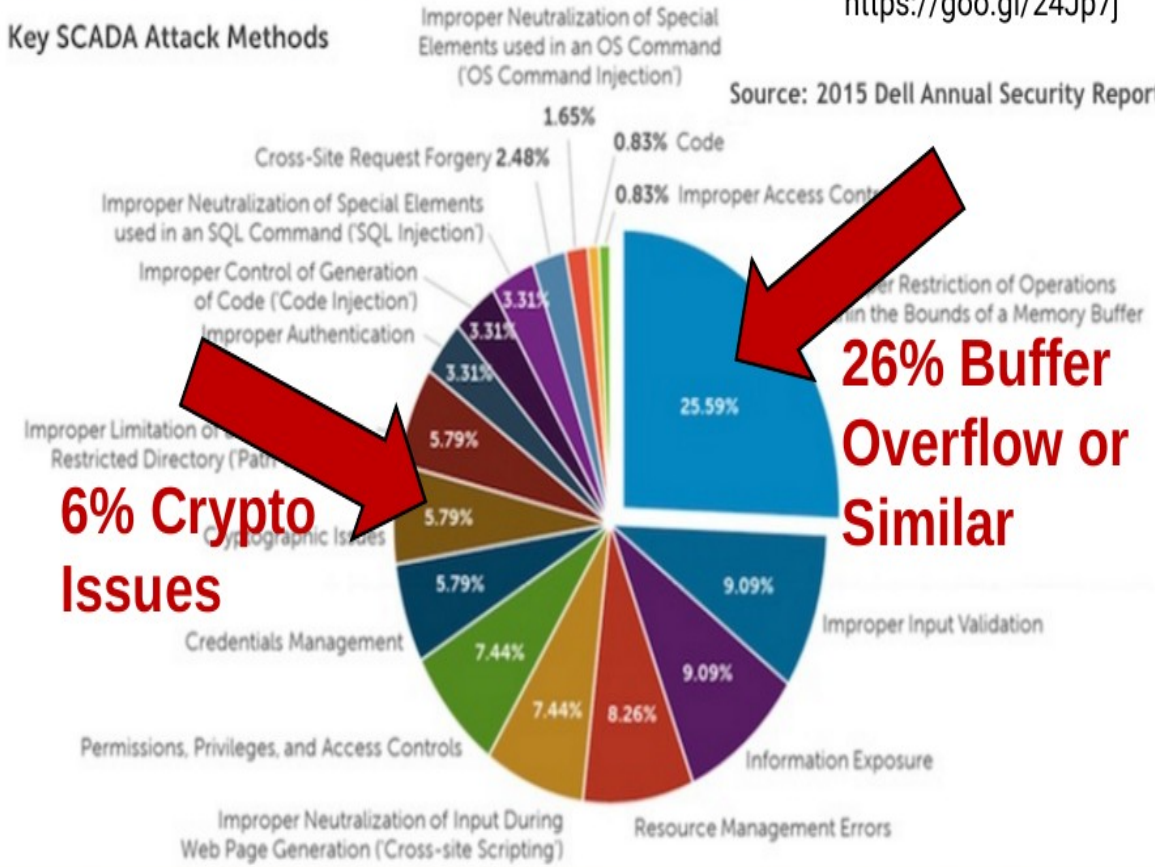
Attacks Against SCADA Systems Doubled in 2014: Dell

By Mike Lennon on April 13, 2015

Dell SonicWALL saw global SCADA attacks increase against its customer base from 91,676 in January 2012 to 163,228 in January 2013, and 675,186 in January 2014.

<https://goo.gl/24Jp7j>

Key SCADA Attack Methods



3. Software assembly for power train ECU

TOY-MDL04983210

After the 4th Steering Committee, rebuilding of engine control and actions for software assembly were started.

(1) Achievements

① Identification of current issues with software assembly Ongoing

- There are C sources for which there is no specification document. (e.g., communication related)
- Specification document and C source do not correspond one-to-one. (e.g., cruise, communication related)

② Activities to improve the spaghetti-like status of engine control application were started.

(Control structure reform has already started in Engine Div. In coordination with this, software structure reform will be carried out. As a first step, it has been decided to transfer two employees from Engine Div. and carry out trial with purge control.)

Because structure design is not being implement, a "spaghetti" state arises, both TMC and suppliers struggle to confirm overall situation

Without care, systems can quickly get too big and complex, and like dinosaurs, will eventually go extinct.



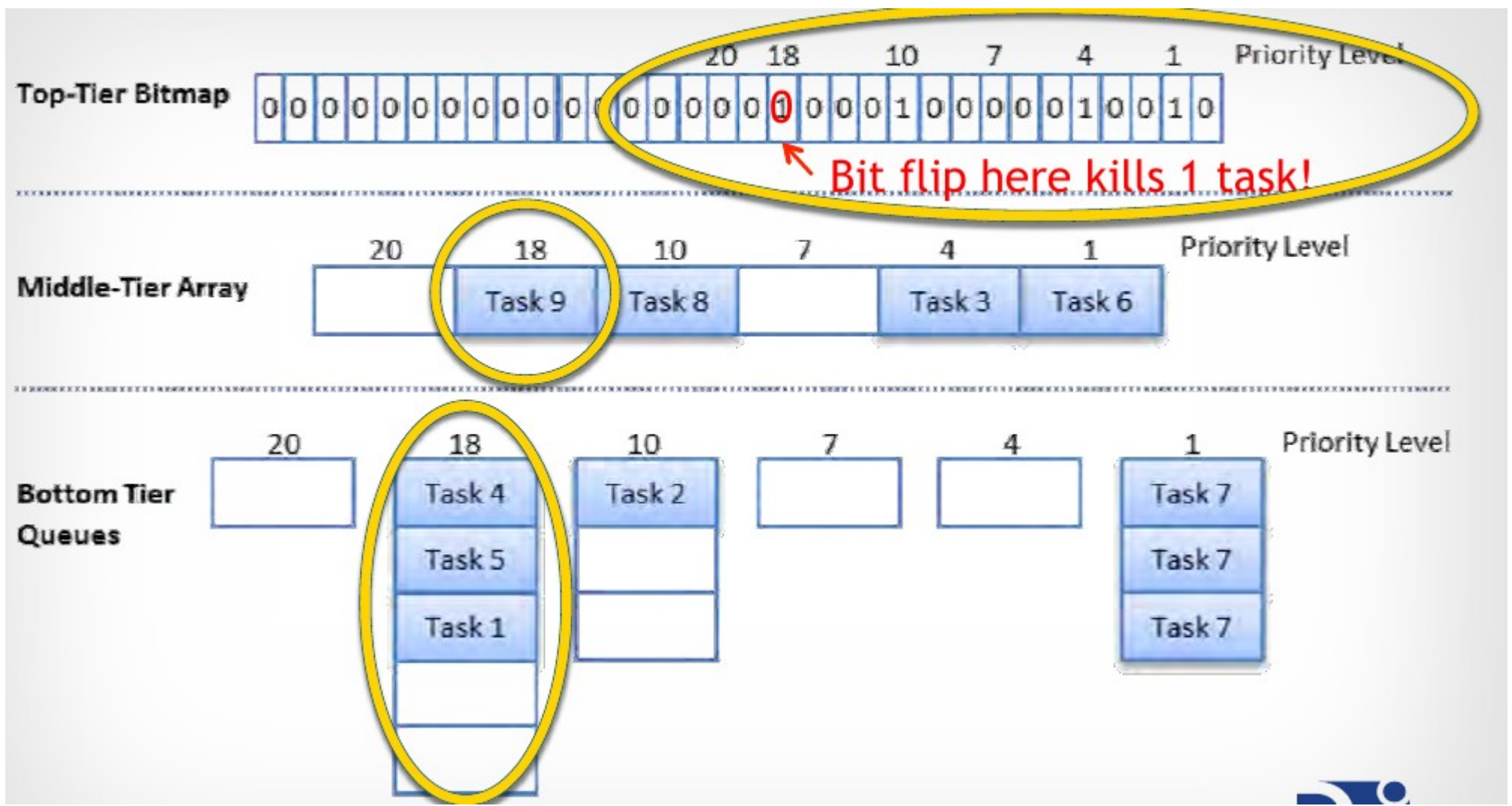
23 TOY-MDL04983219

TOY-MDL04983253
TOY-MDL04983252P-0002

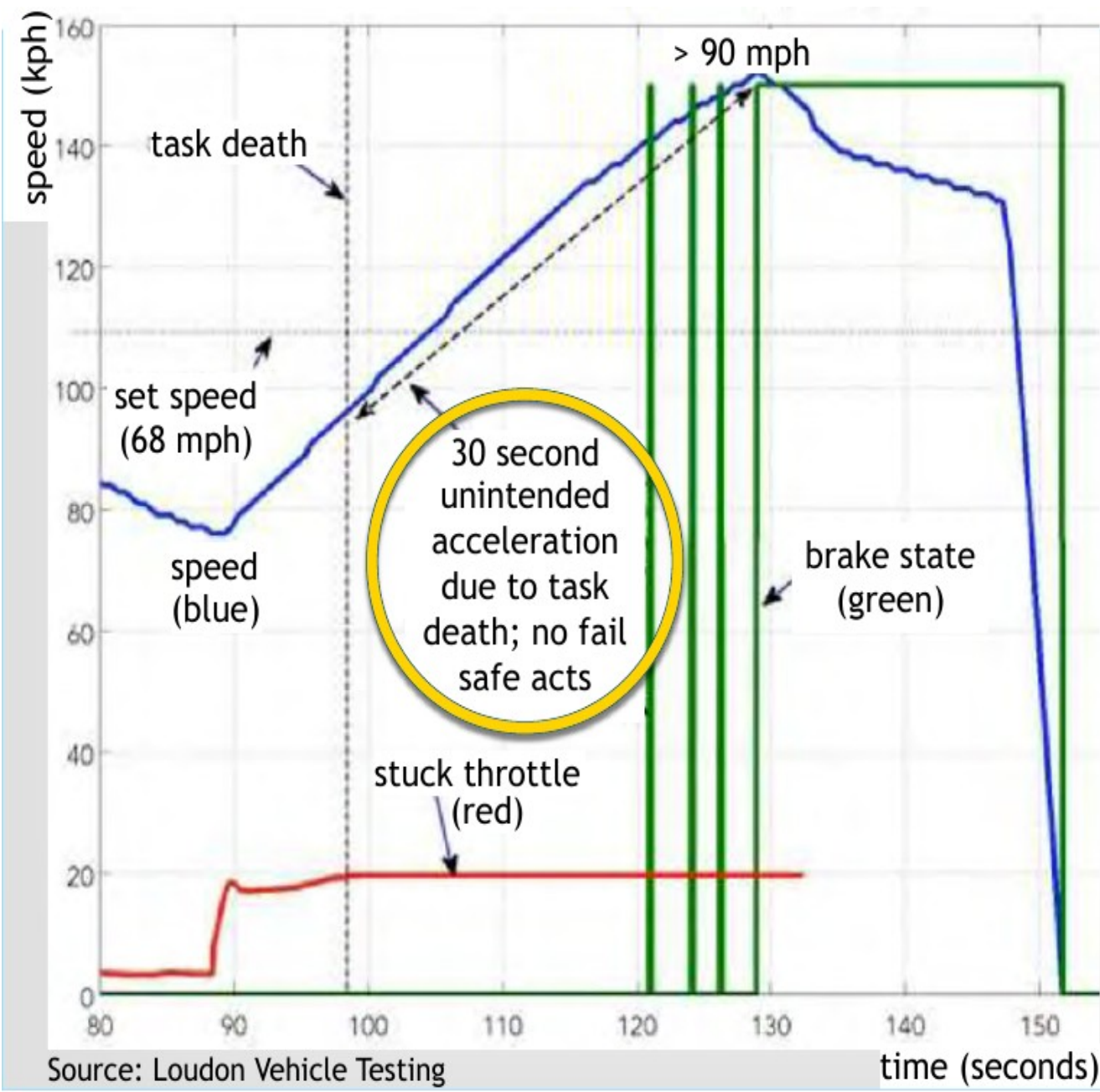
Controle de qualidade do código de um motor automotivo

- Análise da qualidade do código do motor do Camry L4 2005
 - 256K linhas de código fonte (SLOC) em C
 - 80000 violações ao padrão MISRA C
 - Variáveis não inicializadas, cast de tipos não seguros
 - 2272 declarações de variáveis globais
 - 10000 leituras e escritas a variáveis globais
 - Falta de lock em código concorrente
 - Condições de corrida confirmadas (bug)
 - Recursão que usava a pilha toda; sem proteção na pilha
 - Watchdog não efetivo
 - Sem gerenciamento de configuração, bug tracking
- Busca por “koopman toyota”

Exemplo de Falha do SOTR



Exemplo de Falha do SOTR



- Representative of task death in real-world
- Dead task also monitors accelerator pedal, so **loss of throttle control**
✓ Confirmed in tests
- When this task's death begins with brake press (any amount), **driver must fully remove foot from brake to end UA**
✓ Confirmed in tests



Problemas de software estão aumentando?

SOFTWARE NOW TO BLAME FOR 15 PERCENT OF CAR RECALLS

YOU CAN'T JUST HOLD THE HOME AND LOCK BUTTONS TO SOLVE THIS ONE June 2, 2016

Software-Related Vehicle Recalls

■ Number of recalls ■ Number of affected vehicles

J. D. Power Safety HQ / NHTSA's safecar.gov



The research firm J.D. Power, through its Safety IQ application, found that there have been 189 distinct software recalls issued over five years—covering more than 13 million vehicles. These weren't merely interface-related issues either; 141 of these presented a higher risk of crashing.

- Software pode ser o sucesso da empresa ou quebrá-la
- Não existem ferramentas para melhorar o desenvolvimento do software?

Quem está escrevendo o código?

Stack Overflow
Nearly half
are self-taught

"There are many ways to learn how to code. Forty-eight percent of respondents never received a degree in computer science, 33 percent of respondents never took a computer science university course," the report said. "System administrators are most likely to be self-taught (52 percent). Enterprise level services developers are most likely to have an industry certification (13 percent)."



- Em embarcados
 - Maioria é especialista de domínio (eng mecânico, elétrica, etc)
 - Trabalhadores mais jovens já tiveram programação
- Quem paga mais?
 - TI ou indústria que faz o software de controle?

Como chegamos a este ponto?

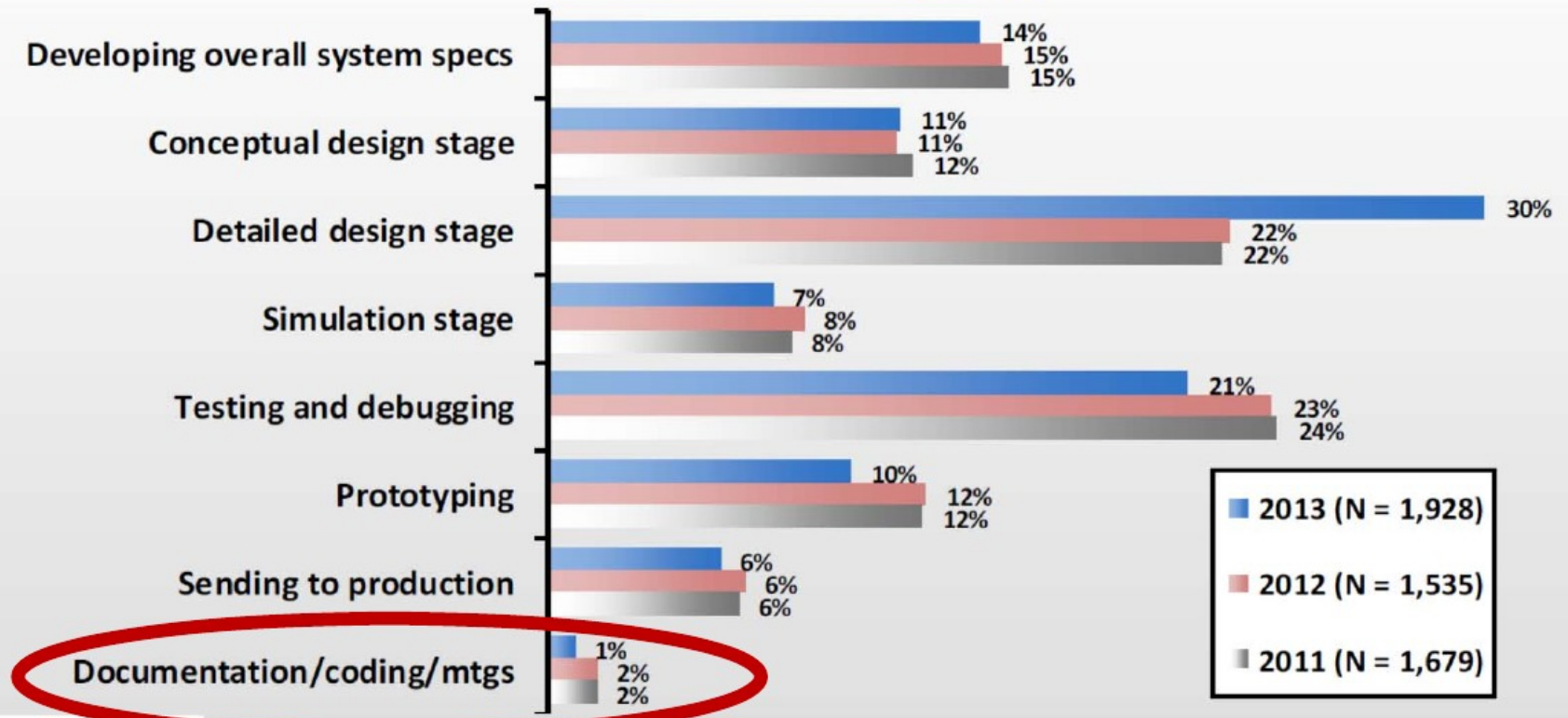
- Resposta simples: uma linha de código ruim por vez
 - A ênfase se deu no **código**, não em **engenharia de software**

- Resposta mais profunda:
 - Falta de habilidades técnicas
 - Falta em habilidades de processo/procedimentos
 - Falta no entendimento de gerenciamento
 - Falta de educação/ensino

Codificar não é o que leva mais tempo..

2013 Embedded Market Study

What percentage of your design time is spent on each of the following stages?



 **designwest**
April 22-25, 2013
McEnergy Convention Center
San Jose, CA

<http://e.ubmelectronics.com/2013EmbeddedStudy/index.html>

Copyright © 2013 by UBM. All rights reserved.

- Habilidades básicas
 - Programação, A/D, D/A, matemática
- Construção de software
 - Modularidade, complexidade, estilo
- Software de tempo real
 - SOTR, Interrupções, escalonamento, concorrência, tempo
- Projeto de sistema robusto
 - Watchdog, stack overflow, exceptions
- Validação
 - Estratégias de teste, cobertura, traceability
 - Revisões pelos pares, métodos formais

Exemplo de habilidades de processo

- Processo de desenvolvimento
 - Modelo, artefatos, métricas
 - Requisitos e estimação
 - Arquitetura, projeto de software, projeto de testes

- Processo de deployment
 - Gerenciamento de configuração
 - Controle de versão
 - Tracking de bugs e resoluções
 - Building, deployment, e defeitos em campo

Exemplo de habilidades especializadas

- Safety
 - Projeto de sistemas críticos
 - Plano de segurança
 - Padrões de segurança
 - Redundância

- Segurança (security)
 - Projeto de sistema seguro
 - Plano de segurança
 - Criptografia, protocolos de segurança, gerenciamento de chaves
 - Secure boot

Exemplo de Equívocos

- **FALSO** “funciona na maioria das vezes” então pode fazer o deployment
 - É mais caro consertar uma bagunça do que dar um passo atrás e fazer direito
- **FALSO** compilar o código rapidamente indica progresso
- **FALSO** testar melhora a qualidade do software
 - Remove os bugs “fáceis” e frequentes
 - Testes simples não acharão bugs relacionados ao tempo
- **FALSO** revisão pelos pares é caro
 - ~10% do custo para encontrar 50-70% dos bugs
- **FALSO** desenvolvimento de software é rápido e barato; qualquer um pode escrever código
 - Leva 1-2 SLOC/pessoa/hora para ter um bom código embarcado
 - Data final fixa + requisitos inflexíveis = falha do projeto

How a dumb software glitch kept thousands from reaching 911

Issue

By Brian Fung October 20, 2014



(New York National Guard / Flickr) <https://goo.gl/0d5ANZ>

Glitches in the touchscreens

with a cars



Air
ern
Airbu

Car syst
By Leo Ki

Some
dange
by tur
contro
the ca



MyFord Touch

Charles Artl
The motor c
a great repl
touchscreen
The compar
offering a s
replaces the

But on April 9, the software responsible for assigning the codes maxed out at a pre-set limit; the counter literally stopped counting at 40 million calls. As a result, the routing system stopped accepting new calls, leading to a bottleneck and a series of cascading failures elsewhere in the 911 infrastructure.



l/4bS5rd om9

,000
on tro
lure.

don't make
ially if the

n the US
m, which
en.

Definição de Teste

- Teste de software engloba
 - Dar entradas ao software (“workload”)
 - Executar um pedaço do software
 - Monitorar o estado do software ou sua saída com propriedades esperadas, como:
 - Conformidade com os requisitos
 - Preservação de invariantes (ex: nunca se aplica freios com acelerador juntos)
 - Faz o “match” dos valores de saída esperados
 - Não tem surpresas, como falhas ou comportamentos inesperados

Definição de Teste

- A ideia geral é tentar encontrar bugs ou erros através da execução do programa
- As seguintes técnicas são úteis, mas não são testes:
 - Model checking
 - Análise estática (checagem do erro do compilador)
 - Revisões do código

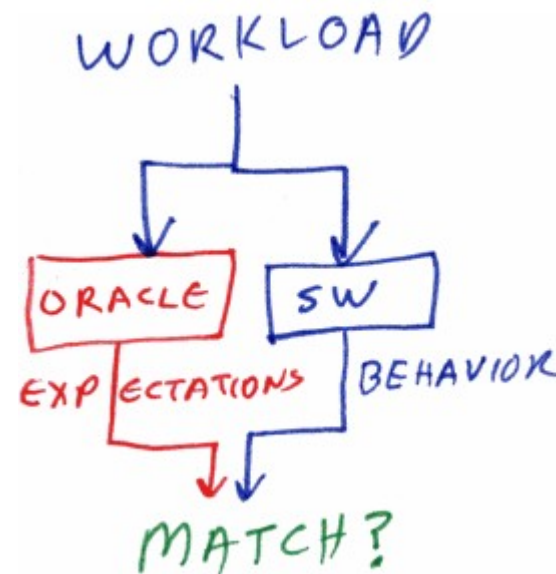
Terminologia

■ Worload

- Entadas aplicadas ao software sendo testado
- Cada teste é executado com um workload específico

■ Comportamento

- Saídas observadas do software sendo testado
- Algumas vezes saídas especiais são adicionadas para melhorar a observabilidade do software (testes para capturar o estado interno)



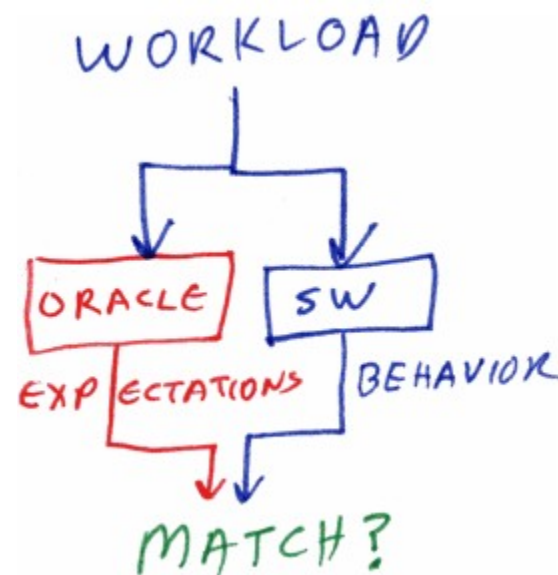
Terminologia

■ Óráculo

- Um modelo que prevê o correto funcionamento
- Comparar o comportamento com a saída do óráculo diz se o teste passou ou falhou

■ O óráculo pode ter diversas formas

- Observador humano
- Diferentes versões do mesmo programa
- Scripts criados de valores estimados baseados no workload
- Lista de invariantes que deve ser verdadeira



O que é um “bug”?

- Abordagem simplista
 - Um “bug” é um defeito no software = software incorreto
 - Um defeito no software viola a especificação

- Abordagem realista
 - Falha ao prover o comportamento requerido
 - Prover o comportamento incorreto
 - Prover um comportamento que não é requerido
 - Falha de conformidade com relação a uma restrição de projeto (tempo por exemplo)
 - Omissão ou falha na especificação/requisitos
 - O software executa conforme planejado, mas tem saída errada

Tipos de Teste

- Estilos de testes
 - Teste smoke
 - Teste exploratório
 - Teste caixa preta (black box)
 - Teste caixa branca (white box)

- Vários tipos de testes durante o ciclo de desenvolvimento
 - **Teste unitário** (desenvolvimento)
 - Teste de subsistemas (integração modular)
 - Testes de integração do sistema
 - Teste de regressão (produto)
 - Teste de aceitação (produto)
 - Teste beta (seleção de alguns clientes para teste)

Teste Smoke

- Teste rápido para ver se o software está operacional
 - A ideia vem do hardware – ligue e veja se tem fumaça
 - Smoke teste para carro: ligue a chave e verifique:
 - Tem barulhos no motor
 - Posso colocar a primeira e andar 5 metros, então freiar e parar
 - As rodas viram pra esquerda e direita enquanto o carro estiver parado
- Bom para pegar erros catastróficos
 - Especialmente depois de novos builds or grandes mudanças
- Não é um teste extensível

Teste Exploratório

- Um pessoa analisa o sistema, procurando por resultados não esperados
 - Pode usar ou não uma documentação do comportamento do sistema como guia
 - Está particularmente interessado em capturar comportamentos “estranhos”
- Vantagens
 - Um “testador” experiente pode encontrar muitos defeitos desta forma
- Desvantagens
 - Não tem uma medida de cobertura do teste
 - Uma pessoa sem experiência não irá encontrar muitos defeitos

Teste Caixa Preta

- Teste projetado com conhecimento do comportamento
 - Sem conhecimento da implementação
 - Frequentemente chamados de teste funcional
- A ideia é testar o que o software faz, não como a função é implementada
 - Exemplo: teste caixa preta do controle de cruzeiro
 - Testa operação em várias velocidades
 - Testa operação com velocidade baixa e alta
 - Mas sem ter nenhum conhecimento sobre o controle
- Vantagens
 - Testa o comportamento final do software
 - Pode ser escrito independentemente do projeto de sw
 - Usado para testar diferentes implementações
- Desvantagens
 - Não necessariamente conhece os limites
 - Pode ser difícil cobrir todas as partes do código

Teste Caixa Branca

- Testes projetados com conhecimento do projeto de software
 - Frequentemente chamados de teste estrutural
- Ideia é exercitar o software conhecendo como ele é projetado
 - Exemplo: teste caixa branca do controle de cruzeiro
 - Testa a operação em cada ponto do loop da tabela de controle
 - Testes que passam por todas as condições/desvios
- Vantagens
 - Ajuda a cobertura do código
 - Bom para atingir os casos limites e casos especiais
- Desvantagens
 - Testes baseados no projeto podem perder a “big picture” do sistema e seus problemas
 - Testes precisam ser modificados caso o código também seja
 - Difícil de testar o código que não está lá (funcionalidade)

Cobertura do Código

- Cobertura é a noção de o quão completo o teste foi feito
 - Usualmente uma porcentagem (ex: 97% de cobertura de desvios)
- Cobertura no teste caixa branca
 - Porcentagem de condições testadas
 - Porcentagem de entradas em tabelas usadas para computação
- Cobertura no teste caixa preta
 - Porcentagem de requisitos testados
 - Não testa comportamento extra que não é para estar
- **Importante:** 100% cobertura não significa 100% testado

- Uma unidade é formada por algumas poucas linhas de código
 - Usualmente criada por um único desenvolvedor
 - Usualmente testada pelo programador que escreveu o código
- Propósito do teste unitário
 - Tentar encontrar todos os defeitos “óbvios”
 - Pode ser feito antes ou/e depois da revisão do código
- Abordagens (exploratório e caixa branca)
 - Exploratório ajuda os programadores a construir intuições e entender o código
 - Caixa branca para garantir que todas as partes do código foram testadas (100% dos estados por ex)
 - Alguns testes caixa preta para garantir a “sanidade”

Teste de Subsistema

- Um subsistema é um componente de software relativamente completo
 - Ex: software de controle do motor
 - Usualmente criado por uma equipe
 - Usualmente testado por uma combinação de programadores e “testadores” independentes
- Objetivo
 - Tentar encontrar todos os defeitos “óbvios”
 - Pode ser feito antes ou/e depois da revisão do código
- Abordagens (maioria caixa branca; alguns caixa preta)
 - Caixa branca para cobertura de código
 - Caixa preta deve testar o comportamento da interface comparando com a especificação para evitar “surpresas” na integração do sistema
 - Teste smoke é útil para garantir que modificações não quebram a integração

Teste de Integração do Sistema

- Completo sistema com múltiplos componentes
 - Ex: carro
 - Criado por múltiplas equipes organizadas em diferentes grupos
 - Usualmente testado por organizações de testes independentes
- Objetivos
 - Garantir que os componentes e que o comportamento do sistema estejam corretos
 - Encontrar falhas na especificação da interface que causam problemas ao sistema
 - Encontrar comportamentos não esperados
- Abordagens (maior parte caixa preta)
 - Garantir o atendimento dos requisitos
 - Caixa branca para testar as interfaces
 - Teste exploratório para ajudar a encontrar interações estranhas dos componentes
 - Teste smoke para componentes independentes

Teste de Regressão

- O teste de regressão garante que o bug que foi corrigido se mantém corrigido
 - Frequentemente se baseia no teste que foi usado para reproduzir o erro antes da correção
- Asseguram que as funcionalidades não foram quebradas depois de uma mudança
 - Subconjunto dos testes de integração e unitário (ex: nightly build e ciclos de testes)
- Objetivo
 - Houve uma mudança no software? Ele foi quebrado?
 - No caso de um ciclo iterativo de desenvolvimento, assegurar que o sistema funciona no fim de cada ciclo
- Abordagens
 - Combinação dos testes que tem boa cobertura e tempo
 - Se concentra em áreas que tiveram bugs anteriormente

Teste de Aceitação

- Assegura que o sistema possui todas as funcionalidades prometidas
 - Realizado por clientes ou delegados a terceiros
 - Pode ainda envolver uma entidade certificadora ou um observador independente
- Objetivo
 - O sistema atende a todos os requisitos?
 - Pode ser aplicado a todo o sistema (hardware), não somente ao software
 - Usado para determinar a qualidade do software (se poucos ou nenhum bug é encontrado, o software tem qualidade)
- Abordagem
 - Caixa preta do sistema e 100% dos requisitos de alto nível

- Uma versão beta é o software completo perto de ser finalizado
 - Teoricamente todos os bugs foram corrigidos
 - Ideia é fornecer a versão aos usuários e verificar se existem grandes problemas
- Objetivo
 - Ver se o software é bom o bastante, para uma comunidade pequena e amigável
- Abordagens
 - Exploratória
- Defeitos significam que existe um “buraco” em algum lugar
 - Se os processos são bons, provavelmente é falha nos requisitos
 - Se é somente um bug, porque não foi visto nas revisões e testes anteriores?

Número de pessoas envolvidas em teste

- A taxa de pessoas de teste x desenvolvedores depende da aplicação alvo
 - Desenvolvimento web: 1 “testador” por 5-10 desenvolvedores
 - Microsoft: 1 “testador” por 1 desenvolvedor
 - Software safety-critical: 4-5 “testadores” por 1 desenvolvedor
 - Horas de testes e horas de desenvolvimento
 - Tempo gasto em revisão e teste unitário conta como teste, mesmo se feito por desenvolvedores
- Ainda tem mais
 - Teste de aceitação externo
 - Teste beta
- Custo de teste e validação é frequentemente ~50% do custo do software
- Para um bom sistema embarcado, custo de teste chega a 60%
- Para safety-critical systems chega a 80%!

Discussão e comentários

- Como é a realidade de testes na sua empresa?
- É suficiente para desenvolvimento de sistemas críticos que devem ser certificados?
- Como melhorar?

- Introdução a metodologias de teste de software
- **Introdução ao framework de testes cmocka**
 - Exemplos
 - Exercícios
- Introdução ao framework de testes Unity
 - Integração com valgrind para detecção de vazamentos de memória
 - Integração com nm para análise de variáveis globais
 - Exemplos
 - Exercícios
- Introdução ao framework de testes google test para C++
 - Exemplos
 - Exercícios

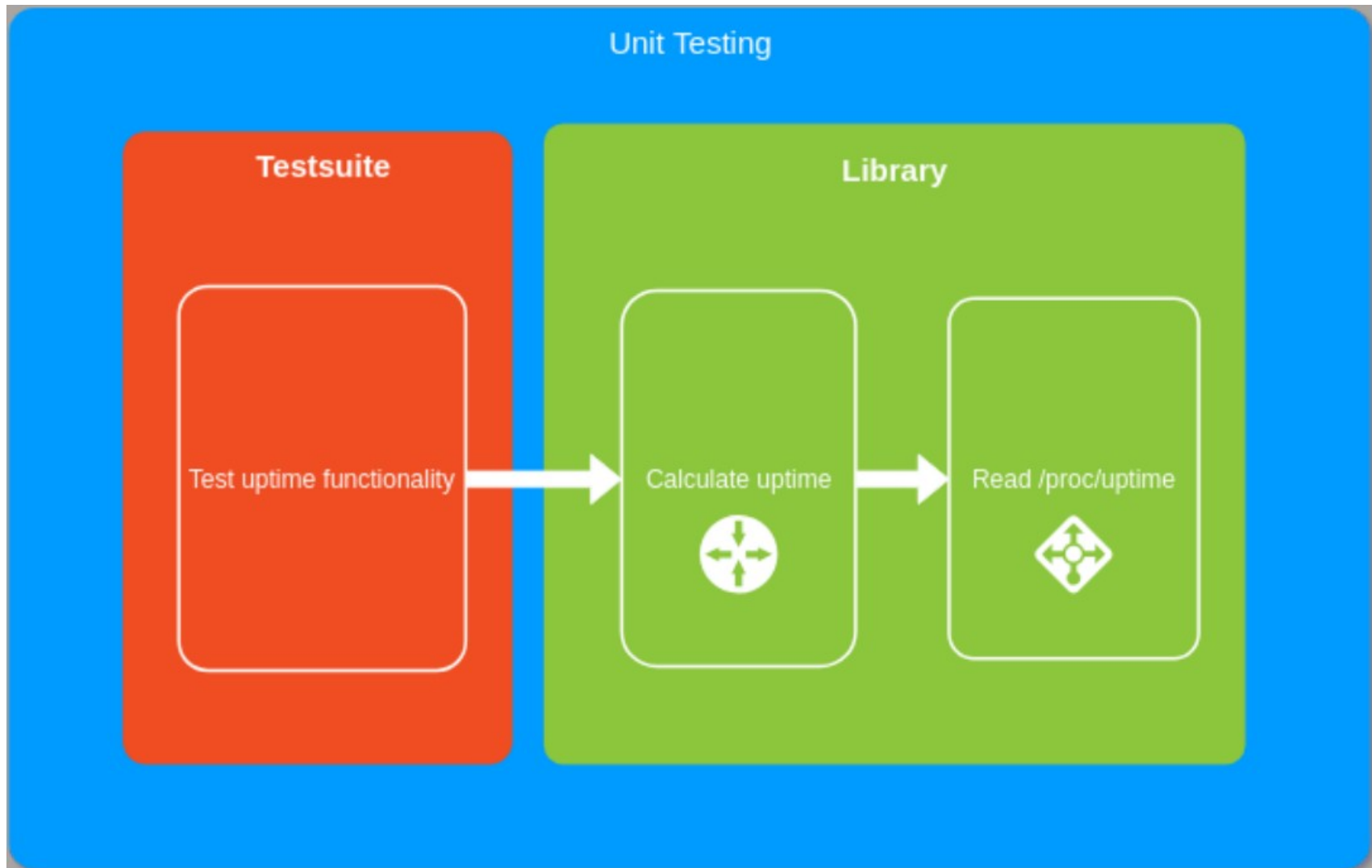
- O objetivo do projeto é prover um framework de testes unitários para a linguagem C
 - Suporte para diferentes plataformas
 - Suporte a diferentes sistemas operacionais
 - Depende somente da biblioteca padrão C
 - Suporte a diferentes compiladores
 - Suporte a sistemas embarcados

- Disponível online em <https://cmocka.org/>
 - `sudo apt-get install libcmocka-dev`
 - API disponível em <https://api.cmocka.org/>
- É o sucessor do cmockery originalmente desenvolvido pela google
- Baseado no conceito de “mock objects”
 - to mock = to imitate something
 - São objetos de simulação que imitam a implementação do objeto real
 - Úteis para estimular as dependências de uma interface e testar uma interface em isolamento

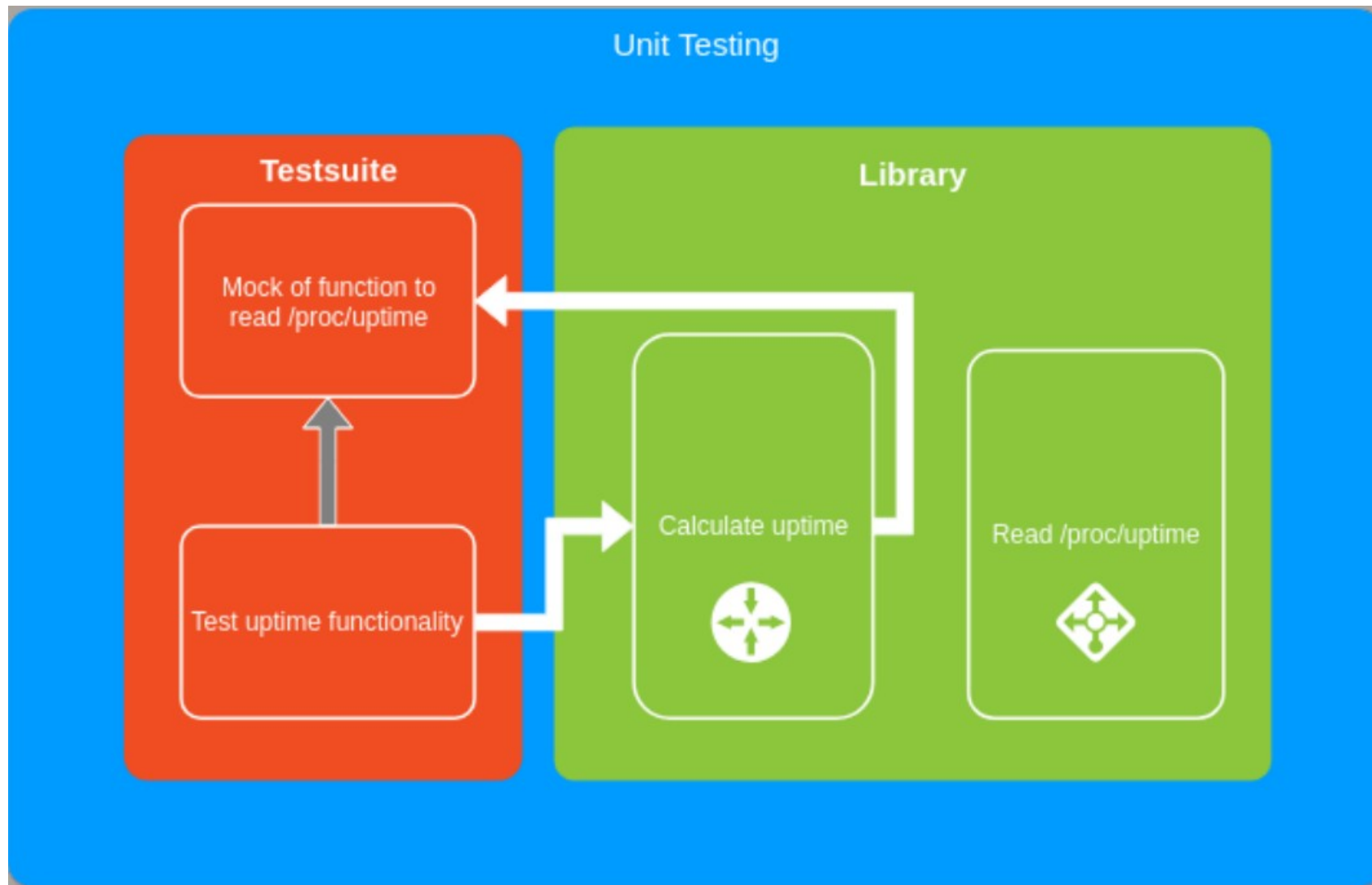
■ Mock objects

- API específica para criá-los
- Ao invés de chamar as funções reais, os testes chamam as funções mock
- As funções mock verificam que os parâmetros e ordem foram corretamente chamados
- Útil para isolar o comportamento de funções complexas ou dependentes de hardware

■ Teste unitário tradicional



■ Teste unitário com mocking



- Como funciona o “mocking”
 - Usa uma função wrapper para um símbolo
`ld --wrap="símbolo"`
 - Suportado por diversos linkers, como `ld`, `ld.bfd`, `ld.gold` e `llvm-ld`
- Se a função testada for
`int uptime(double *uptime_secs, double *idle_secs)`
- A função mock chamada será

```
int __wrap_uptime(double *uptime_secs, double
*idle_secs) {
    .....
}
```

- O símbolo da função original `uptime()` será renomeado para `__real_uptime`
- O símbolo `uptime` é renomeado para `__wrap_uptime`
- Dessa forma ainda é possível chamar a função original

■ Test fixtures

- Funções de setup e teardown que podem ser compartilhadas por múltiplos casos de teste
- Provêem funcionalidades comuns aos testes
- Setup configura o ambiente antes de executar os testes
- Teardown destrói o ambiente de testes após sua execução

■ Grupos

- Suporte a grupos de testes

- Tem mecanismos para tratamento de exceções
 - Capaz de recuperar o estado do teste quando uma exceção acontecer, como sigfault
 - Tratamento para SIGKILL, SIGSEGV, etc
- Formatos de saída
 - Formato próprio indicando falha ou sucesso dos testes
 - Suporta também xUnit XML (suportado pelo Jenkins), Subunit (usado pelo Samba) e test anything protocol (protocolo que permite a comunicação entre testes de unidade e um equipamento de teste)

- Tem suporte para testar
 - Vazamento de memória sem uso de ferramentas auxiliares como valgrind
 - Buffer overflow e underflow
- Não suporta multithread
 - Não é seguro executar testes em programas multithread devido a variáveis globais internas
 - Rodar como “CMOCKA_TEST_ABORT='1' ./my_threading_test”
 - Irá abortar o teste em caso de falhas

■ Integração com sistemas embarcados

- Algumas plataformas de sistemas embarcados não possuem as definições de tipos necessárias pelo framework de teste
- Para contornar o problema, é possível criar um arquivo header chamado `cmocka_platform.h` com as definições e tipos que faltam
- Depois pode passar o arquivo quando compilar o framework de teste
cmake
-DCMOCKA_PLATFORM_INCLUDE=/home/compiler/my/include_directory ..

- Conjunto de asserts
- Boolean
 - `assert_true(x)`
 - `assert_false(x)`
- Ponteiros
 - `assert_non_null(ptr)`
 - `assert_null(ptr)`
 - `assert_ptr_equal(ptr1, ptr2)`
 - `assert_ptr_not_equal(ptr1, ptr2)`

■ Inteiros

- `assert_int_equal(a, b)`
- `assert_int_not_equal(a, b)`
- `assert_in_range(valor, minimo, maximo)`
- `assert_not_in_range(valor, minimo, maximo)`

■ Floats

- `assert_float_equal(a, b, margem)`
- `assert_float_not_equal(a, b, margem)`

■ Código de retorno

- `assert_return_code(rc, error)`

■ Strings

- `assert_string_equal(a, b)`
- `assert_string_not_equal(a, b)`

■ Comparação de memória

- `assert_memory_equal(ptr1, ptr2, tamanho)`
- `assert_memory_not_equal(ptr1, ptr2, tamanho)`

■ Conjunto de inteiros

- `assert_in_set(valor, valores[], tamanho)`
- `assert_not_in_set(valor, valores[], tamanho)`

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
/* A test case that does nothing and succeeds. */
/* O teste não retorna nada e recebe **void */
static void null_test_success(void **state) {
    (void) state; /* unused */
}
int main(void) {
    const struct CMUnitTest tests[] = {
        cmocka_unit_test(null_test_success),
    };
    return cmocka_run_group_tests(tests, NULL, NULL);
}
```

cmocka – Exemplo com assert

```
static void integer_failure(void **state) {  
    int i = 4;  
  
    assert_int_equal(i, 5);  
}
```

- Escrever uma função mocking
 - O framework disponibiliza 3 alternativas para serem usadas dentro de uma função mocking
 - Checagem de parâmetros
 - Armazenada valores esperados dos parâmetros para comparação quando a função de teste for chamada
 - Mocking
 - Ordem de chamada
 - Verifica a ordem que as funções são chamadas
- Documentação em
 - <https://api.cmocka.org/modules.html>

Verificar os parâmetros

- Funções expect_*
 - Exemplo:

`expect_value(função, parâmetro, valor)`

Ir  adicionar um evento para testar se o par metro tem o valor esperado quando a fun  o de teste for chamada

`expected_value()`   chamada por `check_expected()`

Verificar os parâmetros

```
int __wrap_mock(char *name) {  
    check_expected(name);  
}  
  
void test_foo(void **state) {  
    expect_string(__wrap_mock, name, "wurst");  
    foo("wurstbrot");  
}
```

Verificar os parâmetros

- Outros exemplos de expect_*
 - expect_memory(função, parâmetro, *ptr, tamanho)
 - expect_in_range(...)
 - expect_not_in_range(...)

- Descrição completa em
 - https://api.cmocka.org/group__cmocka__param.html

Ordem de chamada

- `expect_function_call()`
 - Coloca na pilha de chamadas esperadas a função que deverá ser chamada para comparação
- `function_called()`
 - Retira da pilha de chamadas esperadas

```
void chef_sing(void);


void code_under_test()
{
    chef_sing();
}

void some_test(void **state)
{
    expect_function_call(chef_sing);
    code_under_test();
}
```

```
void chef_sing()
{
    function_called();
}
```



```
int __wrap_mock(char *name) {  
    return mock_type(int);  
}  
  
void test_foo(void **state, {  
    int rc;  
  
    will_return(__wrap_mock, 0);  
  
    rc = foo("wurstbrot");  
    assert_return_code(rc, errno);  
}
```



Função retorna
o valor definido
em will_return

Vazamento de memória

- Substituir as chamadas para `malloc()`, `calloc()` e `free()` por `test_malloc()`, `test_calloc()` e `test_free()`
- Toda vez que `test_free()` é chamada, há um teste que verifica se há corrupção dos dados
- Todos os blocos de memória alocados com `test_*`() são mantidos pela biblioteca do `cmocka`
- Quando o teste finaliza e há algum bloco ainda alocado, sinaliza o vazamento de memória
 - https://api.cmocka.org/group__cmocka__alloc.html

Exemplo de Teste Unitário

- Código disponibilizado pelo curso em
 - teste_de_software/c
 - mocka/cmocka-exemplos/cmocka-unit-test-example
- Dois diretórios
 - src tem o código original sem nenhum teste
 - test tem o código do teste unitário
 - cmocka é disponibilizado junto com o diretório e é compilado antes do teste através do comando “make”

Setup e teardown

- As funções chamadas para inicializar e finalizar os testes
 - Exemplo em `teste_de_software/cmocka/cmocka-exemplos/cmocka-unit-test-setup-teardown-example/test`

```
/* These functions will be used to initialize
   and clean resources up after each test run */
int setup (void ** state) {
    return 0;
}

int teardown (void ** state) {
    return 0;
}
```

- Exemplo online
<https://www.samlewis.me/2016/09/embedded-unit-testing-with-cmocka/>
- Explora o uso do mocking para testar a lógica do código, sem ter acesso ao hardware
- Processo de teste unitário em um sensor de temperatura
 - TI TMP101 → sensor de temperatura através de I2C
- Objetivo é usar o cmocka para testar a API que que lê do sensor

Exemplo: sensor de temperatura

■ Código para leitura do sensor

```
#include "tmp101.h"

static const float TMP_BIT_RESOLUTION = 0.0625;

float tmp101_get_temperature(void)
{
    // Need to set the TMP101 pointer register to point to the temp register
    uint8_t pointer_address = 0;
    i2c_transmit_blocking(TMP101_ADDRESS, 0, &pointer_address, 1);

    // The TMP101 stores 12 bit samples that are retrieved in two byte blocks
    uint8_t data[2];
    i2c_read_blocking(TMP101_ADDRESS, 0, &data[0], 2);

    // the 1st byte is bits 12 to 4 of the sample and the 2nd byte is bits 4 to 0
    // see page 16 of the TMP_101 datasheet
    uint16_t temperature_bits = (data[0] << 4) | (data[1] >> 4);

    // The 12 bit sample is represented using 2s complement, for simplicity
    // (and because there's no 12 bit int representation), scale up the sample
    // to 16 bits and adjust the bit resolution when converting later
    int16_t temperature = temperature_bits << 4;

    // shift the sample back down and convert by the TMP_101 bit resolution
    return ((temperature / 16) * 0.0625f);
}
```


Exemplo: sensor de temperatura

- Tabela disponibilizada pela TI que relaciona os 12 bits de saída do sensor com a temperatura

Table 1. Temperature Data Format		
TEMPERATURE (°C)	DIGITAL OUTPUT	
	BINARY	HEX
128	0111 1111 1111	7FF
127.9375	0111 1111 1111	7FF
100	0110 0100 0000	640
80	0101 0000 0000	500
75	0100 1011 0000	4B0
50	0011 0010 0000	320
25	0001 1001 0000	190
0.25	0000 0000 0100	004
0	0000 0000 0000	000
-0.25	1111 1111 1100	FFC
-25	1110 0111 0000	E70
-55	1100 1001 0000	C90
-128	1000 0000 0000	800

- Queremos controlar o valor lido do sensor pela função `i2c_read_blocking`

Exemplo: sensor de temperatura

- Uma forma, sem usar teste unitário, seria:

```
void i2c_read_blocking(uint8_t address, uint8_t offset, uint8_t* pData, uint8_t data_size)
{
    #ifdef TESTING
    return DUMMY_VALUE
    #endif

    //normal i2c logic here
}
```

- Qual o problema dessa abordagem?

Exemplo: sensor de temperatura

- Usando o cmocka, primeiramente se define as mock functions

```
void __wrap_i2c_transmit_blocking(uint8_t address, uint8_t offset, uint8_t* data,
uint8_t data_size)
{
    // allows the calling test to check if the supplied parameters are as expected
    check_expected(address);
    check_expected(offset);
}

void __wrap_i2c_read_blocking(uint8_t address, uint8_t offset, uint8_t* pData, uint8_t
data_size)
{
    // allow the calling test to specify the data it wants back
    // and copy it back out
    for(int i=0; i < data_size; i++) {
        pData[i] = mock_type(uint8_t);
    }
}
```

- Linker recebe --wrap=i2c_read_blocking e --wrap=i2c_transmit_blocking

Exemplo: sensor de temperatura

■ Escrita do teste

```
static void test_negative_temperature(void **state)
{
    will_return(__wrap_i2c_read_blocking, 0b11100111);
    will_return(__wrap_i2c_read_blocking, 0b00000000);

    assert_true(tmp101_get_temperature() == -25);
}
```

- As duas chamadas a `will_return` setam o que a função `i2c_read_blocking` escreveria no vetor `pData`
 - Data sheet do sensor define que os 12 bits de dados do sensor são retornados em dois bytes
 - Define também o valor correspondente a -25 graus
 - Mais fácil testar a amplitude de temperaturas

Exemplo: sensor de temperatura

- Vejamos o código
- Compartilhado com o curso
`teste_de_software/cmocka/cmocka-exemplos/cm`
`ocka-embedded-example`
- Diretório `src` tem o código sem testes
- Diretório `test` tem o código com os testes do `cmocka`
- Para rodar, entrar em `test` e digitar “make”
 - Irá descompactar e compilar o `cmocka`
 - Irá compilar o teste com o código original do sensor e com o `cmocka`

- Usar o cmocka para testar parte do código do sistema de ar condicionado
- Disponível no moodle

- Digamos que desenvolvemos um código de um banco e devemos testar se o dinheiro foi corretamente depositado
- Não temos o código função `deposit()` pois é propriedade de terceiros:
 - `int deposit(int money, const char* bank);`
 - A função tem 3 requisitos:
 1. Aceitar depósito somente se a conta é válida
 2. Somente aceitar valores maiores que 100
 3. Não aceitar depósito no banco chamado “WEG”
- Gostaríamos de testar o código de `production_code()` para diversas entradas

- Código disponível em `teste_de_software/cmocka/cmocka-exemplos/exercicio-banco`
 - Diretório `src` contém o código da aplicação
 - Diretório `test` contém o exercício que consiste em escrever duas funções de teste
 - Diretório `test-solution` contém um exemplo de solução que veremos depois (não olhar)

- Introdução a metodologias de teste de software
- Introdução ao framework de testes cmocka
 - Exemplos
 - Exercícios
- **Introdução ao framework de testes Unity**
 - Integração com valgrind para detecção de vazamentos de memória
 - Integração com nm para análise de variáveis globais
 - Exemplos
 - Exercícios
- Introdução ao framework de testes google test para C++
 - Exemplos
 - Exercícios

- Teste unitário de código aberto
 - <https://github.com/ThrowTheSwitch/Unity>
- Documentação no projeto do github
 - <https://github.com/ThrowTheSwitch/Unity/blob/master/docs/UnityGettingStartedGuide.md>
- API para teste unitário na linguagem C
 - Pequena e funcional
- Projeto para ser simples e executar em qualquer plataforma

■ São 3 arquivos principais

- **unity.c**

- Implementação de algumas funções
- Deve ser compilado e ligado na aplicação de teste

- **unity.h**

- Declaração das macros que fazem os testes unitários
- Deve ser adicionado na aplicação de teste através de `#include "unity.h"`

- **unity_internals.h**

- Definições e funções de uso interno
- Tem alguns `#ifdefs` para verificar tipos e tamanhos com o intuito de ser cross-plataforma

Como criar um teste

- Geralmente é um arquivo de teste para cada módulo a ser testado
 - O arquivo de teste deve incluir `unity.h`
- O arquivo de teste deve incluir duas funções
 - `setUp()`
 - Contém qualquer coisa que deve ser executada antes do teste
 - `tearDown()`
 - Qualquer coisa que deve ser executada depois do teste
 - Pode deixar uma ou as duas em branco se desejar
- O resto do arquivo é um conjunto de testes
 - As funções de testes podem iniciar com o nome `test_`

Como criar um teste

- No final do arquivo terá a função main()
 - Chamará a macro UNITY_BEGIN()
 - Chamará RUN_TEST para cada teste
 - Finalizará com UNITY_END()
- Existe um script que ajuda a gerar os testes
 - https://github.com/ThrowTheSwitch/Unity/blob/master/uto/generate_test_runner.rb
 - Cria a função main e as chamadas dos testes

O código de teste

```
#include "unity.h"
#include "file_to_test.h"
void setUp(void) {
    // set stuff up here
}
void tearDown(void) {
    // clean stuff up here
}
void test_function_should_doBlahAndBlah(void) {
    //test stuff
}
void test_function_should_doAlsoDoBlah(void) {
    //more test stuff
}
// not needed when using generate_test_runner.rb
int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_function_should_doBlahAndBlah);
    RUN_TEST(test_function_should_doAlsoDoBlah);
    return UNITY_END();
}
```

- Uma coleção de assertions (assert()) macro em c)
- Unity organize essas assertions e separa do código fonte
- Convenção de nome e parâmetros
 - TEST_ASSERT_X({modifiers}, {expected}, actual, {size/count})
 - O assert mais simples usa apenas “actual”
 - Actual: é o valor sendo testado e é obrigatório
 - Modifiers: são máscaras, ranges, bit flags, delta em ponto flutuante
 - Expected: é o valor esperado comparado ao valor actual
 - Size/count: refere-se ao tamanho de strings, número de elementos em um array

- Os asserts são complementados com uma versão que imprime uma mensagem
 - `TEST_ASSERT_X_MESSAGE({modifiers}, {expected}, actual, {size/count}, message)`
- Asserts para arrays
 - `TEST_ASSERT_EQUAL_TYPEX_ARRAY(expected, actual, {size/count})`

Asserts para Boolean

- TEST_ASSERT (condition)
- TEST_ASSERT_TRUE (condition)
- TEST_ASSERT_FALSE (condition)
- TEST_ASSERT_NULL (pointer)
- TEST_ASSERT_NOT_NULL (pointer)
- TEST_ASSERT_EMPTY (pointer)
- TEST_ASSERT_NOT_EMPTY (pointer)

- TEST_ASSERT_EQUAL_INT (expected, actual)
- TEST_ASSERT_EQUAL_INT8 (expected, actual)
- TEST_ASSERT_EQUAL_INT16 (expected, actual)
- TEST_ASSERT_EQUAL_INT32 (expected, actual)
- TEST_ASSERT_EQUAL_INT64 (expected, actual)
- TEST_ASSERT_EQUAL_UINT (expected, actual)
- TEST_ASSERT_EQUAL_UINT8 (expected, actual)
- TEST_ASSERT_EQUAL_UINT16 (expected, actual)
- TEST_ASSERT_EQUAL_UINT32 (expected, actual)
- TEST_ASSERT_EQUAL_UINT64 (expected, actual)

Asserts para hexadecimal

- TEST_ASSERT_EQUAL_HEX (expected, actual)
- TEST_ASSERT_EQUAL_HEX8 (expected, actual)
- TEST_ASSERT_EQUAL_HEX16 (expected, actual)
- TEST_ASSERT_EQUAL_HEX32 (expected, actual)
- TEST_ASSERT_EQUAL_HEX64 (expected, actual)

Asserts para char, bit mask, comparação

- TEST_ASSERT_EQUAL_CHAR (expected, actual)
- TEST_ASSERT_BITS (mask, expected, actual)
- TEST_ASSERT_GREATER_THAN_INT8 (threshold, actual)
- TEST_ASSERT_GREATER_OR_EQUAL_INT16 (threshold, actual)
- TEST_ASSERT_LESS_THAN_INT32 (threshold, actual)
- TEST_ASSERT_LESS_OR_EQUAL_UINT (threshold, actual)
- TEST_ASSERT_NOT_EQUAL_UINT8 (threshold, actual)

- TEST_ASSERT_INT_WITHIN (delta, expected, actual)
- TEST_ASSERT_INT8_WITHIN (delta, expected, actual)
- TEST_ASSERT_INT16_WITHIN (delta, expected, actual)
- TEST_ASSERT_INT32_WITHIN (delta, expected, actual)
- TEST_ASSERT_INT64_WITHIN (delta, expected, actual)
- TEST_ASSERT_UINT_WITHIN (delta, expected, actual)
- TEST_ASSERT_UINT8_WITHIN (delta, expected, actual)
- TEST_ASSERT_UINT16_WITHIN (delta, expected, actual)
- TEST_ASSERT_UINT32_WITHIN (delta, expected, actual)
- TEST_ASSERT_UINT64_WITHIN (delta, expected, actual)
- TEST_ASSERT_HEX_WITHIN (delta, expected, actual)
- TEST_ASSERT_HEX8_WITHIN (delta, expected, actual)
- TEST_ASSERT_HEX16_WITHIN (delta, expected, actual)
- TEST_ASSERT_HEX32_WITHIN (delta, expected, actual)
- TEST_ASSERT_HEX64_WITHIN (delta, expected, actual)
- TEST_ASSERT_CHAR_WITHIN (delta, expected, actual)

Ponteiros e strings

- TEST_ASSERT_EQUAL_PTR (expected, actual)
 - Asserts that the pointers point to the same memory location.

- TEST_ASSERT_EQUAL_STRING (expected, actual)
 - Asserts that the null terminated ('\0') strings are identical. If strings are of different lengths or any portion of the strings before their terminators differ, the assertion fails. Two NULL strings (i.e. zero length) are considered equivalent.

- TEST_ASSERT_EQUAL_MEMORY (expected, actual, len)

Documentação completa dos asserts

- Disponível em
 - <https://github.com/ThrowTheSwitch/Unity/blob/master/docs/UnityAssertionsReference.md>
- Inclui arrays, ponto flutuante, variação em arrays, comparação dos valores dos elementos do array, double,

Exemplo simples

- Código disponível no material compartilhado
- Simples exemplo de uma função de troca de valores com ponteiros

- Código disponível no material compartilhado
- Testes unitários de uma estrutura de dados do tipo pilha genérica em C

Exercício: TDA conjuntos

- Código disponível no material compartilhado
- Tipo de Dado Abstrato (TDA) para manipulação de conjuntos
- Operações implementadas são
 - Criar um conjunto;
 - Destruir um conjunto;
 - Retornar o número de elementos dentro do conjunto;
 - Inicializar um conjunto como conjunto vazio;
 - Realizar a união entre dois conjuntos;
 - Realizar a intersecção entre dois conjuntos;
 - Realizar a diferença entre dois conjuntos;
 - Verificar se um elemento pertence ou não a um conjunto;
 - Inserir um elemento em um conjunto;
 - Remover um elemento em um conjunto;
 - Atribuir um conjunto a outro (exemplo: conjunto a = conjunto b);
 - Verificar se dois conjuntos são iguais;
 - Retornar o valor do menor elemento dentro de um conjunto;
 - Retornar o valor do maior elemento dentro de um conjunto;
 - Imprimir todos os elementos do conjunto;

Exercício: TDA conjuntos

- O exercício consiste em escrever os testes para duas operações do TDA
 - Implementar as funções `test_conjunto_uniao()` e `test_conjunto_interseccao()` no arquivo `test-main.c`
 - Compilar com “make” em caso de Linux
 - Solução em `test-main-solucao.c` (VEREMOS DEPOIS)
- `void conjunto_uniao(conjunto_t *a, conjunto_t *b, conjunto_t *c)`
 - Recebe os conjuntos "a" e "b" e retorna a uniao entre eles no conjunto "c"
 - Pre-condicao: os conjuntos "a", "b" e "c" devem ser conjuntos validos
 - Pos-condicao: o conjunto "c" contera a uniao entre os conjuntos "a" e "b"
- `void conjunto_interseccao(conjunto_t *a, conjunto_t *b, conjunto_t *c)`
 - Recebe os conjuntos "a" e "b" e retorna a interseccao entre eles no conjunto "c"
 - Pre-condicao: os conjuntos "a", "b" e "c" devem ser conjuntos validos
 - Pos-condicao: o conjunto "c" contera a interseccao entre os conjuntos "a" e "b"

- Introdução a metodologias de teste de software
- Introdução ao framework de testes cmocka
 - Exemplos
 - Exercícios
- Introdução ao framework de testes Unity
 - Integração com valgrind para detecção de vazamentos de memória
 - Integração com nm para análise de variáveis globais
 - Exemplos
 - Exercícios
- Introdução ao framework de testes google test para C++
 - Exemplos
 - Exercícios

- É o framework de testes desenvolvido pela google
- Versão atual é 1.11
 - <https://github.com/google/googletest>
- Projetos GoogleTest e GoogleMock foram fundidos
- Testes xUnit
 - Testes e estruturas derivados do Sunit
 - Test runner: executável
 - Test case: classe base na qual os testes unitários são derivados
 - Test fixtures: pré-condições para rodar o test
 - Test suite: conjunto de testes que compartilham fixtures
 - Test execution: setup() e tearDown() - similar ao Unit

- É multiplataform
 - Windows
 - Linux
 - MacOS
- Alguns princípios de projeto
 - Testes independentes e repetíveis
 - Testes devem ser portáteis e reusáveis
 - Quando um teste falha, deve prover o máximo de informações possível
 - Para o teste que falhou e continua no próximo
 - Testes devem ser rápidos
- Necessita C++ 11 e cmake ou Bazel para compilar

■ Conceitos básicos

- Asserts
 - Success, nonfatal failure ou fatal failure
- Test suite contém um ou mais testes
- Quando múltiplos testes em uma test suite compartilham objetos comuns e subrotinas, você pode colocá-los em uma classe test fixture
- Um programa de teste contém múltiplas test suites

- Assertions que geram falhas fatais
 - `ASSERT_*`
 - `ASSERT_EQ(x.size(), y.size()) << "Vectors x and y are of unequal length";`

- Assertions que geram falhas não fatais
 - Geram a falha mas continuam a execução

 - `EXPECT_*`

```
for (int i = 0; i < x.size(); ++i) {  
    EXPECT_EQ(x[i], y[i]) << "Vectors x and y differ at index " << i;  
}
```

- Fatal assertions
 - `ASSERT_TRUE(condition);`
 - `ASSERT_TRUE(condition);`
- Nonfatal assertions
 - `EXPECT_TRUE(condition);`
 - `EXPECT_FALSE(condition);`

Comparação entre 2 valores

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_EQ(val1, val2);</code>	<code>EXPECT_EQ(val1, val2);</code>	<code>val1 == val2</code>
<code>ASSERT_NE(val1, val2);</code>	<code>EXPECT_NE(val1, val2);</code>	<code>val1 != val2</code>
<code>ASSERT_LT(val1, val2);</code>	<code>EXPECT_LT(val1, val2);</code>	<code>val1 < val2</code>
<code>ASSERT_LE(val1, val2);</code>	<code>EXPECT_LE(val1, val2);</code>	<code>val1 <= val2</code>
<code>ASSERT_GT(val1, val2);</code>	<code>EXPECT_GT(val1, val2);</code>	<code>val1 > val2</code>
<code>ASSERT_GE(val1, val2);</code>	<code>EXPECT_GE(val1, val2);</code>	<code>val1 >= val2</code>

Comparação float

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_FLOAT_EQ(val1, val2);</code>	<code>EXPECT_FLOAT_EQ(val1, val2);</code>	the two <code>float</code> values are almost equal
<code>ASSERT_DOUBLE_EQ(val1, val2);</code>	<code>EXPECT_DOUBLE_EQ(val1, val2);</code>	the two <code>double</code> values are almost equal

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_NEAR(val1, val2, abs_error);</code>	<code>EXPECT_NEAR(val1, val2, abs_error);</code>	the difference between <code>val1</code> and <code>val2</code> doesn't exceed the given absolute error

Comparação de string

Fatal assertion	Nonfatal assertion	Verifies
<code>ASSERT_STREQ(str1, str2);</code>	<code>EXPECT_STREQ(str1, str2);</code>	the two C strings have the same content
<code>ASSERT_STRNE(str1, str2);</code>	<code>EXPECT_STRNE(str1, str2);</code>	the two C strings have different contents
<code>ASSERT_STRCASEEQ(str1, str2);</code>	<code>EXPECT_STRCASEEQ(str1, str2);</code>	the two C strings have the same content, ignoring case
<code>ASSERT_STRCASENE(str1, str2);</code>	<code>EXPECT_STRCASENE(str1, str2);</code>	the two C strings have different contents, ignoring case

Criando um teste

- Use a macro TEST()
 - Não retorna valores
- Dentro da função de teste, pode-se usar qualquer notação da linguagem C++ e os asserts do google test
- Os resultados dos testes são determinados pelos asserts
 - Se existem testes que falham ou não

```
TEST(TestSuiteName, TestName) {  
    ... corpo do teste ...  
}
```

Criando um teste

- Teste para uma função fatorial
 - int Factorial(int n);

```
// Teste de fatorial de 0
TEST(FactorialTest, HandlesZeroInput) {
    EXPECT_EQ(Factorial(0), 1);
}

// Teste de números positivos
TEST(FactorialTest, HandlesPositiveInput) {
    EXPECT_EQ(Factorial(1), 1);
    EXPECT_EQ(Factorial(2), 2);
    EXPECT_EQ(Factorial(3), 6);
    EXPECT_EQ(Factorial(8), 40320);
}
```

Com dois testes

Teste Fixtures

- Usar a mesma configuração de dados para múltiplos testes
 - Permite reusar a mesma configuração de objetos em diferentes testes
- Para criar uma “Fixture”
 - Derivar uma classe de `::testing::Test`
 - Dentro da classe, declare os objetos que pretende usar
 - Se necessário escreva um construtor para inicializar os objetos ou o método `SetUp()`
 - Se necessário escreva um destrutor ou `TearDown()` para realizar a liberação de qualquer recurso
 - Se necessário define métodos compartilhados pelos testes

Teste Fixtures

- Quando se usar Teste Fixtures, deve-se usar a macro **TEST_F()** e não **TEST()**

```
TEST_F(TestFixtureName, TestName) {  
    ... corpo do teste ...  
}
```

- O parâmetro “TestFixtureName” é o nome da classe Fixture
- Para cada TEST_F() o framework irá um novo test Fixture em tempo de execução, faz sua inicialização, o executa, e termina chamando TearDown()

Teste Fixtures - Exemplo

```
// E is the element type.
template <typename E>
class Queue {
public:
    Queue();
    void Enqueue(const E& element);
    // Returns NULL if the queue is empty.
    E* Dequeue();
    size_t size() const;
    ...
};
```

```
class QueueTest : public ::testing::Test {
protected:
    void SetUp() override {
        q1_.Enqueue(1);
        q2_.Enqueue(2);
        q2_.Enqueue(3);
    }

    // void TearDown() override {}

    Queue<int> q0_;
    Queue<int> q1_;
    Queue<int> q2_;
};
```


Teste Fixtures - Exemplo

```
TEST_F(QueueTest, IsEmptyInitially) {
    EXPECT_EQ(q0_.size(), 0);
}

TEST_F(QueueTest, DequeueWorks) {
    int* n = q0_.Dequeue();
    EXPECT_EQ(n, nullptr);

    n = q1_.Dequeue();
    ASSERT_NE(n, nullptr);
    EXPECT_EQ(*n, 1);
    EXPECT_EQ(q1_.size(), 0);
    delete n;

    n = q2_.Dequeue();
    ASSERT_NE(n, nullptr);
    EXPECT_EQ(*n, 2);
    EXPECT_EQ(q2_.size(), 1);
    delete n;
}
```

Teste Fixtures - Exemplo

- Quando o teste fixture anterior executa, acontece o seguinte:
 - 1) Googletest cria um objeto QueueTest (digamos t1)
 - 2) t1.Setup() inicializa t1
 - 3) O primeiro teste (IsEmptyInitially) executa em t1
 - 4) t1.TearDown() é chamado no final
 - 5) t1 é destruído
 - 6) Os passos anteriores são repetidos para o objeto QueueTest do outro teste, executando DequeueWorks

Execução do testes

- Tanto TEST() como TEST_F() registram os testes no framework quando chamadas
- Depois de definir os testes, é possível invocá-los usando RUN_ALL_TESTS()
 - Retorna 0 se todos os testes foram bem sucedidos
 - 1 caso contrário

- Vejamos o arquivo `exemplo_main.cc` compartilhado
 - Dentro de `teste_de_software/google_test/exemplo_main.cc`
- A função `::testing::InitGoogleTest(&argc, argv);`
 - Faz o parse da linha de comando procurando pelas flags do framework

Parâmetros de execução

- Google teste suporta uma ampla variedade de flags (parâmetros) passados na linha de comando
- Listar os nomes dos testes
 - `./exec -gtest_list_tests`

```
TestSuite1.
  TestName1
  TestName2
TestSuite2.
  TestName
```

Parâmetros de execução

■ Executar apenas um subconjunto dos testes

- `--gtest_filter="padrão"`
- Padrão pode ser '*' (qualquer string), '?' (qualquer caracter), ':' separador de padrões, '-' resultado negativo

■ Exemplo:

- `./foo_test --gtest_filter=*` (executa todos os testes)
- `./foo_test --gtest_filter=FooTest.*` (executa todos os testes dentro da Suite FooTest)
- `./foo_test --gtest_filter=-*DeathTest.*` (roda todos os testes que não são da Suite DeathTest)
- `./foo_test --gtest_filter=FooTest.*-FooTest.Bar` (roda todos os testes da FooTest com exceção do FooTest.Bar)
- `./foo_test --gtest_filter=*Null*:~*Constructor*` (roda qualquer teste que tenha contenha "Null" ou "Constructor" no seu nome)

Parâmetros de execução

- Repetir os testes
 - `--gtest_repeat=`
 - `--gtest_repeat=1000`
 - `--gtest_repeat=-1` (pra sempre)
 - `./foo_test --gtest_repeat=1000 --gtest_break_on_failure`
(para na primeira falha)

- Escrever a saída em um arquivo
 - `--gtest_output="xml:path_to_output_file"` (xml)
 - `--gtest_output="json:path_to_output_file"` (json)

- Mais sobre parâmetros e opções de execução
 - <https://github.com/google/googletest/blob/master/googletest/docs/advanced.md>

Instalação do Google Test

- No ubuntu, iniciar com o pacote de desenvolvimento
 - `sudo apt-get install libgtest-dev`
- Necessário compilar os fontes para gerar os arquivos de biblioteca necessários
 - `sudo apt-get install cmake # instalar cmake`
 - `cd /usr/src/gtest`
 - `sudo cmake CMakeLists.txt`
 - `sudo make`
 - Copiar ou fazer link simbólico das `libgtest.a` e `libgtest_main.a` para `/usr/lib`
 - `sudo cp *.a /usr/lib`

Instalação do Google Test

■ No Windows

- <https://kezunlin.me/post/aca50ff8/>
- Substituir a versão para
<https://github.com/google/googletest/archive/release-1.10.0.zip>

■ No eclipse

- <https://www.ics.uci.edu/~pattis/common/modules46/googletestpc.html>

- Exemplo simples
 - Teste de raiz quadrada
- Para compilar o código
 - Entrar no diretório `cd google_test/exemplo_raiz_quadrada`
 - Compilar
`cmake CMakeLists.txt`
`make`
 - Rodar
`./runTests`

Exemplo: usando classe Fixture

- Entrar em exemplo_classes
 - No diretório tem os arquivos sem nenhum teste
 - Dentro do diretório test tem os arquivos com teste
- Para compilar
 - Comando “make”
- Para rodar
 - ./main-test
- Para rodar com argumentos do google test
 - ./main-test --gtest_output="xml:test.xml" –
gtest_repeat=10
 - Geração de xml com resumo dos testes
 - Repetição
 - Permite a integração com jenkins (automação de testes)

- Philip Koopman. Better Embedded System Software. 2010.
- Embedded System Software Quality. ISSRE 2016 Workshop Keynote. Philip Koopman.
- Embedded Software Testing. Distributed Embedded Systems. Carnegie Mellon University. Philip Koopman.
- <https://betterembsw.blogspot.com/>
- <http://www.yolinux.com/TUTORIALS/Cpp-GoogleTest.html>

Obrigado!

