# Embedded Unit-Tests und Mocking mit CMock

Simon Raffeiner, Leitwerk AG
Appenweier, Germany
sraffeiner@leitwerk.de, sraffeiner@stud.fh-offenburg.de

***Abstract*—Softwareentwicklung für eingebettete Systeme hat sich in den vergangenen Jahren meist kaum verändert. Obwohl die Konzepte hinter Agiler Entwicklung, Test-Driven Development und Extreme Programming auch auf eingebettete Systeme übertragen wurden (siehe [Gre02], [Dow04] und [KBE06]) liegen die Potentiale in vielen Projekten brach. Unit Testing und Mocking werden weitestgehend ignoriert. Das Problem entsteht aus der Fehleinschätzung vieler Entwickler, dass Software für eingebettete Systeme ohne Betriebssystem schwer zu testen ist weil die notwendigen Angriffspunkte (Automatisierbarkeit, Frameworks) fehlen und die knapp bemessenen Ressourcen den Einsatz größerer Test-Frameworks unmöglich macht. Statt dessen beschränkt man sich meist auf Systemtests.**
***Kernthema dieses Dokumentes ist die Einführung in das CMock[1] Mocking Framework, das in Verbindung mit dem Unity[2] Unit Test Framework genutzt werden kann um White-Box-Tests für C-Programme generieren.***

## I. Einleitung

Grudlegende Kenntnisse zu den Themen "White-Box Testing" und "Unit Testing" im Bereich der Softwareentwicklung werden vorausgesetzt.
Kapitel 2 gibt einen Überblick über die grundlegenden Probleme beim Testen von eingebetteten Systemen. Kapitel 3 beschreibt kurz die Historie hinter der Entstehung des CMock Frameworks, Kapitel 4 die generelle Funktionalität und Kapitel 5 das interne Design. Kapitel 6 geht beispielhaft auf die Verbindung mit Unity ein, während Kapitel 7 den Ressourcenverbrauch auf verschiedenen Plattformen analysiert (embedded und PC). Das Dokument schließt ab mit Kapitel 8, einer Zusammenfassung der Limitierungen und einem kurzen Ausblick in die Zukunft.

## II. Testen von eingebetteten Systemen

Eingebettete Systeme unterscheiden sich in Hinblick auf verfügbare Ressourcen, Rechenleistung und Interaktionsmöglichkeiten für Debugging-Zwecke fundamental von herkömmlichen Personal Computern. Soll Unit Testing im großen Stil eingesetzt werden - vorzugsweise für jede nicht-triviale Funktion in jedem nicht-trivialen Modul - muss die Durchführung der Tests Teilgut in die verwendete Toolchain integriert sein, so schnell wie möglich ablaufen und Ergebnisse produzieren die so exakt wie möglich denen des Produktivsystems entsprechen. Ein Entwickler muss im Idealfall jederzeit einen Testlauf starten können damit eventuell auftretende Fehler sofort bemerkt werden, in größeren Projekten kann es sogar sinnvoll sein ein dediziertes Test-System aufzubauen das jede Änderung am Quellcode sofort automatisch übersetzt, testet und notfalls Alarm schlägt. Die am häufigsten verwendeten Lösungsmöglichkeiten sind nachfolgend aufgelistet:

- Ein spezielles Entwicklungssystem mit einer Debugging-Schnittstelle, das ein automatisiertes Hochladen und Ausführen eines Test-Binaries erlaubt. Die Ergebnisse werden über eine geeignete Schnittstelle (RS-232, JTAG etc.) an den Host-PC zurückübermittelt oder z.B. an eine definierte Speicheradresse geschrieben, die wiederum über die Debugging-Schnittstelle ausgelesen werden kann. Diese Methode nimmt am meisten Zeit in Anspruch, wird aber dem Anspruch an realitätsnahe Ergebnisse gerecht und erlaubt auch das Testen von Peripherie (Timer, Zähler etc.).
- Ein Emulator auf dem Entwicklungs-PC. Diese Methode ist üblicherweise schneller als die Verwendung eines Entwicklungssystems, aber der Emulator muss die gleichen Automatisierungsmöglichkeiten (Hochladen des Binaries, Ausführung und Auslesen der Ergebnisse) bieten - was nicht immer der Fall ist. Viele Emulatoren sind außerdem überhaupt nicht in der Lage Peripheriegeräte zu emulieren, oder die Emulation ist zumindest nicht akkurat.
- Übersetzen der Tests für die Architektur des Entwickler-PCs (Cross-compiling) und anschließende lokale Ausführung. Dies ist möglich wenn in Hochsprachen wie C programmiert wird, erfordert allerdings auch ein spezielles Augenmerk auf "Feinheiten" wie etwa die Länge von Datentypen. Diese Methode ist die schnellste und am Einfachsten durchzuführen, erlaubt aber keinerlei Zugriff auf spezifische Peripherie. Die Ergebnisse müssen in Hin-

[1] http://cmock.sourceforge.net
[2] http://embunity.sourceforge.net

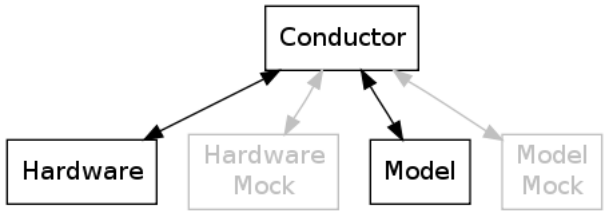


Fig. 1. Komponenten im Model-Conductor-Hardware pattern

sicht auf Laufzeit, Ressourcenverbrauch etc. nicht mit denen auf dem eingebetteten System übereinstimmen, es ist auf diese Weise aber sehr gut möglich Fehler in Hardwareunabhängiger Logik zu finden.

Der größte Unterschied zwischen Personal Computern mit grafischen Benutzeroberflächen und eingebetteten Systemen ist das Fehlen der Präsentations-Schicht (View): Die meisten eingebetteten Systeme erfordern überhaupt keine Interaktion mit dem Benutzer, und selbst wenn Eingabe- und Anzeigemöglichkeiten bestehen (z.B. LEDs, LCD, Taster) werden diese üblicherweise auf der gleichen Ebene behandelt wie etwa Sensoren und Relais.

[KBE06] führt das "Model-Conductor-Hardware" (MCH) Pattern für eingebettete Systeme ein. Es ist im Kern eine Adaption des bekannten "Model-View-Controller" [BM00] Patterns, ersetzt aber die in Desktop-Anwendungen existierende Präsentations-Schicht durch eine generische Hardware-Schicht. MCH diktiert eine Modularisierung allen Quellcodes in lose gekoppelte Module, wobei jedes Modul nur Unter-Module besitzen darf die sich in eine der drei Kategorien einsortieren lassen: Ansprechen der Hardware (Treiber), speichern/verarbeiten von Daten (Model) oder Entscheidungslogik (Conductor). Module und Unter-Module kommunizieren über eindeutig definierte Schnittstellen - hier handelt es sich um Methodenaufrufe die in Header-Dateien definiert sind. Sobald eine Schnittstelle festgelegt ist darf sämtliche Kommunikation nur noch über die darin definierten Methoden ablaufen. Dies erhöht sowohl die Testbarkeit als auch die Wiederverwendbarkeit drastisch, Module können ausgetauscht werden solange Sie die Spezifikation vollständig erfüllen.
Da Module in der Realität zusammenarbeiten müssen um eine bestimmte Funktionalität zu erfüllen ist es nicht ohne weiteres möglich ein einzelnes Modul eigenständig zu testen. Um dies dennoch zu ermöglichen werden Mock-Module verwendet - minimale Implemetierungen der Schnittstelle ohne funktionierendes "Innenleben". Mocks werden automatisch generiert und darauf hin "trainiert" die Spezifikation der Schnittstelle (Aufrufparameter, Rückgabewerte) zu simulieren. Da das Modul unter Test nun mit nur noch mit Mocks zusammenarbeitet die ein exakt bekanntes Verhalten zeigen ist das Gebot der losen Kopplung wieder erfüllt.
Die positiven Auswirkungen der Modularisierung auf die generelle Code-Qualität und potentielle Wiederverwendbarkeit in anderen Projekten sind bereits ausführlich bewiesen worden, [BM00] zeigt darüber hinaus einen nur sehr kleinen Overhead auf (wenn optimierende Compiler verwendet werden).

## III. Geschichte

CMock ist eine Entwicklung der Atomic Objects, LLC[3] und entstammt einem Projekt für Savant Automation[4] [FBKW07], einem der großen usamerikanischen Hersteller von führerlosen Transportfahrzeugen.
Savant beauftragte Atomic Objects Ende 2005 mit der Neuprogrammierung der Firmware für zwei ARM9[5]-basierte Steuerplatinen. Atomic Object setzte auf einen agilen[6] Entwicklungsprozess, benötigte dafür aber eine Tool-Chain die alle Möglichkeiten bot die das Team bereits bei der Entwicklung von Desktop- und Web-Anwendungen angewendet hatte und gleichzeitig auf einem Microchip PIC-Controller mit 256 Byte RAM und 32 Kilobyte ROM funktionierte.
Das Projekt wurde erfolgreich durchgeführt und brachte eine ganze Reihe von Entwicklungen hervor, darunter das bereits erwähnte Model-Conductor-Hardware Pattern [KBE06] und mehrere Frameworks (Argent, Unity und CMock). Während die Entwicklung der Firmware für die erste Steuerplatine noch neun Monate in Anspruch genommen hatte, konnte die Software für die zweite Platine (bei ähnlicher Komplexität) durch Verbesserungen am Prozess in nur noch vier Monaten fertiggestellt werden.
Im Rahmen einer Präsentation (siehe [WV08]) für die Embedded Systems Conference Boston im Oktober 2008 wurden sowohl Unity als auch CMock untereiner Open-Source Lizenz freigegeben. Greg Williams, Michael Karlesky und Mark VanderVoord, Mitarbeiter bei Atomic Objects, führten die Entwicklung bis zur aktuellen Version 1.2.2 weiter fort.

[3] http://www.atomicobject.com
[4] http://www.savantautomation.com
[5] http://www.arm.com, die dominierende Architektur in eingebetteten Systemen
[6] http://www.agilealliance.org

## IV. Funktionalität

Das CMock Framework besteht aus einer Serie von Skripten für die Interpretersprache Ruby[7] und generieriert aus in der Programmiersprache C verfassten Header-Dateien Quellcode für Mock-Ojekte. Alle Ausgaben sind wiederum C-Code.
CMock ist spezialisiert auf Mocking und bietet daher keinerlei Funktionalität für Unit Tests, was üblicherweise die Nutzung eines zusätzlichen Frameworks notwendig macht. Generatoren für Unity, Ignore und CException[8] werden bereits mitgeliefert.

### A. Distribution

Dieses Dokument basiert auf der Ende 2008 freigegebenen Version 1.2.2. Das Quellcode-Archiv enthält folgende Verzeichnisse:

- **config** Konfigurationsdateien
- **docs** Dokumentation
- **examples** Beispiele
- **iar** Dateien für IAR Systems "Embedded Workbench Compiler for ARM architectures"'
- **lib** Das eigentliche Framework
- **test** Unit Tests für CMock selbst
- **vendor** Notwendige Bibliotheken für die Unit Tests

Wenn CMock in ein Projekt eingebunden wird sind nur die Ruby-Skripte aus dem Unterverzeichnis "lib" und das Unity Framework aus dem Verzeichnis "vendor" notwendig.

## V. Internes Design

CMock folgt dem Design der meisten Mocking Frameworks: Biete eine Möglichkeit anzugeben wie oft, mit welchen Aufrufparametern und welchen Rückgabewerten eine Methode gerufen wird, speichere alle Daten intern und imitiere damit die Methode. Sind alle Aufrufparameter wie erwartet liefere den bekannten Rückgabewert, falls nicht melde einen Fehler.
Da C im Gegensatz zu anderen Umgebungen wie z.B. Java oder .NET keine Code-Manipulation zur Laufzeit unterstützt und CMock auf eingebettete Systeme ausgelegt ist muss sämtlicher Quellcode vor dem Übersetzungsvorgang erzeugt werden. Dies geschieht in drei Schritten (siehe auch Abbildung 2):

- Parser: Liest alle Header-Dateien ein und extrahiert die notwendigen Informationen
- Generator: erzeugt mittels Plug-Ins den notwendigen Mock-Code

[7]http://www.ruby.org, eine verbreitete objektorientierte Skriptsprache

[8]http://apps.sourceforge.net/mediawiki/cexception/, ein einfacher Ausnahme-Mechanismus für ANSI C

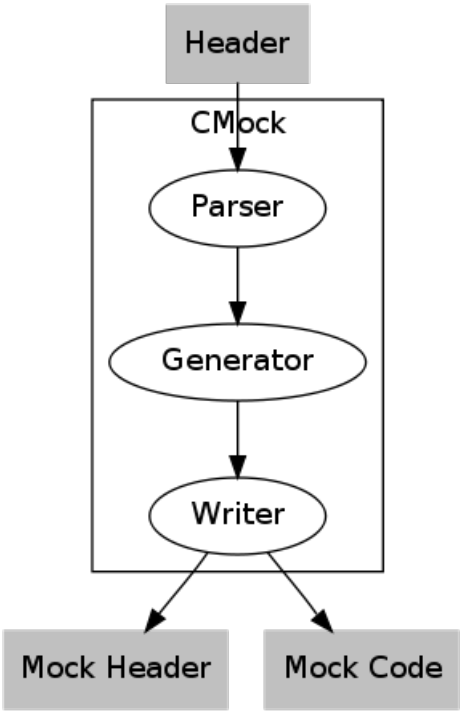


Fig. 2. Datenfluss innerhalb von CMock

- File Writer: Eine Zwischenschicht zum Dateisystem, schreibt das endgültige Ergebnis in Dateien.

Abbildung 3 stellt die internen Beziehungen zwischen den Modulen schematisch dar. Zwei zusätzliche, nicht näher beschriebene Module liefern Konfigurationseinstellungen (config) und vorgefertigte C-Codeteile für den Generator (generator_utils).

### A. Der Parser

Der Parser steckt in einer einzigen Datei, cmock_header_parser.rb. Er extrahiert Informationen aus einer Header-Datei und legt folgende Daten in einer internen Has-Struktur ab:

- Zusätzlich eingebundene Header-Dateien (können notwendige Datentypen beinhalten)
- Externe Variablen
- Methodennamen, Modifikatoren (z.B. static), Datentypen von Rückgabewerten und Aufrufparameter, Namen der Aufrufparameter

Abbildung 4 stellt eine Beispiel-Header-Datei für einen Temperaturfilter dar (liegt der CMock-Distribution bei), Abbildung Figure 5 die daraus extrahierten Informationen .

Der Parser liest derzeit die Eingabedatei Zeile für Zeile ein überspringt nicht benötigte Informationen (z.B Kommentare oder leere Zeilen). Von den verbleibenden Zeilen werden zunächst solche

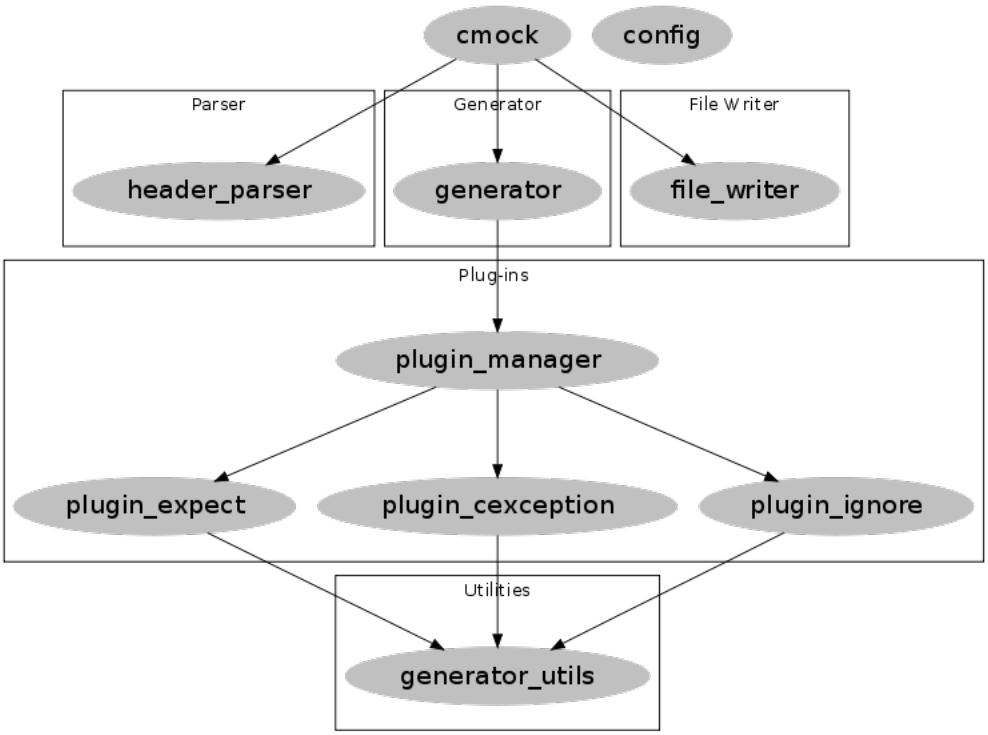


Fig. 3. Interne Abhängigkeiten zwischen Modulen

```
#ifndef _TEMPERATUREFILTER_H
#define _TEMPERATUREFILTER_H
#include "Types.h"

void TemperatureFilter_Init(void);
float TemperatureFilter_GetTemperatureInCelcius(void);
float TemperatureFilter_ProcessInput(float temperature);
#endif // _TEMPERATUREFILTER_H
```

Fig. 4. TemperatureCalculator.h: Beispiel-Header-Datei für ein Temperatur-Filter-Modul

verarbeitet die auf zusätzlich einzubindende Header-Dateien hinweisen, im letzten Schritt dann solche mit Methodendeklarationen.
Die Extraktion wird durch reguläre Ausdrücke realisiert und funktioniert in der Praxis stabil. Da der Parser jedoch keinerlei Plug-Ins vorsieht wäre eine Adaption auf eine andere Programmiersprache als C nur durch einen kompletten Ersatz des Parsers möglich.

### *B. Der Generator*

Die Erzeugung des Quellcodes wird durch einen Kern (cmock_generator.rb) und mehrere Plug-Ins (cmock_generator_*.rb) realisiert, die der Plug-In Manager (cmock_plugin_manager.rb) verwaltet.
Der Kern übernimmt die folgenden Aufgaben:

- Kommunikation mit dem Parser (Übernahme der erzeugten Hash-Struktur)
- Kommunikation mit dem File Writer (Ausgabe des endgültigen Quellcodes)
- Generation des Mock-headers mit allen benötigten Includes und externen Variablen
- Generation der globalen *_Init(), *_Destroy() und *_Verify() Methoden
- Generation der Mock-Instanz

Die Erzeugung der Mock-Methoden wird an Plug-Ins delegiert um eine bessere Integration mit verschiedenen Unit Test Frameworks zu ermöglichen. Listing 6 zeigt den generierten Mock Header für den Temperatur-Filter, Listing 7 die zugehörige Mock-Instanz. Die eigentlichen Methoden der nun gemockten Schnittstelle sind nicht Teil der erzeugten Header-Datei, sondern die Schnittstelle - in diesem Fall “TemperatureFilter.h” - wird zu Beginn eingebunden. Der eigentliche Quellcode des Mocks ist an dieser Stelle aus Platzgründen nicht dargestellt, ist allerdings auch

```
{:includes=>["Types.h"], :externs=>[], :functions=>[{:modifier=>"", :
   rettype=>"void", :args=>[], :var_arg=>nil, :name=>"
   TemperatureFilter_Init", :args_string=>"void"}, {:modifier=>"", :
   rettype=>"float", :args=>[], :var_arg=>nil, :name=>"
   TemperatureFilter_GetTemperatureInCelcius", :args_string=>"void"}, {:
   modifier=>"", :rettype=>"float", :args=>[{:type=>"float", :name=>"
   temperature"}], :var_arg=>nil, :name=>"TemperatureFilter_ProcessInput"
   , :args_string=>"float␣temperature"}]}
```

Fig. 5. Aus dem Header extrahierte Informationen in der internen Darstellung

```
#ifndef _MOCKTEMPERATUREFILTER_H
#define _MOCKTEMPERATUREFILTER_H
#include "TemperatureFilter.h"

void MockTemperatureFilter_Init(void);
void MockTemperatureFilter_Destroy(void);
void MockTemperatureFilter_Verify(void);

void TemperatureFilter_Init_Expect(void);
void TemperatureFilter_GetTemperatureInCelcius_ExpectAndReturn(float
   toReturn);
void TemperatureFilter_ProcessInput_ExpectAndReturn(float temperature,
   float toReturn);
#endif
```

Fig. 6. MockTemperatureCalculator.h: Generierter Mock-Header

```
static struct MockTemperatureFilterInstance
{
  unsigned char allocFailure;
  unsigned short TemperatureFilter_Init_CallCount;
  unsigned short TemperatureFilter_Init_CallsExpected;
  unsigned short TemperatureFilter_GetTemperatureInCelcius_CallCount;
  unsigned short TemperatureFilter_GetTemperatureInCelcius_CallsExpected;
  float *TemperatureFilter_GetTemperatureInCelcius_Return;
  float *TemperatureFilter_GetTemperatureInCelcius_Return_Head;
  float *TemperatureFilter_GetTemperatureInCelcius_Return_HeadTail;
  unsigned short TemperatureFilter_ProcessInput_CallCount;
  unsigned short TemperatureFilter_ProcessInput_CallsExpected;
  float *TemperatureFilter_ProcessInput_Return;
  float *TemperatureFilter_ProcessInput_Return_Head;
  float *TemperatureFilter_ProcessInput_Return_HeadTail;
  float *TemperatureFilter_ProcessInput_Expected_temperature;
  float *TemperatureFilter_ProcessInput_Expected_temperature_Head;
  float *TemperatureFilter_ProcessInput_Expected_temperature_HeadTail;
} Mock;
```

Fig. 7. Excerpt from MockTemperatureCalculator.c: Generierte Mock-Instanz

nicht weiter interessant da er automatisch aus vorgefertigen Code-Teilen erzeugt wird.

Die Mock-Instanz ist das Herz eines jeden Mock-Moduls. Sie speichert folgende Informationen:

- allocFailure, Anzahl der aufgetretenen Fehler bei der Speicherreservierung
- *_Return_CallCount (tatsächliche Anzahl Aufrufe) und *_Return_CallsExpected (Erwartete Anzahl Aufrufe) für jede Methode
- *_Return (nächstes Element), Return_Head (Start) und Return_HeadTail (Ende), Pointer in ein Array mit allen trainierten Rückgabewerten
- *_$variable (nächstes Element), *_$variable_Head (Start) und *_$variable_HeadTail (Ende), Pointer in ein Array mit den erwarteten Aufrufwerten jedes Parameters jeder Methode

Die zusätzlich erzeugten Mock-Methoden dienen der Verwaltung dieser Datenstruktur. *_Init() setzt

alle Zähler und Arrays in den Ausgangszustand zurück, *_Destroy() gibt sämtlichen reservierten Speicher wieder frei, *_Verify() vergleicht mit Methoden aus dem benutzten Unit Test Framework alle Aufrufzähler mit den tatsächlichen Werten und löst bei Ungleichheit einen Fehler aus.
_Expect(parameter) und *_ExpectAndReturn (parameter, rückgabewert) arbeiten wie von anderen Mock Frameworks gewohnt: Sie erhöhen die Anzahl der erwarteten Aufrufe für die zugehörige Methode um Eins, legen die erwarteten Parameter und, im Falle von *_ExpectAndReturn(), den zu liefernden Rückgabewert in der Mock-Instanz ab.
Die eigentlichen, in der Schnittstelle spezifizierten Methoden, die später vom getesteten Modul aufgerufen werden, sind stets gleich aufgebaut: Vergleichen der erhaltenen Parameter mit den in der Mock-Instanz abgelegten Soll-Werten, Verwalten der Pointer in den Arrays (z.B. weiterspringen auf den nächsten Datensatz), Rückgabe des bekannten Wertes bei Richtigkeit und Auslösen eines Fehlers bei Abweichungen.
Es sei an dieser Stelle angemerkt dass die Zuständigkeit für das korrekte Linken von Test-Drivern und Mocks beim Build-System liegt. Da Mocks und "echte" Module identische Methodensignaturen besitzen kann es sonst zu Fehlermeldungen bezüglich mehrfach existierender Symbole kommen.

### C. Der File Writer

Der "File Writer" schreibt die erzeugten Quellcodes in temporäre Dateien und verschiebt diese nach erfolgreichem Ablauf in die konfigurierten Verzeichnisse. Da sich auf dieser Ebene nur einfache Methoden befinden sei an dieser Stelle auf die Datei "cmock_file_writer.rb" verwiesen.

### D. Integration mit Unity

CMock arbeitet sehr stark mit dem Unity Framework zusammen. Obwohl die Code-Erzeugung eigentlich unabhängig vom Framework sein sollte werden in manchen Teilen automatisch Unity-Aufrufe eingefügt.

Listing 8 zeigt ein Beispiel für die Nutzung des generierten Mocks in einem Test-Driver: Das Modul "UsartModel" ist für die Umwandlung der aktuellen Temperatur in einen String zur Ausgabe auf ein LC-Display zuständig. Der benötigte Temperaturwert wird durch Methodenaufrufe vom TemperatureFilter abgefragt. Der Test-Driver initialisiert den Mock, "trainiert" Ihn auf die benötigten Methodenaufrufe und Rückgabewerte und kontrolliert dann das vom UsartModel gelieferte Endergebnis. In diesem Fall soll die Temperatur beim ersten Aufruf bei 25 Grad Celsius liegen, der zweite Aufruf testet dann eine Fehlerbedingung (es sei angenommen dass eine unendlich negative Temperatur einen Fehler signalisiert).
Durch die Nutzung des Mocks ist der Test unabhängig von echten Sensoren und auch unabhängig von einem tatsächlich existierenden UsartModel-Modul. Es ist nun sogar möglich dieses Modul bereits zu entwickeln und zu testen wenn mit der Arbeit an den anderen Modulen noch gar nicht begonnen wurde - solange die Schnittstelle stabil bleit und eingehalten wird unterscheidet sich ein trainierter Mock nicht von einem "echten" Modul.

```
ruby lib/cmock.rb -oconfig.yml src/
   uart.h src/sensor.h
```

Fig. 9. Beispiel: Aufruf über den Interpreter

## VI. Aufruf

CMock wird üblicherweise in eine Toolchain eingebunden um automatisch Mock-Objekte zu generieren. Hierfür werden zwei Modi angeboten: Das Skript "ruby.rb" kann direkt über den Ruby-Interpreter aufgerufen werden, falls bereits Ruby-Skripte existieren bietet CMock aber auch eine Klasse mit einigen Methoden an. Beide Modi sind an Funktionalität identisch, ein Blick in das "ruby.rb" Skript zeigt, dass dort im Kern einfach eine Instanz des Objektes erzeugt und dann die notwendigen Methoden ausgeführt werden.

### A. Aufruf über den Interpreter

Beim direkten Aufruf über den Interpreter werden alle Parameter als Pfade zu C-Header-Dateien oder einer optionalen Konfigurationsdatei interpretiert. CMock iteriert dann über alle Pfade und generiert für jede Header-Datei einen Mock. Listing 9 zeigt einen beispielhaften Aufruf.

Konfigurationsdateien müssen im YAML[9]-Format vorliegen. Eine vollständige Liste aller Optionen findet sich in der Projektdokumentation.
Wenn keine Konfigurationsdatei angegeben wird gelten die Standard-Einstellungen, die Mocks werden dann separat im Unterverzeichnis mocks/ abgelegt.

[9]http://www.yaml.org/, YAML Ain't Markup Language, ein menschenlesbarer Standard zur Serialisierung von Daten

```
void testGetFormattedTemperature(void)
{
  TemperatureFilter_Init();

  TemperatureFilter_GetTemperatureInCelcius_ExpectAndReturn(25.0f);
  TEST_ASSERT_EQUAL_STRING("25.0 C\n", UsartModel_GetFormattedTemperature
     ());

  TemperatureFilter_GetTemperatureInCelcius_ExpectAndReturn(-INFINITY);
  TEST_ASSERT_EQUAL_STRING("Temperature sensor failure!\n",
     UsartModel_GetFormattedTemperature());

  TemperatureFilter_Verify();
  TemperatureFilter_Destroy();
}
```

Fig. 8. Nutzung des generierten Mocks in einem Test-Driver für ein anderes Modul

```
cmock = CMock.new(options)
cmock.setup_mocks(list)
cmock.generate_mock(source)
```

Fig. 10. Ruby functions offered by CMock

### B. Aufruf aus anderen Ruby-Skripten oder über Rake

CMock kann als natives Ruby-Objekt aus anderen Ruby-Skripten heraus genutzt werden, Listing 10 zeigt die dafür angebotenen Methoden.

Der Konstruktor akzeptiert die selben Optionen die auch in der YAML-Konfigurationsdatei angegeben werden können. Werden keine Optionen angegeben gelten die Default-Einstellungen. `setup_mocks()` generiert Mock-Objekte für eine Liste von Eingabedateien, `generate_mock()` für eine einzelne Datei.

## VII. Integration in existierende Projekte

Die CMock-Distribution und die Beispieldateien nutzen von Haus aus das Rake Build-System, die Integration in eine bestehende Tool-Chain gestaltet sich allerdings einfach: Es muss lediglich ein Build-Schritt eingefügt werden der vor der Compilierung die Mocks erzeugt. Das Projekt-Wiki enthält eine Anleitung[10] wie dies in der verbreiteten Eclipse-Entwicklungsumgebung[11] konfiguriert werden kann.

Die meisten Projekte haben eine bereits existierende Ordnerstruktur, es liegt daher nahe die ersten wenigen Mocks und Unit Tests zusammen mit den zugehörigen Modulen im selben Verzeichnis abzulegen. Dies erweist sich in der Praxis jedoch meist als unnötig problematisch: Entwicklungsumgebungen mit erweiterter Funktionalität wie z.B. Indexierung, Quellcode-Navigation und Automatischer Vervollständigung geraten angesichts der nun plötzlich mehrfach vorhandenen Methoden (je ein Mal im eigentlichen Modul und zusätzlich in dessen Mock) "aus dem Tritt", daneben erscheinen zusätzlich die internen Mock-Methoden (Init(), Destroy(), *_Expect() etc.). Es ist meist möglich einzelne Dateien von diesen Funktionen auszuschließen, allerdings steigt damit auch der Pflegeaufwand. Werden Modul-Dateien, Mocks und Unit Tests dagegen jeweils in einem eigenen Ordner geführt müssen diese nur ein Mal ausgeblendet werden.

[10] http://sourceforge.net/apps/trac/cmock/wiki/EclipseIde, Eclipse IDE Integration

[11] IDE, Integrated Development Environment

```
cmock:
  mock_path: 'mocks/'
  includes:
    - 'Types.h'
  plugins:
    - 'ignore'
```

Fig. 11. Beispiel für einen Konfigurationsabschnitt in YAML

Alle CMock-Parameter, inklusive der Ausgabepfad für die generierten Dateien, sind über eine YAML-Datei konfigurierbar. Listing 11 zeigt ein Beispiel für einen Konfigurationsabschnitt der CMock anweist die Header-Datei "Types.h" zwangsweise einzubinden, das "ignore"-Plugin zu laden und die Ergebnisse in das Unterverzeichnis "mocks" relativ zum aktuellen Pfad zu schreiben.

Ein größeres Projekt kann den Einsatz mehrerer Konfigurationsdateien erfordern.

## VIII. Ressourenverbrauch

Die meisten eingebetteten Systeme besitzen nur sehr knapp bemessene Speicher für Code (In-

| | Code | Instance |
|---|---|---|
| Atmel AVR (Atmega8) | 1664 | 31 |
| Atmel AVR (Atmega32) | 1714 | 31 |
| Intel 8051 | 1681 | 31 |
| Intel i386 | 1775 | 60 |
| AMD x86_64 | 1804 | 104 |

Fig. 12. Ressourcenverbrauch auf verschiedenen Plattformen

terner/externer Flash-Speicher oder EEPROM) und Daten (interner/externer RAM). Wenn Unit Tests direkt auf dem Gerät oder in einem Emulator ausgeführt werden sollen muss die Speicherbelegung so klein wie möglich ausfallen, damit so viele Tests wie möglich aus einem einzigen Binary heraus durchgeführt werden können. Dies verringert die Anzahl der notwendigen Test-Binaries und beschleunigt den Vorgang.
Die erste Spalte in Tabelle 12 führt die Größe des aus Listing 4 erzeugten Maschinencodes für verschiedene Architekturen (PC und Embedded) auf, die zweite Spalte die Größe der leeren Mock-Instanz. Diese Werte ändern sich zur Laufzeit nicht.
Zu diesen beiden Werten kommt zur Laufzeit noch der zur Speicherung aller Aufrufparameter und Rückgabewerte notwendige Speicherplatz hinzu: Jeder Aufruf von *_Expect(parameters) oder *_ExpectandReturn (parameters, return) erweitert die internen Arrays. Das Endergebnis ist daher abhängig von den abzulegenden Datentypen und der Anzahl trainierter Werte. Außerdem muss die verwendete Implementierung von malloc() Speicherbereiche nicht byte-weise reservieren sondern kann auch mit größeren Blöcken arbeiten.
Folgende C-Compiler wurden genutzt: avr-gcc 4.3.2 für Atmel AVR, SDCC 2.8.0 #5117 für Intel 8051, GCC 4.3.3 für Intel i386, GCC 4.3.3 für AMD x86_64. Alle Compiler wurden über die entsprechenden Aufrufparameter angewiesen den Maschinencode so kompakt wie möglich zu halten.

Wie aus der Tabelle ersichtlich wird das Beispiel auf allen Plattformen in ähnlich viel Maschinencode übersetzt, je nach Gerät gelten aber verschiedene Maßstäbe: Der Atmega8 etwa besitzt nur 8 KiloByte Flash-Speicher und ein KiloByte RAM. Diese Größe reicht für eine ganze Reihe von Projekten aus, beispielsweise wurde ein MP3-Player mit diesem Controller realisiert. Auf der anderen Seite reicht der Speicher allerdings nur für zwei oder drei Mock-Objekte mit zugehörigen Unit-Tests und notwendigen Sub-Systemen (z.B. Kommunikation mit dem Host) aus. Einige Vertreter des 8051 verfügen über nur 128 Byte RAM, auf diesen Geräten ist der Einsatz von Unit Tests und CMock teilweise unmöglich, da nicht genug Speicherplatz für das Training zur Verfügung steht.
Die meisten größeren Projekte setzen allerdings auf Umgebungen mit weniger restriktiven Limitierungen auf, etwa den Atmega32 (32 KiloByte Flash, 2 KiloByte RAM) oder fortgeschrittenere Modelle (ARM, MIPS, ColdFire etc.). Intel i386 und AMD x86_64 sind als Beispiele für Cross-Compiling auf Entwickler-PCs aufgeführt, wo Speicher üblicherweise kein Problem darstellt.

## IX. Limitierungen und Fazit

Der Inhalt dieses Dokumentes zeigt, dass CMock die grundlegenden Anforderungen an ein Mock Framework für die Programmiersprache C erfüllt. Dem Einsatz in eingebetteten Systemen stehen allerdings möglicherweise mehrere Argumente entgegen.
CMock setzt sehr stark auf malloc(), realloc() und free() aus der Standard Library. Nicht alle Laufzeitumgebungen bieten diese Funktionalität an, in einigen Projekten ist der Einsatz aus Sicherheitsgründen verboten - sowohl in Test-Drivern als auch in Produktiv-Code. Derzeit existiert hierfür keine Lösung.
Die starke Abhängigkeit von Unity erzeugt möglicherweise Konflikte wenn in einem existierenden Projekt bereits ein anderes Unit Test Framework eingesetzt wird oder Unity nicht mit den Projektregeln vereinbar ist. Obwohl die Code-Erzeugung vollständig über Plug-Ins abgewickelt werden sollte ist dies nicht an allen Stellen der Fall, ein kompletter Verzicht auf Unity ist daher im Moment nicht möglich.

Das CMock Framework ist ein noch recht junges Projekt und wurde seit der Freigabe aktiv weiterentwickelt. Da der Einsatz von Unit Tests und Mocks im Embedded-Umfeld derzeit aber in vielen Unternehmen noch kein Thema ist wird sich in Zukunft noch zeigen müssen ob CMock das Framework der Wahl ist oder einer der Mitbewerber das Feld erobert.

## References

[BM00] Andy Bower and Blair McGlashan, editors. *Twisting The Triad - The evolution of the Dolphin Smalltalk MVP application framework.* ESUG 2000 International Smalltalk Conference, 2000.

[Dow04] Micah Dowty. Test driven development of embedded systems. University of Colorado at Boulder, March 2004.

[FBKW07] Matt Flettcher, William Bereza, Mike Karlesky, and Greg Williams, editors. *Evolving into Embedded Development*. Agile 2007 Conference, August 2007.

[Gre02] James W. Grenning, editor. *XP and Embedded Systems development*. Object Mentor Inc., 5101 Washington Street, Suite 1108, Gurnee, IL 60031 USA, 1 edition, March 2002.

[KBE06] Michael J. Karlesky, William I. Bereza, and Carl B. Erickson, editors. *Effective Test Driven Development for Embedded Software*. IEEE Electro/Information Technology Conference, 2006.

[WV08] Greg Williams and Mark VanderVoord, editors. *Embedded Feature Driven Design Using TDD and Mocks*. Embedded Systems Conference Boston 2008, October 2008.