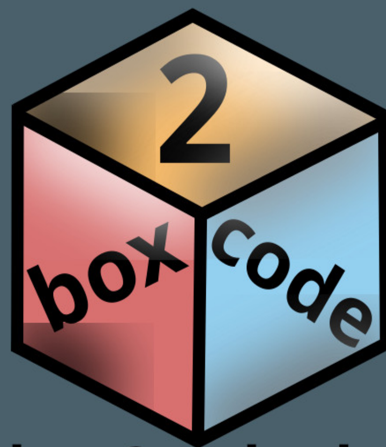


Firmware mit C++

Stefan Scholz - 2017

Inhaltsverzeichnis

Firmware mit C++ 1.1
Missverständnisse 1.2
Auf geht's - Praxis 1.3
Wie Testen 1.4

Einleitung

Dieser Artikel richtet sich Vorrangig an Entwickler die Firmware in C entwickeln, C++ vielleicht schon ausprobiert haben und enttäuscht wieder zu C zurück sind. Meiner Meinung nach eignet sich C++ hervorragend für die embedded Entwicklung. Allerdings ist es in C++ auch leicht sich ins eigene Knie zu schießen. Ich werde versuchen einen Ansatz zu beschreiben, der einfach umsetzbar ist und dabei möglichst die Vorzüge von C++ zum Vorschein bringt.

Warum C++

- Objektorientierung
- Leichter wiederverwendbar
- Weniger Code
- Leichter lesbarer Code
- Leichter testbarer Code

Warum dieser Artikel

Ich hab diesen Artikel am 23. September 2017 in das Firmenwiki meines Arbeitgebers geschrieben (natürlich in meiner Freizeit - war ein Samstag sagt die Versionsgeschichte :-)) Grund war, dass ich bei einigen Projekten zwar C++ für Firmware eingesetzt sah - diese aber für unnötig kompliziert hielt.

Es gab dort Vorgehensweisen wie:

- Jedes Modul ist ein Singleton
- Jedes Modul hat einen Konstruktor mit allen Abhängigkeiten - mit der Argumentation, dass man sonst nicht Testen kann
- Es gab sehr viele Tasks - teilweise sogar für einfache Treiber
- Es war schwer im Code eine logische Struktur zu erkennen

Missverständnisse

In C++ muss jede Funktion eine Methode einer Klasse sein - sonst ist das nicht objektorientiert

C++ ist eine Multiparadigmensprache und das es freie Funktionen gibt ist eine der absoluten Stärken. Ich hoffe ich stoße auf Zustimmung wenn ich behaupte eine Funktion ist wesentlich einfacher zu testen als eine Klasse, vorallem da es keinen internen Zustand gibt. Die STL und boost sind gute Beispiele dafür so sind Algorithmen wie std::sort oder std::find einfach freie Funktionen. Seit einiger Zeit wird versucht das Interface vieler Containerklassen auf das nötigste zu reduzieren und dafür freie Funktionen anzubieiten. Dies erhöht die Kapselung der jeweiligen Klasse da deren Interface dadurch schmäler wird. In neueren Compilerversionen sind cont.begin() und std::begin(cont) gleichwertig.

Um Code wiederzuverwenden muss ich Vererbung einsetzen

Das Mittel der Wahl für Wiederverwendung ist die Komposition. Vererbung wird oft missverstanden - das Problem ist für eine Korrekte Vererbung muss die abgeleitete Klasse jederzeit an Stelle der Elternklasse treten könnnen und das ist oft nicht gewährleistet und verursacht dann später große Probleme.

Wo dagegen überhaupt nichts einzuwenden ist, ist die Interfacevererbung. Wenn eine Klasse in der Lage ist ein Interface zu bedienen kann sie normalerweise an jeder Stelle verwenden, an der dieses Interface benötigt wird und das ist super.

Ich weiß es gibt Gegenbeispiele wie zum Beispiel die Verwendung von Vererbung in GUI-Bibliotheken, in der man einfach von einem Knopf oder Fenster ableitet um diese zu modifizieren. Ich begründe das an dieser Stelle einfach mal damit, dass wohl die entsprechenden Set-Methoden fehlen auch wenn die Vorstellung grausig ist ;-)

Der beste Spruch den ich zu diesem Thema je gehört habe ist:

> Vererbe nicht um wiederzuverwenden sondern um wiederverwendet zu werden !!

Genau das ist auch wieder der der Punkt mit den GUI-Bibliotheken. Ich vererbe um z.B. in einem LayoutManager wiederverwendet zu werden oder auf grafische Events reagieren zu können - nicht unbedingt um möglichst viel Code wiederzuverwenden.

Ein Softwaresystem sieht aus wie ein Baum

Im Studium habe ich noch gelernt:

> es gibt ein Systemobjekt, dieses hat Subsysteme und diese bestehen wieder aus Modulen um alles miteinander zu verbinden werden Konstruktoren eingesetzt

Das ist der völlig falsche Ansatz - es ist so unendlich schwer in einer solchen Struktur etwas hinzuzufügen oder zu entfernen. Außerdem geht jegliche Kommunikation von Objekten, die eigentlich direkt sein sollte tausend Umwege.

Was ich normalerweise anstrebe ist ein Design, das eher aussieht wie eine Digitalschaltung. Jedes Objekt kann, wie ein IC, für sich alleine stehen (Hat also einen Konstruktor, der im Idealfall keine anderen Objekte benötigt). Dann werden diese einfach miteinander verknüpft. Die Abhängigkeiten also sozusagen injiziert.

Der Vorteil ist - braucht man ein neues Objekt das eine bestimmte Aufgabe erfüllt, fügt man es einfach hinzu und legt ein paar Drähte um. So ist's super

Das ist natürlich nicht meine Erfindung und jeder der die hervorragende QT-Bibliothek etwas kennt, weiß wovon ich rede ;-)

Die Initialisierung des Gesamtsystems passiert bestenfalls an nur einer Stelle z.B. in main() oder in einem Application-Objekt - je nach Gusto.

Ein Interface ist einfach die Kopie aller public Methoden einer Klasse

Nein !! genau das nicht desto kleiner ein Interface, desto leichter kann ich dieses woanders wieder verwenden. Auch beim Testen hat dies große Vorteile.

Ich schreibe ein Interface nicht für die Klasse selbst, sondern für eine Anfrage die von einer anderen Klasse ausgeht. Aber das sehn wir dann später im Architekturbeispiel.

Das Konzept nennt sich Interface-Segregation für den gebildeten Eindruck ;-)

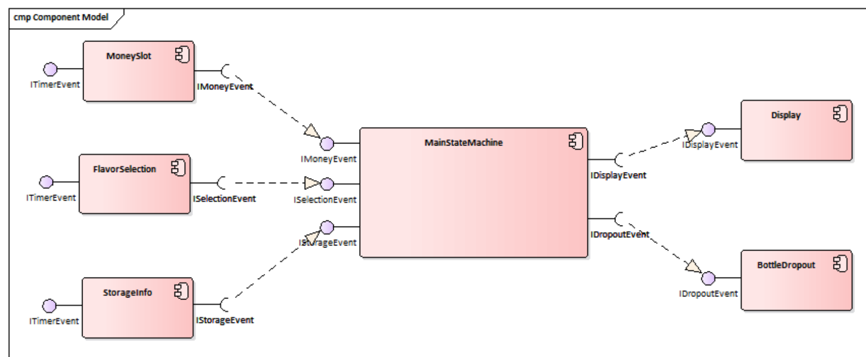
Also für jede Klasse auch noch ein Interface ?

Nein, das macht keinen großen Sinn. Um mit manchen Klassen zu arbeiten muss man sie einfach komplett kennen. Mein Lieblingsbeispiel hierfür ist eine String-Klasse. Diese hat norwalerweise keine Abhängigkeiten und lässt sich auch gut für sich alleine testen. Genauso irgendwelche Klassen, die irgendwelche Daten transportieren, Container etc. Ich nenne Sie einfach mal Utility-Klassen.

Die anderen, für die ein Interface sinnvoll ist, könnte man vielleich als Management-Klassen bezeichnen.

Auf geht's - Praxis

Wir bauen ein ereignisgesteuertes System: Einen einfachen Getränkeautomat Interrupts, DMA und Ähnliches wären zum Verständnis etwas hinderlich - deswegen pollen wir einfach das Zeug alle 100ms - ich hoffe das ist erstmal okay so.



Ein Wort zu Interrups

In diesem Beispiel werde ich Interrupts nur simulieren. Wichtig ist, dass man Interrupts nie direkt Code ausführen lässt. Bei FreeRTOS zum Beispiel würde ich empfehlen in einem Interrupthandler nur einen Event in eine Queue zu packen. Ein Task ist dann dafür zuständig Event-Elemente aus der Queue zu nehmen und an das System zu schicken. Er übernimmt dann die Rolle eines Dispatchers. Bei mbed-os gibt es die noch viel elegantere EventQueue für diese Zwecke. Bei Arduino sieht TaskManagerIO recht nett aus.

Die Events in event.h

```
struct TimerEvent
{
};

struct MoneyEvent
{
};

struct SelectionEvent
{
    uint8_t selection;
};

struct StorageEvent
{
    bool fillStatus;
}

enum class Message
{
    Empty = 0,
    InsertMoney = 1,
    SelectFlavor = 2,
    ...
}

struct DisplayEvent
{
    Message msg;
}

struct DropoutEvent
{
    uint8_t selection;
}
```

Die Interfaces in interfaces.h

Kreativität beim benennen der Interfaces wird völlig überbewertet. Manchmal ist etwas Monotonie sogar sehr hilfreich. Man muss einfach später nicht mehr überlegen, wie man die Methode jetzt genannt hat und hat den Kopf frei für andere Dinge.

```
class IMoneyEvent
{
public:
  virtual ~IMoneyEvent() {}
  virtual void react(const MoneyEvent &evt) = 0;
};

class ISelectionEvent
{
public:
  virtual ~ISelectionEvent() {}
  virtual void react(const SelectionEvent &evt) = 0;
};

class IStorageEvent
{
public:
  virtual ~IStorageEvent() {}
  virtual void react(const StorageEvent &evt) = 0;
};

// ... Der Rest sieht genauso aus
```

Systeminitialisierung - hier einfach in main()

```
int main()
{
    // Objekte auf dem Stack anlegen

    Display display;
    MoneySlot moneySlot;
    FlavorSelection selection;
    BottleDropout dropOut;
    StorageInfo storageInfo;
    MainStateMachine mainStateMachine;

    // Verbindungen herstellen


moneySlot.registerMoneyEventHandler(&mainStateMachine);

selection.registerSelectionHandler(&mainStateMachine);

storageInfo.registerStorageEventHandler(&mainStateMachine
);


mainStateMachine.registerDisplayEventHandler(&display);

mainStateMachine.registerDropoutEventHandler(&dropOut);

    // Und action

    TimerEvent timerEvent;

    while (true)
    {
        moneySlot.react(timerEvent);
        selection.react(timerEvent);
        storageInfo.react(timerEvent);
        wait_ms(100);
    }
}
```

Beispiel für eine Komponente

Dies ist natürlich nur die einfachst mögliche Implementierung. Es macht Sinn auch mehrere Handler registrieren zu können. Aber an dieser Stelle reicht das wohl erstmal aus. Ist kein Handler registriert, dann ist ein assert völlig angemessen. Denn es weist darauf hin, dass wir unser eigenes System falsch zusammengebaut haben. Viel Fehlerbehandlung ist daher nicht angebracht.

```
class MoneySlot : public ITimerEvent
{
public:

    MoneySlot() : m_moneyEventHandler(NULL) {}

    void react(const TimerEvent &evt)
    {
        assert(m_moneyEventHandler);
        if (coin())
        {
            MoneyEvent evt;
            m_moneyEventHandler->react(evt);
            acceptCoin();
        }
    }

    void registerMoneyEventHandler(IMoneyEvent
*meHandler)
    {
        m_moneyEventHandler = meHandler;
    }

private:

    IMoneyEvent *m_moneyEventHandler;
};
```

Wie wird die Statemachine implementiert

Für die Statemachine habe ich mir tinyfsm ausgesucht. Eine winzige header-only Bibliothek. Diese verwendet zwar ein paar Templates - ist aber gut zu verstehen und besonders einfach in der Verwendung. Abhängigkeiten gibt es nahezu keine.

Wir sagen einfach es gibt drei Zustände:

- Idle - Der Automat wartet auf eine Benutzeraktion - Display zeigt: "Münze einwerfen"
- MoneyInserted - Der Benutzer hat eine Münze eingeworfen - Display zeigt: "Getränk wählen"
- Deliver - Die Flasche wird ausgeworfen und es wird wieder nach Idle gewechselt.

```
// Header

class MainStateMachine : public
tinyfsm::Fsm<MainStateMachine>, IMoneyEvent,
ISelectionEvent
{
public:

    virtual void entry(void) { };

    virtual void react(const MoneyEvent &evt) {}

    virtual void react(const SelectionEvent &evt) {}

    virtual void exit(void) { };
};

// Implementierung

class Idle : public MainStateMachine
{
public:

    void entry()
    {
        displayInsertCoin();
    }

    void react(const MoneyEvent &evt)
    {
        transit<MoneyInserted>();
    }

};

class MoneyInserted : public MainStateMachine
{
public:

    void entry()
    {
        displaySelect();
    }

    void react(const SelectionEvent &evt)
    {
        transit<Deliver>();
```

```
    }

};

class Deliver : public MainStateMachine
{
public:

    void entry()
    {
        openBottleDispenser();
        closeBottleDispenser();
        transit<Idle>();
    }
};

FSM_INITIAL_STATE(MainStateMachine, Idle)
```

Wie Testen ?

Hier ein einfacher Test - verwendet GoogleMock und GoogleTest

```
class MockDisplay : public IDisplayEvent
{
public:
    MOCK_METHOD1(react, void(const DisplayEvent &evt));
};

class MockBottleDropout : public IDropoutEvent
{
public:
    MOCK_METHOD1(react, void(const DropoutEvent &evt));
};

TEST(MainStateMachineTest, DoesItWork)
{
    MockDisplay display;
    MockBottleDropout dropout;

    EXPECT_CALL(display, react(const DisplayEvent
&evt)).Times(AtLeast(1));
    EXPECT_CALL(dropout, react(const DropoutEvent
&evt)).Times(AtLeast(1));

    MainStateMachine mainStateMachine;


mainStateMachine.registerDisplayEventHandler(&display);

mainStateMachine.registerDropoutEventHandler(&dropOut);

    MoneyEvent moneyEvent;
    SelectionEvent selectionEvent;

    mainStateMachine.react(moneyEvent);
    mainStateMachine.react(selectionEvent);
}

int main(int argc, char** argv)
{
    ::testing::InitGoogleMock(&argc, argv);
    return RUN_ALL_TESTS();
}
```