C++ FOR (Friday Morning)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

> Da der Nachmittagsteil am Freitag entfällt, folgt am Ende dieses Abschnitts keine Übung.

Pragmatische Leitgedanken zur Software-Entwicklung

- "Open-Close"-Principle

- Don't Repeat Yourself

"Open Close"-Prinzipie

Gemäß diesem Prinzip sollte jede Software-Architektur einen gesunden Ausgleich zwischen zwei gegeneinander stehende Zielen gewährleisten:

- Software sollte *offen* für Veränderungen sein, beispielsweise
  - Anpassung an künftig geänderten Bedarf,
  - absehbare, anstehende Erweiterungen und
  - Verwendung in ähnlich gelagerten Fällen.
- Software sollte aber auch *robust* sein in dem Sinne, dass
  - Änderungen nicht versehentlich oder in ansonsten unbeabsichtigter Weise erfolgen;
  - zumindest sollten unbeabsichtigte Änderungen leicht zu identifizieren sein;
  - ebenso solche Änderungen, die zielgerichtet im Rahmen der Offenheit erfolgen aber aus irgend einem Grund unvollständig geblieben sind.

Mechanismen zur Strukturierung

Klassen

Klassen fassen "Daten" und "Verarbeitung" zusammen und sind somit ein Mechanismus zur Kapselung, der den Blick auf die abstrakten Operationen lenkt, weg von Datenstrukturen und Algorithmen.

Unterprogramme

Unterprogramme teilen Verarbeitungsschritte auf, vom komplexen Gesamtablauf bis hinunter zu kleinen, einfach zu überschauenden und gut testbaren Einheiten.

Bibliotheken

Bibliotheken sind Sammlungen wiederverwendbarer Komponenten, die für sich betrachtet kein "Eigenleben" führen sondern erst "von außen" zum Leben erweckt werden.

Mechanismen zur Strukturierung (2)

Frameworks

Frameworks folgen dem "*Hollywood-Principle*": Don't call us, we call you.

Sie stellen eine oft eine relativ komplexe Gesamtfunktionalität zur Verfügung und enthalten *Erweiterungspunkte*, an denen individuelle Anpassungen erfolgen können.

Geplante Erweiterbarkeit

JA – aber auch: "*Keep It Small and Simple!*"

Beginnen Sie also stets mit der **einfachsten Variante**

- einer Klasse,
- eines Algorithmus,
- einer Applikation,
- eines Programmsystems,
- ...

welche Ihr Problem (gerade so) löst, und erweitern Sie diese ggf. inkrementell gemäß neu erkanntem Bedarf.

Don't Repeat Yourself

Eine wichtige Erkenntnis zur erfolgreichen Arbeitsteilung zwischen "*Mensch und Computer*" ist:

Menschen

- besitzen oft ein hohes Maß an Kreativität,
- sind aber in aller Regel schlecht darin, Dinge präzise zu wiederholen,
  - sei es in immer wieder ein- und derselben Weise,
  - oder auch mit kleinen, systematischen Variationen.

Computer

- besitzen kaum echte Kreativität,*
- sind aber extrem gut darin, Dinge präzise zu wiederholen:
  - Insbesondere ermüden sie nicht bei ständig wiederholten und
  - dabei allenfalls leicht variierenden Tätigkeiten.

---

*: Es ist dabei nebensächlich, dass per Computer gelegentlich "überraschende Ergebnisse" erzielt werden können – etwa in der Art, dass ein Computer ein Musikstück komponieren könnte, welches es vielleicht sogar in die Hitparade schafft. Interessanter ist die Frage nach einem Programm, mit welchem ein Computer im Dialog mit einem menschlichen Partner von diesem nicht schon sehr bald als Maschine enttarnt wird, wie es im Fall der Turing-Tests letzten Endes doch immer wieder geschieht.

Mechanismen zur Wiederverwendung

Klassen

Relativ allgemein und universell gehaltene Klassen bilden in der Regel die kleinsten wiederverwendbaren Bausteine im Rahmen der objektorientierten Vorgehensweise.

Unterprogramme

In einem eher klassischen, prozeduralen Entwurf dominieren Unterprogramme, mitunter ergänzt durch zugehörige Datenstrukturen.

Datenstrukturen

Mit Hilfe der C++ Templates lassen sich sehr gut wiederverwendbare Bausteine für Datenstrukturen realisieren.*

---

*: OK ... technisch gesehen sind die STL-Container natürlich Klassen, allerdings geht es dabei vor allem um Typ-Generizität in Bezug auf häufig erforderliche Datenstrukturen.

C++ as Multi-Paradigm Language

Commonly C++ is viewed as **Object Oriented Programming** Language.

This is correct, but only part of the truth, because C++ also*

- is a classical **Procedural Programming** Language,
- in which nearly all of **C is contained as Sub-Set**,
- thereby allowing to get **optionally close to the Hardware** …
- … with many compilers allowing to actually **mix in Assembler Code**,
- on the other hand supports **Generic Programming**
- and **Meta-Programming** via **Templates**,
- C++11 even adopted elements of **Functional Programming**,
- and sometimes its **Preprocessor** comes in handy as last resort.

This is both, good and bad at the same time:
While it allows to chose the paradigm most appropriate to solve a given problem, (not only) novices may easily make a bad choice and end up much worse as if there had been not so many options.

*: For a nice reading – musing and amusing at the same time – lookup Execution in the Kingdom of Nouns by Steve Yegge.

Choosing the Right Paradigm

There is no easy answer but what you can do is:

- Gather Experience!
- Allow yourself to fail – but be sure to learn the lesson.
- Apply corrections as soon as things start to run in the wrong direction.

> In mission critical projects consider to hire a C++ expert for coaching.

And: Do not blame the C++ language for "insufficient support" of a certain style you know and like from a different programming language, which you personally happen to favour.*

---

*: If you are a team leader, place all your team's discussions about the "right" programming language and C++ "deficiencies" early within your time frame. Maybe there is actually a better choice as C++, if for the whole project, take it! Otherwise C++ may not be appropriate or best for parts of the project. Consider to give those members who are precious for any reason but strongly opposed to C++ their own corner, where they can work with what they like most, be it C#, Java or Scala, Haskell, Lisp ...
But at a point **declare all discussions closed** and have your team members commit themselves to C++ ... from which they still can and should chose the appropriate mix of paradigms, of course.

How to Recognize that Things Start to Go Wrong?

See the first section of this chapter for the favourable deeper guiding principles of any software development.

If you observe increasing failure to reach these goals, you might have chosen

- **the wrong** (combination of) **C++ paradigm**(s) or
- C++ is an inadequate programming language*

for the problem at hand.

*: In C++ the former is far more probable, though – of course – the latter is not impossible, especially if the general team knowledge or commitment is not C++ centric.

Design Patterns – Critically Reviewed

To begin with:

> Design patterns are a good thing, really!

Having said that, be sure to understand the following:

- Each design pattern has a purpose – understand which it is!
- Each design pattern has a context – make sure the context of the problem you want to solve matches.

> Do not slavishly apply patterns just because "patterns are good".

The GoF Book

Design patterns became famous in the second half of the 1990s through the GoF-Book.*

While this book surely filled a gap that had already been open far too long at that time, it should not be overestimated:

- At the time the GoF book was published many C++ compilers already supported templates. But templates were not in that widespread use as later, after they became part of the C++98 standard.
- Therefore most any abstraction in the GoF book is via base classes and virtual member functions (aka. late binding, dynamic polymorphism, etc.) – with consequences detailed on the next pages.
- The authors even remind to that fact in their book's introductory chapter (pg. 21/22 in the 1994 edition), but some of the more slavish GoF followers seem to have skipped reading that part.

---

*: *GoF* is the abbreviation for "*Gang of Four*" and honours the fact that, while **four authors** contributed, those who referred to the book often were too lazy to remember (or enumerate) all authors. Here they are: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Design Patterns and the GoF Book – A Personal Opinion

It seems, design patterns generally and the GoF book in particular can be understood in two different ways:

1. **Train the Brain** to recognize recurring structures in software projects and know what to do about them, i.e.
    - to which degree the pattern makes sense in a given context,
    - when it might be especially appropriate,
    - when not, where are its limits, pitfalls,
    - what are the alternatives,
    - how some pattern relates to other patterns, ...
2. **Implement the pattern in a specific way**:*
    - No doubt, the implementation style shown and discussed is best practice in *SmallTalk*.
    - It may also be appropriate for *Java* (therefore *Scala* too), *C#*, ... maybe *Python* ... *Objective-C++* ...
    - ... but it needs to be taken with the proverbial "grain of salt" for C++, mostly because of multi-paradigm nature of that language.

---

*: Or what I call *GoF book style* on the pages following – and what is the particular target of my criticism.

Shortcomings of the GoF Book Implementations

The implementation style for the patterns discussed in the GoF-Book tends to shift type-safety from compile-time to run-time.*

> If you happen to be a GoF book fan and think the word "shortcoming" sounds depreciating, feel free to replace it by something more neutral.

- **There is a – sometimes small, sometimes big – gain:**
  - It may help to substantially reduce the modules to recompile in a large software system after a tiny or even moderate change.
- **There is a – sometimes big, sometimes small – loss:**
  - Problems reaching from slightly flawed designs to subtle or even quite obvious implementation errors may only show when the software runs, not yet when it is compiled.

*: In all fairness and not to lift the burden to give better error messages for problems when compiling C++ templates, chose from the following two scenarios: **1.** You sit for hours and are close to desperation because of an error message the compiler throws at you for a template and you will probably have to call your boss now, telling her the major release planned for tomorrow will be delayed. **2.** You come in in the morning and everybody is already impatiently waiting for you because production came to a grinding halt two hours ago ... and all you have to start with is a core dump of some software you wrote.

Design Patterns and OOP-Languages

Because of the GoF book prevalence in the design patterns world, some of its proponents measure the "quality" of a programming language by the degree to which it supports dynamic polymorphism and introspection – both no absolute strengths of C++.

- The GoF book – though it seems to relate to C++ as all of its examples are in C++ and only some few are in SmallTalk too – still has a strong connection to the world of SmallTalk.
- The market share of SmallTalk was already declining in the mid 1990s, mostly caused by the rise of Java and later C#.

In the years following, in their spirit – not syntax – Java and C# advanced to get much closer to SmallTalk as C++ ever tried.*

> Considering design patterns only in GoF book style, while dropping C++-Templates mostly or completely, makes C++ surely look inferior.

*: There is a famous quote of Bjarne Stroustrup, who – when asked about SmallTalk and whether C++ should be extended in this direction too – had answered: *SmallTalk is the best SmallTalk that exists.*

DP-Examples

In the following a small subset of classical design patterns is discussed with respect to how they relate to the multi-paradigm nature of C++.

> A complete and detailed discussion is far beyond this presentation – there are other courses with more in-depth coverage of the topic.

The DP examples following have **not** been selected because they are

- the ones "most typically" or
- "most frequently" used,

nor do they represent

- particularly "good" or
- particularly "bad" examples

for design patterns in C++.

DP-Example: Iterator

Using Iterators to run through sequences is long-standing practice.

There are two main advantages:

- Type-Safety and
- Abstraction.

> While it is often worthwhile to consider implementing iterators as nested helper class for new kinds of containers, there is very little reason in C++ doing this in GoF book style.

When implementing new container types, determine the appropriate iterator category (i.e. input or output, uni- or bidirectional, ...) and adhere to the category's specific requirements.*

> (i) Then – and **only then** – the new container type is immediately ready for using it with all appropriate STL algorithms.

*: Boost.Iterator may help to avoid some of the more schematic work.

DP-Example: Observer

At its core the observer pattern is about decoupling a number of otherwise independent components or sub-systems so that each one doesn't have to know too many details of the other one.

For dynamic control over coupling* an implementation may simply use C++11 `std::function` (or Boost.Function in C++98) as STL container element and a tiny loop (approx. two lines of code) for message dispatch.

A ready-to-use library approach taking this road is available with

- Boost.Signals and
- Boost.Signals2,

though it supports some extras – like flexible result collectors – hampering performance in a direct comparison to more light-weight approaches without such features.

- See https://testbit.eu/cpp11-signal-system-performance

---

*: Sensibly decoupling components with static connections to each other is more or less a developer's "every day job" (if she understands the requirements of good software engineering); if that were subsumed under the observer pattern, any non-trivial software would be full of observers ...

DP-Example: Composite

The composite in GoF book style ties together its various leaf types with a common base class – including a collection that enables recursive nesting.

> This puts restrictions on the types that can be used – or at least requires wrappers for unrelated classes, so they can have a common parent.

Boost.Variant offers an elegant alternative:

- It can easily tie together any unrelated basic types and classes,
- (therefore commonly used library classes like `std::string` too),
- and even callable code via `std::function`.

For recursive nesting any STL container may be added, e.g. an `std::map`

- holding the variant as its value part,
- keyed with an `std::string` for access.*

(A ready-to-use library solution to consider, especially for composites holding persistent state or configuration data, is Boost.Property_Tree.)

*: In other words: more or less mimicking the core of Python's object model. For some example code (rather meant as proof of concept but to be used as is) see Examples/DynamicVariant/dv-poc.cpp

DP-Example: Template-Method

From the general perspective the design pattern called *Template-Method* is an ideal example for the Open-Close Principle.

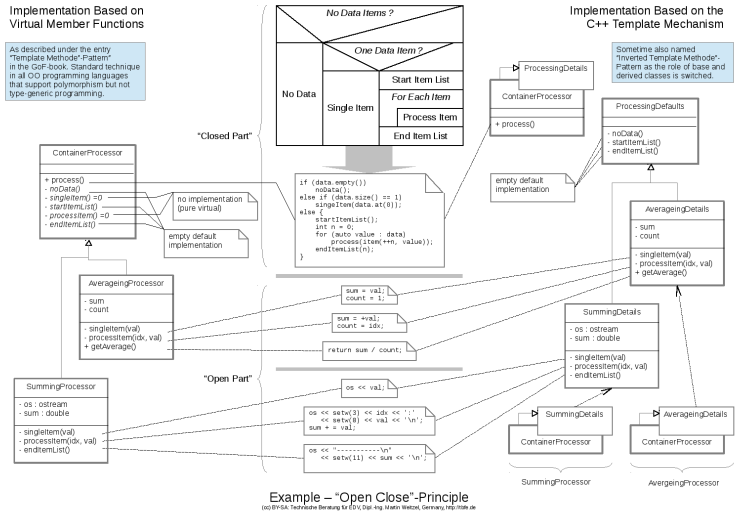
> **[!]** Be careful not to confuse the name given to this pattern with C++-Templates – the similarity is purely accidental.*

What especially may contribute to confusion is that one of the two typical implementations actually makes use of C++-Templates.

*: It may be even assumed that Erich Gamma and his three co-authors may have chosen a different name if C++-Templates had already been in widespread use at the time they worked on their manuscript.

DP Template-Methode Implementation Alternatives

- Based on Dynamic Polymorphism
- Based on C++-Templates



DP Template Method Pattern Based on virtual Member Functions

In the classical implementation* of the GoF Template Method Pattern

- virtual member functions of a base class are
  - pre-planned *Extension Points*,
- at which specific derived classes
  - attach a different functionality, as required.

With respect to the Open-Close-Principle the base class is in the role of the closed part and the various derived classes contribute the open part.

> **[!]** A certain drawback of this technique is that any unused extension point will remain in the executable code as (indirect) call to an empty subroutine.

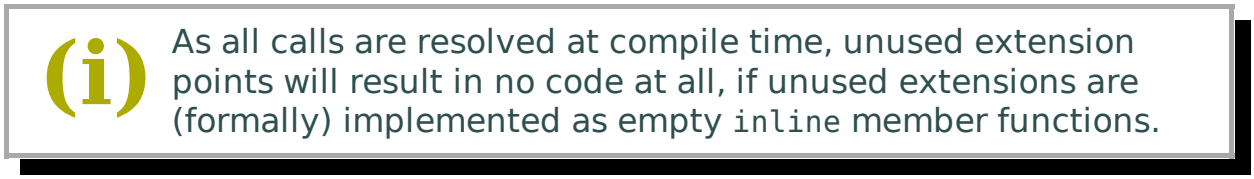
*: See Examples/OpenClose/virtual_functions.cpp

DP Template Method Pattern Based on C++-Templates

When implemented with C++-Templates*

- the derived class is
  - a generic class with pre-planned *Extension Points*, expected as member functions in a base class
- specified only later through a type parameter,
  - therefore attaching different functionality through instantiation with different base classes.

With respect to the Open-Close-Principle the derived class is in the role of the closed part and the various base classes contribute the open part.



*: See Examples/OpenClose/template_baseclass.cpp

DP-Example: State

There are many ways to implement state machines, e.g.

- in "pure C" with nested `switch`-statements (outer for states and inner for events or vice versa) as the most straight-forward approach,
- as a more readable and better maintainable "pure C" alternative with tables of function pointers,
- with the suggested GoF book style being only one of many other alternatives and variants.*

In his book about the Quantum Framework Miro Samek discusses in great depth various possible implementations of state machines in C and C++, also pointing out chances to improve and enhance the GoF book style by using C++-Templates.

*: Of special interest for C++ developers may also be Boost.Fsm and Boost.Msm, which both implement state machines applying C++ meta programming techniques. The architecture of the latter even allows to chose among several *front-ends* (i.e. *DSL-s* to specify the states and transitions) and *back-ends* (i.e. drivers to execute the machine according to incoming events). The default front-end uses a tabular layout similar to the one achievable with a pure C approach using tables of function pointers.

DP-Example: Singleton

The singleton pattern seems easy to understand and apply, but (or maybe because of that) is also the one most often misused or at least used as a much too heavy weight approach to just solve the problem at hand.

In C/C++ `static` local variables provide a compact alternative:*

```
MySingleton &getMySingletonInstance() {
   static MySingleton instance;
   return instance;
}
```

The above and some variations are available as a small series of examples in Examples/Singleton.

> In his book Modern C++ Design Andrei Alexandrescu spends a full chapter on the singleton. Some obvious and some not so obvious problems with the pattern are addressed and possible solutions shown.

*: Despite this simple and elegant alternative exists in C++ (and even in C, if the reference is replaced with a pointer), full-blown GoF book style singletons are not unusual, even in scenarios where a plain "good ol' global" (variable) would have sufficed.

Design Patterns – Closing Remark

To end with:

Design patterns are a good thing, really …

- … when used for the intended purpose …
- … within the appropriate context …
- … implemented in "the C++ way" …

… but probably not so

- as vastly over-engineered solution to a trivial problem,
- if some ready-to-use library solution is thrown overboard because it doesn't follow (seemingly) "best practice" of the GoF book style, or
- if pointing to the latter is used as an argument to radically limit the freedom to chose from the set of paradigms C++ supports.

Summary and Epilogue

- Don't repeat yourself.
- Keep software "open" where you expect future modifications ...
- ... but make it "close" where inadvertent changes could break it.
- Start with the simplest possible solution – don't over-engineer.
- Build on reusable parts, do not invent the wheel over and over again.
- Follow best practices, but be aware of the context in which they apply.
- And don't repeat yourself (oops).

☺ Have fun with C++ ☺