Other Stuff

- Design Patterns in C++
- Late Binding vs. Templates
- Domain Specific Languages
- Preprocessor for Code Generation

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.

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-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 …

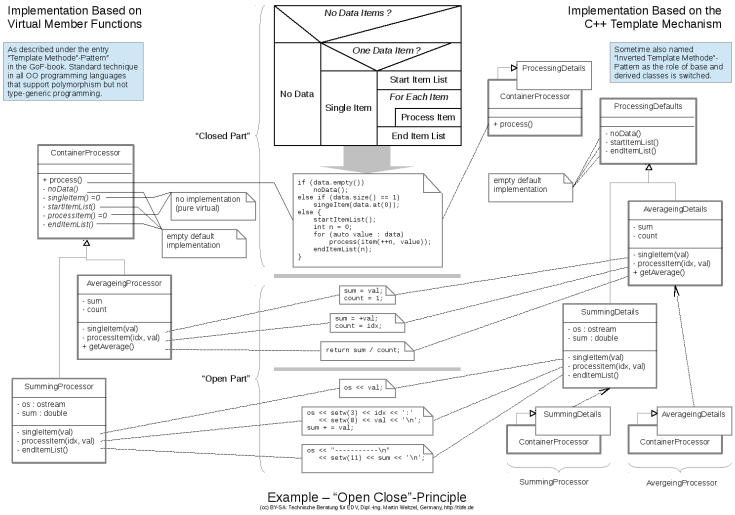
- … when used for the intended purpose …
- … within the appropriate context …
- … implemented in "the C++ way" …

… 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.

Late Binding vs. Templates

- 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.

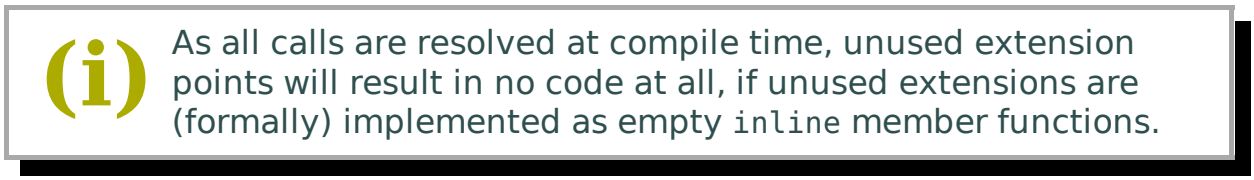
*: See Examples/OpenClose/virtual_functions.cpp

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

Domain Specific Languages

Some Boost libraries are prominent may serve as examples for domain specific languages (DSLs):

- Boost.Spirit for Parsers and Generators
- Boost.Statechart and Boost.Meta_state_machine for State Machines

Furthermore Boost has libraries meant as helpers to **build** DSLs:

- Proto (an EDSL Framework)
- Property Map

Parsers and Generators

From the Boost Documentation:

> Boost.Spirit is an object-oriented, recursive-descent parser and output generation library for C++. It allows you to write grammars and format descriptions using a format similar to Extended Backus Naur Form (EBNF) directly in C++. These inline grammar specifications can mix freely with other C++ code and, thanks to the generative power of C++ templates, are immediately executable. In retrospect, conventional compiler-compilers or parser-generators have to perform an additional translation step from the source EBNF code to C or C++ code.

State Machines

From the Boost Documentation:

> Boost.MSM is a library allowing to easily and quickly define state machines of very high performance.

Proto (EDSL Framework)

From the Boost Documentation:

> Boost.Proto is a framework for building Embedded Domain-Specific Languages in C++. It provides tools for constructing, type-checking, transforming and executing expression templates. More specifically, Proto provides:
>
> - An expression tree data structure.
> - A mechanism for giving expressions additional behaviors and members.
> - Operator overloads for building the tree from an expression.
> - Utilities for defining the grammar to which an expression must conform.
> - An extensible mechanism for immediately executing an expression template.
> - An extensible set of tree transformations to apply to expression trees.

Property Map

From the Boost Documentation:

> The Boost.Property_map Library consists mainly of interface specifications in the form of concepts (similar to the iterator concepts in the STL). These interface specifications are intended for use by implementors of generic libraries in communicating requirements on template parameters to their users. In particular, the concepts define a general purpose interface for mapping key objects to corresponding value objects, thereby hiding the details of how the mapping is implemented from algorithms. The implementation of types fulfilling the property map interface is up to the client of the algorithm to provide. The property map requirements are purposefully vague on the type of the key and value objects to allow for the utmost genericity in the function templates of the generic library.

Fortgeschrittene Nutzung des Präprozessors

- Der C++-Präprozessor beherrscht kein C++
- Verwendung von Makro-Argumenten als String-Literal (Stringizing)
- Verketten von Makro-Argumenten zu neuen Tokens (Token Pasting)
- Systematische, tabellengesteuerte Quelltext-Erzeugung
- Allgemeine Stil-Hinweise und weitere Tipps

Der Präprozessor kennt kein C(++)

Der Syntax des Präprozessors ist extrem einfach und erkennt nur:

- Kommentare sowie Zeichen- und Zeichenketten-Literale*
- Zeilenverkettung durch Abschluss einer Zeile mit Gegenschrägstrich
- Präprozessor-Direktiven in Zeilen beginnend mit einem Hash-Zeichen (White-Space direkt vor und hinter # optional möglich).
- Bezeichner, denen optional ein paar runder Klammern folgt, und innerhalb dieser Kommata (als Argumenttrenner) sowie weitere, paarige runde Klammern (welche die Wirkung enthaltener Kommata als Argumenttrenner aufheben).

[!] Makro-Bezeichner liegen außerhalb der C++-Namespaces und ihr Ersatztext wird ohne Beachtung des syntaktischen Kontexts als einfache Text-Substitution eingesetzt.

*: Dies bedingt sich gegenseitig, denn sonst könnten z.B. Zeichen-Literale und Kommentare keine Gänsefüßchen (`"`) und Zeichenketten-Literale keine Kommentarbegrenzer (`/*`, `*/` und `//`) oder einfache Apostrophe (`'`) enthalten.

Stringizing

Unter *Stringizing* wird die Möglichkeit verstanden, das Argument einer Makro-Expansion als String-Literal zu verwenden:

- Es wird dann quasi automatisch in doppelte Gänsefüßchen eingeschlossen.
- Wenn nötig, werden für einzelne Zeichen Escape-Sequenzen ersetzt.

Als Beispiel ein Makro, der einen Ausdruck textlich und mit dem berechneten Wert auf Standardausgabe schreibt:

```
#define PrintX(x) \
        std::cout << #x << ": " << (x) << '\n';
```

Die Anwendung kann (beispielsweise) wie folgt aussehen:

```
int main() {
    auto value = 42;
    PrintX(value++)
    PrintX(++value)
    PrintX(value)
    PrintX(value *= 2)
}
```

Token Pasting

Hierunter versteht man das "Aneinanderkleben" von Makro-Argumenten und fest vorgegebenen Bestandteilen, um daraus neue Bezeichner zu erzeugen:

```
#define DEFINE_ERROR_THROWER(clazz, name)\
    virtual void clazz::throw_ ## name() {throw clazz::name();}
…
DEFINE_ERROR_THROWER(MyParser, PrematureEndOfFile)
DEFINE_ERROR_THROWER(MyParser, InvalidExpression)
```

Systematische Quelltexterzeugung

"Tabelle" als Vorbereitung

Hierfür könnte z.B. eine Liste von Bezeichnern und Texten vorliegen, für die – evtl. an ganz unterschiedlichen Stellen eines Programms – unterschiedlicher Quelltext systematisch erzeugt werden soll:

```
#define FOR_ALL_ERRORS(m)\
    m(PrematureEndOfFile, "file ends prematurely")\
    m(IncompleteLine    , "line ends prematurely")\
    m(InvalidExpression , "bad expression syntax")\
    … // usw.
```

Die Grundlage dieser Technik besteht darin, den Namen eines zu expandierenden Makros einem anderen Makro als Parameter zu übergeben. Dieser Makro könnte nun mit den beiden auf der nächsten Seite gezeigten Makros aufgerufen werden:

```
// somewhere ...
FOR_ALL_ERRORS(DEF_ERROR_CLASS)
```

```
// ... somewhere else
FOR_ALL_ERRORS(DEF_THROW_HELPER)
```

Eigentliche Quelltexterzeugung

Die einzige Regel ist, dass die an `FOR_ALL_ERRORS` zu übergebenden Makros genau zwei Argumente entgegennehmen müssen:

```
#define DEF_ERROR_CLASS(name, desc)\
    class name : public std::runtime_error {\
        public: name() : std::runtime_error(desc) {}\
    };
```

Diese müssen natürlich nicht zwingend im Ersatztext verwendet werden:

```
#define DEF_THROW_HELPER(name, desc)\
    virtual void throw_ ## name() { throw name(); }
```

Eine intensive, praktische Verwendung dieser Techniken zeigt das Beispiel: Examples/Utilities/TypePrinter/tp-final.cpp

Einige weitere Präprozessor-Tipps

Generell sollten

- Makros zur systematischen Quelltext-Erzeugung einfach gehalten werden, und
- der Präprozessor nur dann für die systematische Code-Erzeugung genutzt werden, wenn es im Sinne der folgenden Ziele geschieht:
  - Signifikante Vereinfachungen
  - Höhere Fehlersicherheit
  - Verbesserte Wartbarkeit*

> Es kann durchaus sinnvoll sein, im Ersatztext von *Helfer-Makros* wiederum *Helfer-Funktionen* und/oder *Helfer-Templates* zu verwenden, insbesondere wenn deren Argumentlisten systematische Ähnlichkeiten und Wiederholungen aufweisen.

*: Z.B. weil absehbar ist, dass weitere, ähnlich gelagerte Fälle nach und nach zu ergänzen sind und/oder künftig für alle bisherigen Fälle gemeinsame Anpassungen erforderlich werden könnten.

Fortsetzungszeilen fördern die Lesbarkeit

Dies wurde in den gezeigten Beispielen schon häufig praktiziert.

Ggf. kann zusätzlich auch erwogen werden,*

- den Gegenschrägstrich für die Fortsetzungszeilen immer in ein und dieselbe Spalte zu setzen,
- und das Ende der Makro-Definition durch einen Kommentar expliziter sichtbar zu machen.

```
#define DEF_ERROR_CLASS(name, desc)                \
class name : public std::runtime_error {           \
    public: name() : std::runtime_error(desc) {}   \
};                                        // END-OF-MACRO
```

*: Probieren Sie aber besser vorab aus, ob eine solche, auf gute Lesbarkeit zielende Formatierung nicht schon beim nächsten Lauf eines "C++-Beautifiers" wieder zerstört wird ... und auch nicht durch die gutgemeinte "Einrückungshilfe" des syntaxbewussten Editors Ihres Kollegen, der damit Ihre Quelltexte ab und zu bearbeitet.

Namenskonventionen für Makros

Da Makronamen nicht mit C++ Namespaces verbunden sind, sollte jedes Projekt verbindliche Regeln aufstellen, welche Namen für Makros verwendet werden dürfen.*

Diese könnten z.B. sein:

- Erstes Zeichen Großbuchstabe,
- gefolgt von zwei oder mehr weiteren Zeichen,
  - die Großbuchstaben, Ziffern oder Tiefstrich sein dürfen,
- Ende mit `_H` aber nur bei Makros, die Include-Guards steuern.

> Die "Mindestens-drei-Zeichen"-Regel beugt Komplikationen vor, die andernfalls durch die verbreitete Praxis entstehen könnten, für Typ-Parameter von Templates `T` (oder `T1`, `T2`, `T3`, ...) und für Wert-Parameter `N` (oder `MIN`, `MAX` ...) zu verwenden.

*: Oftmals weniger bekannt ist auch, dass nicht nur der C89-Standard alle Makronamen, welche mit einem Tiefstrich beginnen, für Zwecke der Implementierung reserviert, sondern andere, evtl. in einem Projekt relevante Standards mitunter deutlich weitergehende Regeln enthalten. So reserviert z.B. POSIX alle Makros für interne Zwecke, deren Namen die mit `LC`, `E`, `SIG` oder `SIG_` beginnen, sofern ein Großbuchstabe oder eine Ziffer folgt.

Makros nur lokal definieren

Abschließend noch zwei weitere vorbeugende Maßnahmen gegen Überraschungen, welche nicht beabsichtigten Ersetzungen des Präprozessors vermeiden helfen:

- Nur über kurze Quelltext-Abschnitte hinweg nötige Makros zur systematischen Erzeugung von Quelltext sollten ggf. direkt nach ihrer letzten Verwendung mit `#undef` wieder gelöscht werden.
- Ferner kann durch einen vorgeschalteten Test mit `#ifdef` das unbeabsichtigte Überschreiben eines (anderen) Makros verhindert werden, der eine weiter ausgedehnte Sichtbarkeit hat und diese auch gezielt haben soll.

Beides zeigt das Beispiel auf der nächsten Seite, welches die oft notwendige Umwandlung von per `enum` definierten Bezeichnern in entsprechende (druckbare) Zeichenketten demonstriert.*

---

*: Dieses Beispiel verwendet noch zwei weitere, mit C++11 eingeführte Neuerungen, nämlich die neue Syntax zur *Typ-Definition mit* `using` und `enum` *Klassen*. Bei letzteren sind die einzelnen Bezeichner qualifiziert zu verwenden. Somit zeigt sich zugleich eine Stärke der Quelltext-Erzeugung durch Makros: um wieder auf die klassischen Aufzählungstypen (zurück) zu wechseln, müsste lediglich der Makro `MAP_COLOUR_TO_STRING` angepasst werden, nicht aber dessen Verwendungen.

```
enum class Colour : unsigned char { Red, Blue, Green };
…
#ifdef MAP_COLOUR_TO_STRING
#error "macro 'MAP_COLOUR_TO_STRING' already defined"
#endif
std::string colour2string(Colour c) {
    switch (c) {
       #define MAP_COLOUR_TO_STRING(c)\
        case Colour::c: return "Colour::" #c;
           // ----------------------------
              MAP_COLOUR_TO_STRING(Red)
              MAP_COLOUR_TO_STRING(Blue)
              MAP_COLOUR_TO_STRING(Green)
           // add more colours to map here
           // ----------------------------
       #undef MAP_COLOUR_TO_STRING
        default: {
            using ULL = unsigned long long;
            return "Colour::#"
                 + std::to_string(static_cast<ULL>(c))
                 + " (not mapped to a name)"
        }
    }
}
```