C++ FOR
(Wednesday Morning)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt im direkten Anschluss an die Mittagspause.

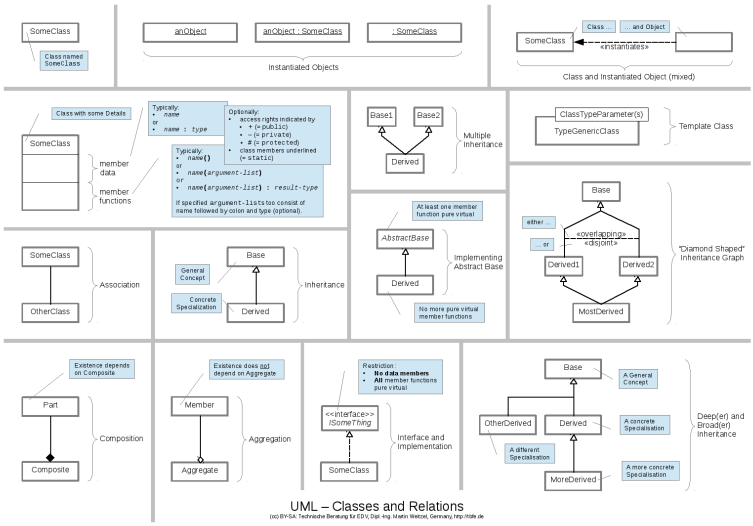
Beziehungen zwischen Klassen

- Klasse (minimal)
- Klasse (detailiert)
- Instanziiertes Objekt
- Parametrisierte Klasse

- Assoziation
- Komposition
- Aggregation

- (Mehrfach-) Vererbung
- Interface
- Abstrakte Basisklasse

- "Rautenförmige" und ...
- ... allgemein mehrstufige Vererbungs-Hierarchien



Klasse (minimal)

Die minmale Darstellung einer Klasse in der UML besteht aus einem Rechteck, in welchem der Klassenname steht.

> Klassen und Beziehungen zwischen Klassen sind die wichtigsten Bestandeteile einer *Objekt-Orientierten-Modellierung* (OOM).

Häufig wird eine solche etwa

- beginnen mit einem High-Level-Design (HLD),
- das eine Reihe von Verfeinerungsschritten durchläuft,
- ...
- und schließlich enden mit einer Lösungs-Implementierung.*

*: Auch im Kern sehr formale Prozesse wie etwa RUP beinhalten dabei die Möglichkeit eines "Tailoring", indem nur diejenigen Artefakte tatsächlich erstellt werden, von denen man sich einen konkreten Nutzen verspricht.

Objektorientierte Vorgehensweise

Die übliche Abfolge der Schritte ist

- zunächst eine Objektorientierte (Problem-) Analyse (OOA) vorzunehmen,
- der ein mehr oder weniger detailliertes Objekt-Orientiertes (Lösungs-) Design (OOD) folgt,
- welches schließlich – vorzugsweise den Mitteln der Objekt-Orientierten Programmierung (OOP) implementiert wird.

Die beiden Begriffe *OOA* und *OOD* werden mitunter auch zusammengefasst zu *OOAD* (Objektorientierte Analyse und Design), oder unter Hinzuziehung der Programmierung zu *OOADP*.

Klassen vs. Objekte

Hierfür benutzt die UML im Wesentlichen die selbe Symbolik – der Unterschied besteht lediglich darin, dass

- bei Objekten der Name unterstrichen ist und
- der Klassenname auch entfallen kann.*

Klassen

Diese kommen einem *Bauplan* gleich und beschreiben in der UML-Darstellung alle *Gemeinsamkeiten* der Objekte, die gemäß diesem Bauplan erstellt werden.

Objekte

Diese sind *instanzierte Klassen* und beinhalten in der UML-Darstellung die *Unterschiede* aus, welche trotz des gemeinsamen Bauplans bestehen.

---

*: Die tatsächlichen Regeln der grafischen Darstellung sind komplizierter und erlauben prinzipiell, dass ein dem Namen des Objekts ein Doppelpunkt und dann der Klassenname folgt, wobei beide Namen optional sind und weggelassen werden können (beispielsweise wenn sie sich aus dem Kontext ergeben).

Klassen und Objekte in C++

In kompilierten Sprachen wie C++, in denen die Unterstützung der Objektorientierten Programmierung nict darin besteht, quasi einen Interpreter für ein Objekt-System zur Laufzeit anzubieten, gibt es in der Regel einen weiteren charakteristischen Unterschied zwischen Klassen und Objekten:

- Klassen sind in C++ **statisch** in dem Sinne, dass sie vollständig zur Compilezeit beschrieben werden, während
- Objekte sich typischerweise zur Laufzeit verändern.

C++-Klassen als Compilezeit-Konstrukt

Die vollständige Festlegung einer Klasse zur Compilezeit ist ein wesentlicher Unterschied zu deutlich dynamischeren OOP-Sprachen, wie etwa

- Smalltalk
- Python oder
- Itcl

deren Klassen und Objekte – auch und gerade hinsichtlich ihrer Member-Daten und -Funktionen zur Laufzeit völlig dynamisch erzeugt werden können, und somit z.B. auch datenabhängig erstellt werden können.*

*: Java und C Sharp sind in dieser Hinsicht wiederum eher ähnlich zu C++, besitzen allerdings mehr standardisierte Möglichkeiten zur Introspektion und können mit Hilfe kleiner Kunstgriffe in großen Teilen den Eindruck eines zur Laufzeit dynamischen Verhaltens bieten.

Klasse (detailiert)

In stärkerer Detaillierung beschrieben kann einer Klasse im UML-Diagramm folgendes hinzugefügt werden:

- Ein Abschnitt mit Attributen
- Ein Abschnitt mit Methoden

Beides wird innerhalb des Klassensymbols durch eine waagrechte Linie getrennt und beides ist optional.

> Pragmatiker weisen gerne darauf hin, dass die UML **keineswegs** die Nutzung aller ihrer notationellen Möglichkeiten vorschreibt sondern der Detaillierungsgrad eines Klassendiagramm stets an dessen Zweck bzw. dem Zielpublikum entsprechend ausgerichtet sein sollte.

Attribute

Diese enthalten *Datenwerte* und sind im UML-Diagramm daran zu erkennen, dass ihnen **keine** runden Klammern folgen. Ein nachgestellter, durch einen Doppelpunkt abgetrennter Datentyp ist dabei optional.

Methoden

Diese enthalten *ausführbare Abläufe* und sind im UML-Diagramm daran zu erkennen, dass ihnen **runde Klammern folgen**, ggf. auch leere.

> (i) In C++ ist hierfür eher die Bezeichnung *Member-Funktionen* üblich.

Innerhalb der Klammern können optional zu übergebende Parameter benannt werden, inklusive deren Typ (nachgestellt und durch Doppelpunkt getrennt), und der geklammerten Parameterliste kann (ebenfalls durch einen Doppelpunkt getrennt) ein Rückgabetyp folgen.

Zugriffsschutz

Ein weiteres, in der detaillierten Darstellung von UML-Klassen mögliches Notationselement bezieht sich auf den Zugriffsschutz und kann sowohl Member-Daten wie auch Member-Funktionen optional vorangestellt werden:

- `+` (öffentlich) – `public` in C++
    - jeder, der Zugriff zu einem Objekt dieser Klasse hat, kann auf diese Member-Daten zugreifen bzw. diese Member-Funktionen ausführen.
- `#` (geschützt) – `protected` in C++
    - erreichbar nur für die Member-Funktionen der eigenen und davon abgeleiteter Klassen.
- `-` (privat) – `private:` in C++
    - erreichbar nur für die Member-Funktionen der eigenen Klasse.

C++ Zugriffschutz: `class` vs. `struct`

Eine Besonderheit gilt in C++ für den Default des Zugriffschutzes:

- In einer `struct` ist dieser `public`,
- in einer `class` dagegen `private`.

Abgesehen von diesem Unterschied sind die Schlüsselworte `struct` und `class` bei der Definition einer Klasse austauschbar.

```
class MyClass {
// starts private
    …
public:
    …
};

struct MyStruct {
// starts public
    …
private:
    …
};
```

```
struct MyClass {
private:
    …
public:
    …
};

class MyStruct {
public:
    …
private:
    …
};
```

Klassen-Attribute und -Methoden

In der detaillierten Darstellung einer UML-Klasse unterstrichene Bezeichner stellen sog. Klassen-Attribute oder -Methoden dar und entsprechen in der C++-Programmierung den Zusatz `static`.

Klassen-Attribute

Diese sind **nicht** pro Objekt sondern pro Klasse gespeichert.*

Klassen-Methoden

Diese können ausschließlich auf Klassenattribute zugreifen. In C++

- bieten sie zum einen eine kleine Effinzienzverbesserung, da bei ihrem Aufruf keine Objektadresse übergeben werden muss, und
- sind zum anderen auch ohne ein vorhandenes Objekt durch Voranstellen des Klassenamens gefolgt vom Scope-Operator (`::`) aufrufbar.

---

*: Eine typische Anwendung wäre z.B. eine fortlaufende Nummerierung aller jemals erzeugten Objekte einer Klasse, wobei ein Klassenattribut die nächste zu vergebende Nummer enthält.

Parametrisierte Klasse

Hierbei handelt es sich um die UML-Sichtweise auf das, was in C++ mit dem Mechanismus der Templates verfügbar ist und auch die Bezeichnung *Generische Programmierung* (Generic Programming) trägt.

Ursprünglich entsprang der Template-Mechanismus in C++ dem Wunsch, Datentypen und Compilezeit-Konstanten – etwa Typ und Anzahl der Elemente in einem Array – parametrisieren zu können, wenn dieses Member einer Klasse ist.

Assoziation

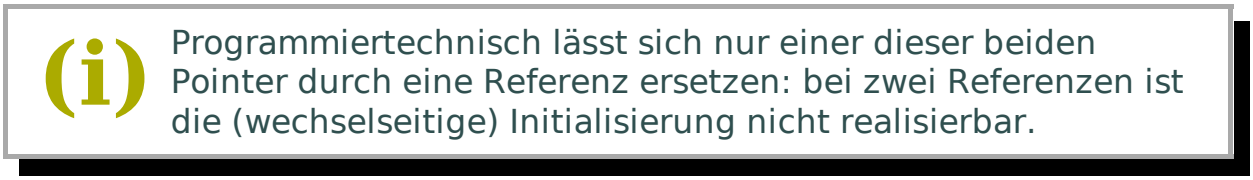
Eine Assoziation ist die unspezifischste aller Beziehungen, die zwischen zwei Klassen bestehen können, und besagt mehr oder weniger nur, dass diese beiden Klassen eine bestimmte Aufgabe **gemeinsam** erledigen.

Die tatsächliche Beziehung kann stärker sein und auch eine der anderen Beziehungsformen annehmen, insbesondere die der

- Aggregation oder
- Komposition.

Ungerichtete Assoziation in C++

Streng genommen erfordert die Assoziation in C++, dass zwei Klassen gegenseitig aufeinander verweisen, was z.B. mittels zweier Pointer.*



Beispiel-Code zur ungerichteten Assoziation

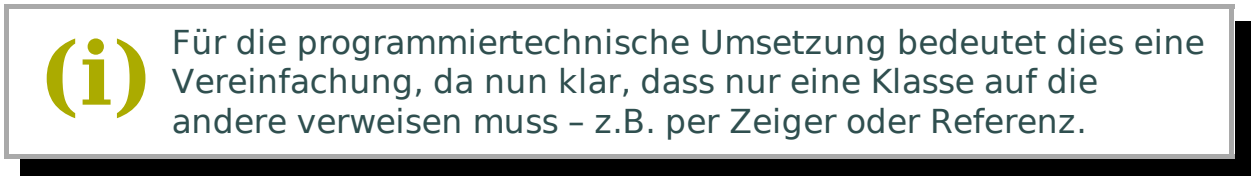
```
class Parent {
    class Child *c;
    …
    Parent() : c(nullptr) {}
    void setChild(Child *c_) {
        c = c_;
    }
};
…
Parent first;
```

```
class Child {
    class Parent &p;
    …
    Child(Parent &p_)
        : p(p_) {
        p.setChild(this);
    }
};
…
Child second(first);
```

*: Siehe: Examples/ClassRelations/general_association.cpp

Gerichtete Assoziation

Durch Hinzufügen eines Pfeils an der Assoziationslinie wird ggf. die *eingeschränkte Navigierbarkeit* dargestellt.*



Beispiel-Code zur gerichteten Assoziation

```
class Parent {
    // CANNOT call its child
    // directly (but the child
    // could hand over this on
    // a call to the parent)
    …
};
…
Parent first;
```

```
class Child {
    class Parent &p;
    …
    Child(Parent &p_)
        : p(p_)
    {}
};
…
Child second(first);
```

*: Siehe: Examples/ClassRelations/directed_association.cpp

Komposition

Komposition wird durch eine ausgefüllte Raute zum Ausdruck gebracht, welche an einem Ende der Assoziationslinie hinzugefügt wird.*

Beispiel-Code zur Komposition

```
class Car {
    class Motor engine;
    …
    Car( … )
        : engine( … )
    {}
};
…
Car someCar; // also creates
             // the engine
```

```
class Motor {
    // will usually be created
    // as part of a car and
    // also scrapped with it
    …
};
```

*: Siehe: Examples/ClassRelations/composition.cpp

Aggregation

Die Aggregation ist eine nur geringfügig stärkere Form der Assoziation, indem sie

- die Klasse auf einer Seite zur *Gesamtheit* (Aggregat)
- die Klasse auf anderen zu *deren Teil* erklärt.

Gesamtheit und Teil können dabei unabhängig voneinander existieren.

In der UML-Darstellung wird an die (Assoziations-) Linie zwischen den beiden beteiligten Klassen auf der Seite der Gesamtheit eine *nicht-ausgefüllte* Raute hinzugefügt.*

*: Aus pragmatischer Sicht ist darauf hinzuweisen, dass Aggregation und Assoziation häufig eng beieinander liegen und mitunter auch "Ansichtssache" sind, womit ein längeres Nachdenken oder gar eine kontroverse Diskussion darüber nicht lohnt, welche Art von Beziehung per UML darzustellen ist.

Beispiel-Code zur Aggregation

```
class Car {
    class Motor *e;
    …
    Car(Motor *e_ = nullptr)
        : e(e_)
    {}
    void setEngine(Motor *e_) {
        assert(e && !e_
           || !e && e_);
        e = e_;
    }
};
```

```
class Motor {
    …
};
…
Motor firstMotors;
Motor otherMotor;
Motor unusedMotor;
```

Prefering pointers over references looks like an appropriate decision if a car, during its lifetime, may get a replacement engine.*

Some Cars (with and without Motor):

```
Car soldCar(&firstMotor);        // a car with and ...
Car usedForCrashTest;            // ... without an engine
…
soldCar.setEngine(nullptr);      // remove old engine ...
soldCar.setEngine(otherMotor);   // ... for replacement
```

*: Siehe: Examples/ClassRelations/aggregation.cpp

Vererbung

Hierbei geht es um die Beziehung zwischen einem

- allgemeinen Konzept und
- dessen Spezialisierung.

In der UML-Darstellung wird an die (Assoziations-) Linie zwischen den beiden beteiligten Klassen auf der Seite der Basisklasse ein nicht ausgefülltes Dreieck hinzugefügt.

Speicher-Layout und Lebenszeit-Kopplung entsprechen zwar der Komposition, als wichtige Besonderheit gilt jedoch das LSP.

(i) Das von Barbara Liskov formulierte Ersetzungs-Prinzip verlangt, dass ein Objekt einer abgeleiteten Klasse (Spezialisierung) stets ein geeigneter Stellvertreter für ein Objekt seiner Basisklasse sein sollte.*

*: Any conventional compiler which only checks the existance of member functions and their correct call with respect to argument and return types can guarantee the LSP only to a very limited degree!

Mehrfachvererbung

Diese liegt vor, wenn eine Klasse zwei Basisklassen hat, also zwei allgemeine Konzepte spezialisiert.

In der Geschichte der Objektorientierung gab es zahlreiche Diskussionen über Sinn und Zweck der Mehrfachvererbung und eine Reihe bekannter OOP-Sprachen – etwa Java – verzichtet auf Mehrfachvererbung.*

*: Java unterstützt allerdings Klassen, die mehrere Interfaces implementieren.

Interface (Schnittstelle)

Bei einem Interface geht es prinzipiell um eine Art Vertrag zwischen zwei Klassen:

- Es gibt einerseits eine (oder mehrere) Klasse(n), die ein bestimmtes Interface implementieren und
- für (in der Regel mehrere) andere Klassen ist damit klar, wie deren Angebot an Member-Funktionen aussieht.

In der UML-Notation steht über dem Namen eines Interfaces das Stereotype `«interface»`.

Anwendung von Interfaces

Generell reduzieren Interfaces die Kopplung, d.h. den Grad zu dem verschiedene Klassen Details voneinander kennen (müssen).

Zudem bieten sie Flexibilität zur Laufzeit, indem aus mehreren verfügbaren Implemetierungen u.U. die für eine Situation am passende gewählt werden kann.

> Interfaces haben vor allem durch *Objektorientierte Entwurfsmuster (OO-DP)* eine gewisse Bekanntheit erlangt und spielen auch im *GoF-Buch* eine wichtige Rolle.

Interfaces vs. Klassen

Technisch gesehen sind Interfaces in C++ einfach in bestimmter Form Weise eingeschränkte Klassen:

- Sie besitzen keine Member-Daten und
- ausschließlich rein virtuelle Member-Funktionen.

Ein spezielles Schlüsselwort, das diese Einschränkungen **erzwingt**, gibt es in C++ nicht.

Abstrakte Basisklasse

Die Bezeichnung *abstrakt* wird für Klassen verwendet, die mindestens eine **rein virtuelle** Member-Funktion haben.

In der UML-Darstellung wird der Name einer abstrakten Klasse kursiv geschrieben.

Abstrakte Klassen können nur als Basisklassen verwendet werden.

Davon abgeleitete Klassen werden dann

- die rein virtuelle(n) Member-Funktion(en) implementieren,
- oder sind selbst (weiterhin) abstrakte Basisklassen.

[!] Der Versuch, Objekte einer abstrakten Klasse zu erzeugen scheitert in C++ mit einem Kompilierfehler.

"Rautenförmige" Vererbungsstruktur

Eine solche entsteht, wenn sich bei Mehrfachvererbung auf der mittleren Ebene Klassen befinden, die wiederum eine gemeinsame Basisklasse besitzen.

Es ist bezüglich der gemeinsamen Basisklasse dann in der UML-Darstellung eine Unterscheidung zwischen zwei Fällen erforderlich, welche die Daten-Member der gemeinsamen Basisklasse betreffen:

- `«overlapping»` – die Daten-Member sind nur einmal vorhanden;
- `«disjoint»` – die Daten-Member sind doppelt vorhanden.

[!] Im zweiten Fall ist in Bezug auf die ganz oben stehende Basisklasse von der ganz unten stehenden Klasse aus gesehen das LSP außer Kraft gesetzt, da es nicht mehr eindeutig ist.

Mehrstufige Vererbung allgemein gesehen

Im Allgemeinen gehen Vererbungs-Hierarchien

- sowohl *in die Breite*, d.h. eine Basisklasse wird oft mehrere direkt abgeleitete Klassen haben,
- wie auch *in die Tiefe*, d.h. es gibt mehrere Stufen – sieht man die direkt abgeleiteten Klassen als "Kinder", gibt es somit auch "Enkel", "Ur-Enkel" usw.

Rautenförmige Hierarchien mit gemeinsamen `«disjoint»` Basisklassen ausgenommen, gilt das LSP über alle Stufen einer in die Tiefe gehenden Vererbungshierarchie.

Mehrfachvererbung und virtuelle Basisklassen

- Prinzip der Mehrfachvererbung
- Virtuelle Basisklassen

- Überlappende Basisklassen

Prinzip der Mehrfachvererbung

Bei der Mehrfachvererbung kann im Speicherlayout nur noch eine der Basisklassen eine gemeinsame Anfangsadresse mit der abgeleiteten Klasse haben.

Dies stellt keine große technische Herausforderung dar sondern bedeutet lediglich eine minimale Komplikation bei der Umsetzung des LSP, wo im Fall der Weiterreichung der abgeleiteten Klasse ggf. ein Offset zum `this`-Zeiger addiert werden muss.

Virtuelle Basisklassen

Virtuelle Basisklassen bilden in C++ den Hintergrund der Lösung dessen, was die UML bei rautenförmigen Vererbungshierarchien im Fall von «overlapping»-Klassen an der Spitze verlangt.

Eine mögliche Implementierung von Mehrfachvererbung bürdet den Overhead den Klassen auf der mittleren Ebene auf, indem diese nicht nur den Datenteil ihrer (virtuellen) Basisklasse enthalten, sondern einen zusätzlichen Zeiger, über den **alle Bezugnahmen auf den Basisklassenteil** erfolgen.

Überlappende Basisklassen

Im konkreten Fall einer rautenförmigen Hierarchie werden

- die Daten-Member der Basisklasse nur ein einziges Mal in der *Most Derived Class* vorhanden sein, und
- die Zeiger in den Klassen der mittleren Ebene werden jeweils auf dieses Daten-Member verweisen.

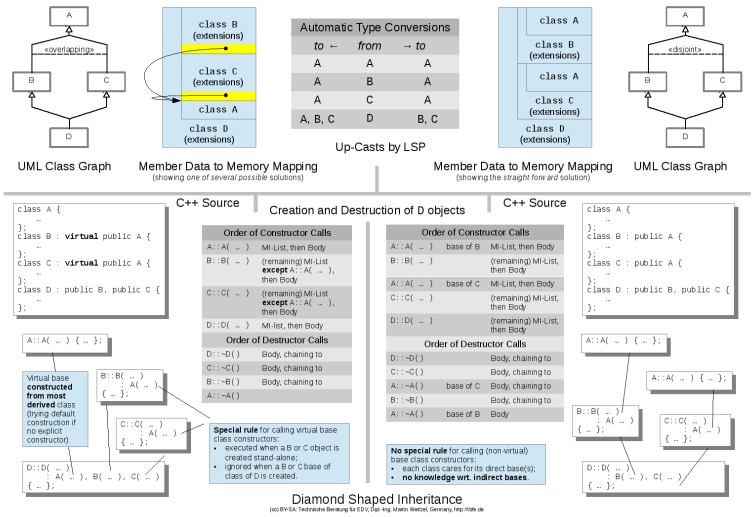
Da alle Bezugnahmen darauf – wie gerade beschrieben – über diese Zeiger laufen, sprechen die Klassen der mittleren Ebene **ein und dasselbe** Basisklassen-Objekt an.

Rautenförmige Ableitungshierarchien

- UML-Darstellung «overlapping» ...
- ... versus «disjoint»

- Abbildung auf Speicher «overlapping» ...
- ... versus «disjoint»

- Konstruktor / Destruktor «overlapping» ...
- ... versus «disjoint»



UML-Darstellung «overlapping»

Die zusammengefasste Darstellung des Falls `«overlapping»` in Bezug auf

- die UML-Darstellung,

erlaubt den direkten Vergleich zum Fall `«disjoint»`.

Click here for Info-Graphic
DiamondShapedInheritance
UML Class Graph («disjoint»)

UML-Darstellung «disjoint»

Die zusammengefasste Darstellung des Falls «disjoint» in Bezug auf

- die UML-Darstellung,

erlaubt den direkten Vergleich zum Fall «overlapping».

Abbildung auf Speicher «overlapping»

Die zusammengefasste Darstellung des Falls «overlapping» in Bezug auf

- die Speicherabbildung

erlaubt den drekten Vergleich zum Fall «disjoint».

Abbildung auf Speicher «disjoint»

Die zusammengefasste Darstellung des Falls «disjoint» in Bezug auf

- die Speicherabbildung

erlaubt den direkten Vergleich zum Fall «overlapping».

Quelltext und C'tor-/D'tor-Ablauf bei «overlapping»

Die zusammengefasste Darstellung des Falls «overlapping» in Bezug auf

- den Quelltext,
- den Konstruktor- sowie
- den Destruktor-Ablauf

erlaubt den drekten Vergleich zum Fall «disjoint».

Quelltext und C'tor-/D'tor-Ablauf bei «disjoint»

Die zusammengefasste Darstellung des Falls «disjoint» in Bezug auf

- den Quelltext,
- den Konstruktor- sowie
- den Destruktor-Ablauf

erlaubt den direkten Vergleich zum Fall «overlapping».

Beispiele zu Klassenbeziehungen

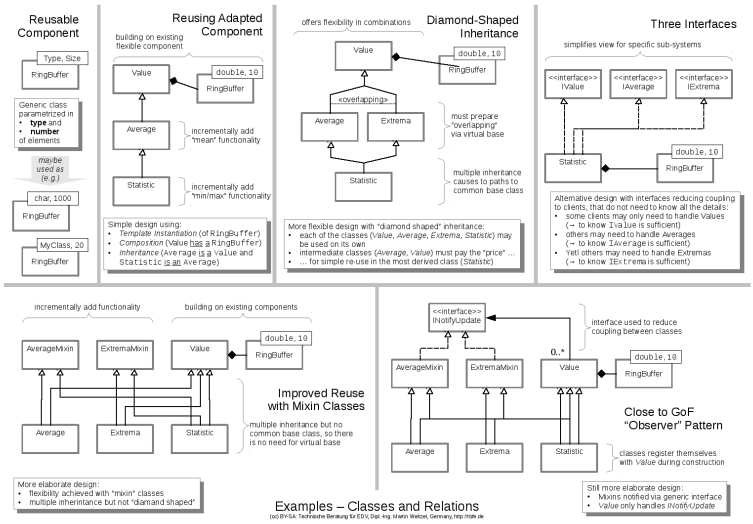
- Eine anpassbare Komponente ...
- ... und deren Nutzung

- Rautenförmige (Mehrfach-) Vererbung

- Drei Interfaces

- Flexible Erweiterbarkeit durch "Mix-Ins"

- Orientiert am GoF *Observer Muster*



Eine (anpassbare) Komponente

Anpassbare Komponenten entstehen normalerweise entweder

- durch Planung, wenn zukünftige Variabilität (korrekt) vorhergesehen wurde, oder
- durch Erfahrung, wenn zu einer bereits bestehende Funktionalität eine – im ersten Entwurf noch nicht eingeplante – neue Variante benötigt wird.

In den konkreten Beispielen* zu diesem Abschnitt ist die Template-Version der `RingBuffer`-Klasse eine solche anpassbare Komponente.

*: Sieha auch Aufgabe Mittwochvormittag (Teil 2) und Examples/ClassDesigns/ReusableComponent

Nutzung der (angepassten) Komponente

Die Nutzung einer (angepassten) Komponente reduziert den Aufwand an immer wieder neu zu schreibendem Code.

In den konkreten Beispielen* zu diesem Abschnitt wird

- die Template-Version der `RingBuffer`-Klasse ggf.
- mit unterschiedlichen Instanziierungs-Argumenten

verwendet.

Die Alternative dazu wäre *Copy&Paste-Programmierung*, also ein klarer Verstoß gegen das *DRY-Pinciple*.

*: Examples/ClassDesigns/AdaptedComponent

Click here for Info-Graphic
Example-ClassDesign
Diamond-Shapend Inheritance

Rautenförmige (Mehrfach-) Vererbung

Besteht die Bereitschaft den Preis für die dann oft notwendigen, virtuellen Basisklassen zu zahlen, lassen sich flexible Kombinationsmöglichkeiten mit einem guten Grad an Wiederverwendung durch Mehrfachverarbung erreichen.

In den konkreten Beispielen* zu diesem Abschnitt werden aus den Klassen `Average` und `Extrema` mittels Mehrfachvererbung zur Klasse `Statistic` kombiniert, womit diese beide Fähigkeiten vereint:

- den Mittelwert aller Daten zu ermitteln, und
- das Minimum und Maximum aller Daten zu ermitteln.

*: Examples/ClassDesigns/DiamondShaped

Drei Interfaces

Die Nutzung von Interfaces kann die Komplexität der Gesamt-Architektur aus der "Sicht einzelner Klienten" etwas vereinfachen.

In den konkreten Beispielen* zu diesem Abschnitt werden

- zu einer gemeinsamen Klasse `Statistic`
- drei Interfaces definiert,

so dass

- hinsichtlich der *Nutzer* dieser Klasse jeweils klar ist,
- auf welches Sub-Set der Gesamt-Funktionalität sie angewiesen sind.

*: Examples/ClassDesigns/ThreeInterfaces

Inkrementale Erweiterbarkeit durch Mix-Ins

Über Mixin-Klassen lässt sich ggf. flexible Kombinierbarkeit schaffen, auch ohne dass virtuelle Basisklassen notwendig werden.

In den konkreten Beispielen* zu diesem Abschnitt werden

- die Klassen `Average`, `Extrema` und `Statistic` flexibel zusammengesetzt durch
- unterschiedliche Kombinationen der Basisklasse `Value`
- mit den beiden Mixins `AverageMixin` und `ExtremaMixin`.

*: Leicht vereinfacht (ohne `RingBuffer`-Komponente) in Aufgabe von Mittwochvormittag (Teil 3, Variante 1)

Orientiert am GoF *Observer Muster*

Das "Flaggschiff" unter allen hier verglichenen Entwürfen.

Es vereint hohe Flexibilität und geringe Kopplung ... hat allerdings auch die komplexeste Implementierung.

In den konkreten Beispielen* zu diesem Abschnitt werden in Anlehnung an das Observer Pattern

- die Klassen `Average`, `Extrema` und `Statistic` flexibel zusammengesetzt durch
- unterschiedliche Kombinationen der Basisklasse `Value`
- mit den beiden Mixins `AverageMixin` und `ExtremaMixin`,

wobei Abhänggkeiten gering gehalten werden

- über das `INotifyUpdate`-Interface und
- ein flexibles Registrierungsprotokoll.

*: Leicht vereinfacht (ohne `RingBuffer`-Komponente) in Aufgabe von Mittwochvormittag (Teil 3, Variante 2)

Praktikum