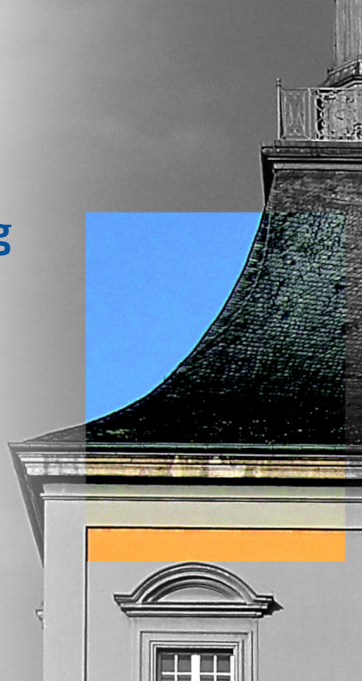
UNIVERSITÄT BONN

Algorithmen und Programmierung

Objektorientierte Programmierung

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



Gitversion: 'c0a60ee8268b408f4d4ed2f51908c808b1ea581f'

KURZE WIEDERHOLUNG

Offne Frage

Welche objektorientierten Modellierungsansätze verwendet man bei Softwareprojekten mittlerer Größe?

UNIVERSITÄT BONN

Übersicht

- ✓ Abstrakte Klassen
- SOLID und insbesondere das Dependency Inversion Principle
- Objekt- und Typbeziehungen
- Ausblick

UNIVERSITÄT BONN

Abstrakte Klassen und Interfaces

Wiederholung: Eine rein virtuelle Memberfunktion besitzt keine Implementierung.

Eine **abstrakte Klasse** ist eine Klasse, die über mindestens eine rein virtuelle Memberfunktion verfügt. Abstrakte Klassen können nicht instanziiert werden.

Ein **Interface** ist eine abstrakte Klasse, die ausschließlich über rein virtuelle Memberfunktionen[1] und keine Membervariablen verfügt.

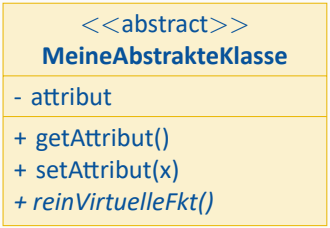
Also sind Interfaces abstrakter als gewöhliche abstrakte Klassen und diese sind abstrakter als gewöhnliche Klassen.

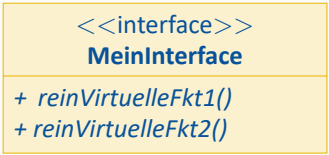
[1] Mit der Ausnahme, dass der Konstruktor nie virtuell ist.

UNIVERSITÄT BONN

Abstrakte Klassen und Interfaces in UML

In UML werden rein virtuelle Funktionen durch kursive Namen oder den Zusatz {abstract} gekennzeichnet. Zur besseren Lesbarkeit darf der Namen einer abstrakte Klassen oder eines Interfaces durch die Beschreibung «abstract» bzw. «interface» ergänzt werden.





Entwurfsprinzipien

Abstrakte Klassen vs Templates

Ziel

Vielen Programmieranfänger:innen fällt es schwer, sich zwischen der Verwendung von abstrakten Klassen und Templates zu entscheiden

Wir diskutieren diesen Umstand

UNIVERSITÄT BONN

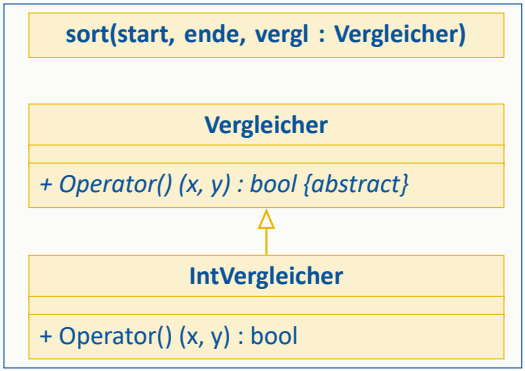
Überblick

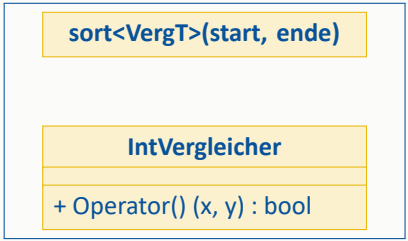
Um sich in jedem Fall zeitarm und sinnvoll zwischen der Verwendung von abstrakten Klassen und Templates zu entscheiden, sind die Inhalte der nächste Vorlesung, viel selbstständige Übung und ausgiebige Diskussionen mit erfahrenen Programmier:innen nötig.

UNIVERSITÄT BONN

Typisches Beispiel

Wie soll eine allgemeine Sortierfunktion modelliert werden? Soll der Vergleicher als abstrakte Klasse modelliert werden und Konkretisierungen an die Sortierfunktion übergeben werden? Oder soll pro Vergleicher eine neue Sortierfunktion aus einem Funktionstemplate erzeugt werden?





UNIVERSITÄT BONN

Faustregel für Programmieranfänger:innen

Falls entschieden werden muss, ob abstrakte Klassen oder Templates verwendet werden sollen und keine weiteren, sehr gute, einschränkende Gründe genannt werden, sollte man sich wie folgt für die Verwendung von abstrakte Klassen bzw. Templates entscheiden.

- Falls eine Ist-Ein-Beziehung ausgedrückt und verwendet werden soll, ist es nachvollziehbarer abstrakte Klassen zu verwenden.
- Falls eine Ist-Ein-Beziehung unwesentlich ist oder nicht ausgedrückt werden soll, ist es nachvollziehbar und effizienter Templates zu verwenden.

In oben gezeigten Beispiel ist die Ist-Ein-Beziehung der Vergleicher unwesentlich. Deshalb werden hier typischerweise Templates verwendet.

Zusammenfassung

Vielen Programmieranfänger:innen fällt es schwer, sich zwischen der Verwendung von abstrakten Klassen und Templates zu entscheiden

Wir haben eine erste Faustregel als Entscheidungshilfe diskutiert

Haben Sie Fragen?

Entwurfsprinzipien

Die SOLID Entwurfsprinzipien

Offene Frage

Welche Prinzipien helfen uns Projekte zu modellieren, die nachvollziehbar strukturiert, wartbar und flexibel sind?

UNIVERSITÄT BONN

Motivation

Bei lang laufenden Projekte, kommen stets neue oder speziellere Komponenten oder Typen hinzu. Dadurch muss die objektorientierte Modellierung und Projektdokumentation wiederkehrend angepasst werden.

Bei sich stetig verändernden Modellierungen ist es offenbar sehr wünschenswert, dass die angebotenen Schnittstellen nicht verändern werden. Das führt zu **Abwärtskompatibilität**. Gleichzeitig ist es sehr wünschenswert, dass möglichst allgemeine Typen in eine bestehende Objektinteraktion integriert werden können.

UNIVERSITÄT BONN

SOLID

SOLID sind fünf miteinander verträgliche Entwurfsprinzipien, um Softwareprojekte möglichst nachvollziehbar, wartbar und flexibl zu entwerfen. Vereinfacht gesagt besagen die Entwurfsprinzipien das Folgende.

- **Single-Responsibility**: Klassen sollen für genau eine Aufgabe verantwortlich sein.
- **Open-Closed**: Klassen sollen erweiterbar, aber nicht veränderbar sein.
- **Liskov-Substitution** haben wir bereits besprochen.
- **Interface-Segregation**: Klassen sollen nicht von Schnittstellen abhängen, die sie nicht verwenden.
- **Dependency-Inversion**: Klassen und Implemtierungsdetails sollen vom Abstrakten abhängig sein.

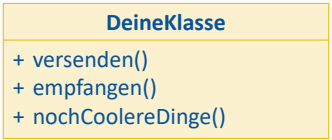
SOLID in objektorientierten Projekten

Die **SOLID**-Entwurfsprinzipien sind in dieser Vorlesung für Klassen formuliert. Sie gelten allgemeiner für objektorientierte Projekte (bei denen allgemeine Objekte auch Dienste bezeichnen können). Hierbei reicht es praktisch aus, das Wort *Klasse* durch *Typ*, *Modul*, *Bibliothek* oder *Softwarebaustein* zu ersetzen.

UNIVERSITÄT BONN

Beispiel I

Angenommen Sie haben eine Klasse modelliert, die ein wichtiges Problem löst. Sie wollen Instanzen dieser Klasse über das Netzwerk kommunizieren. Eine naive Modellierung wäre die Folgende.



Das Single-Responsibility-Prinzip besagt, dass eine Klasse für genau eine Aufgabe verantwortlich sein soll. Die Lösung Ihres Problems und die Kommunikation der Instanzen sind zwei verschiedene Verantwortlichkeiten. Also ist das Single-Responsibility-Prinzip verletzt.

UNIVERSITÄT BONN

Beispiel II

Um das Single-Responsibility-Prinzip zu respektieren, entwerfen wir eine weitere Klasse für die Netzwerkkommunikation.

Kommunikator
+ versenden(x : DeineKlasse) + empfangen(x : DeineKlasse)

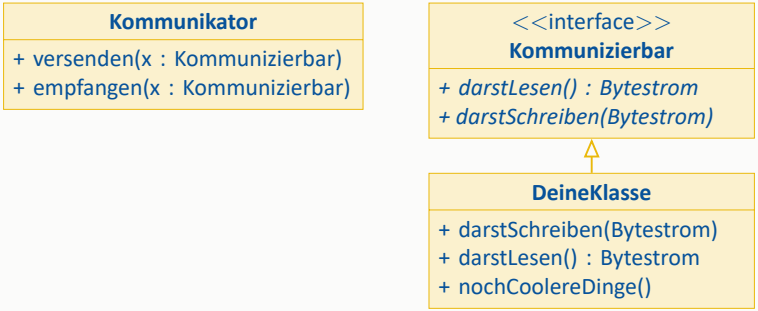
DeineKlasse
+ darstSchreiben(Bytestrom) + darstLesen() : Bytestrom + nochCoolereDinge()

Das Dependency-Inversion-Prinzip besagt, dass Klassen von Abstraktion abhängen soll und nicht von konkreten Typen oder Implemtierungsdetails. Es ist nicht ohne weiteres Möglich, denn Kommunikator für weitere, konkrete Klassen zu verwenden. Also ist das Dependency-Inversion-Prinzip verletzt.

UNIVERSITÄT BONN

Beispiel III

Um das Dependency-Inversion-Prinzip zu respektieren, entwerfen wir ein Interface für kommunizierbare Klassen.

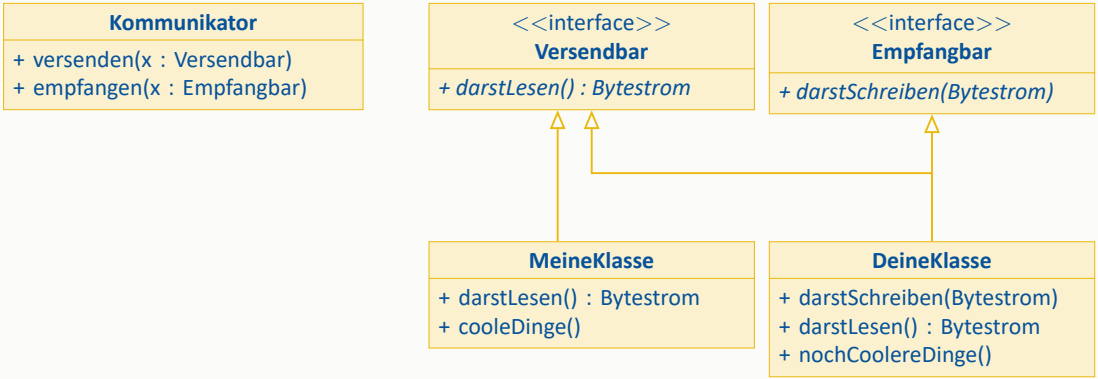


Das Interface-Segregation-Prinzip besagt, dass Klassen nicht von Schnittstellen abhängen sollen, die sie nicht benötigen. Das Interface-Segregation-Prinzip ist verletzt für Klassen, die nur versandt werden sollen, aber nicht empfangen werden müssen.

UNIVERSITÄT BONN

Beispiel IV

Um das Interface-Segregation-Prinzip zu respektieren, teilen wir das Interface auf.



Zusammenfassung

SOLID beschreibt fünf miteinander verträgliche Entwurfsprinzipien, um Softwareprojekte möglichst nachvollziehbar, wartbar und flexibl zu entwerfen

Haben Sie Fragen?

Entwurfsprinzipien

Das Dependency Inversion Principle

Offene Frage

Was besagt und bedeutet das Dependency Inversion Principle genauer?

UNIVERSITÄT BONN

Dependency Inversion Principle
Schichten

Das **Dependency Inversion Principle** ist ein Entwurfsprinzip, um größere Projekten langfristig erweiterbar und wartbar zu modellieren.

Größere, gut organisierte Projekte sind meistens in Schichten organisiert. Die höheren Schichten stellen allgemeine Funktionalitäten zur Verfügung und die tieferen Schichten sind bei der Implementierung der höhren Schichten beteiligt. In erster Annäherung dürfte man behaupten, dass die Klassen der niedrigeren Schichten nur dazu da sind, um die Klassen der höheren Schicht zu realisieren.

Beispiel

In der **C++**- oder **Java**-Standardbibliothek gibt es viele Container. Diese sind zur direkten Verwendung gedacht und liegen somit in einer höheren Schicht. Besagte Container werden von Datenstrukturen realisiert, zum Beispiel Listenknoten, AVL-Bäumen oder Rot-Schwarz-Bäumen. Diese Datenstrukturen sind Implemtierungsdetails. Also liegen Sie in einer niedrigeren Schicht.

UNIVERSITÄT BONN

Dependency Inversion Principle

Das Dependency Inversion Principle setzt bei Modellierung der Schichten ein, um die Abhängigkeiten zwischen den Schichten möglichst klein zu halten. Das Ziel ist es, Abhängigkeiten zwischen zwei Schichten auf eine gemeinsame, abstrakte Schnittstelle zu reduzieren.

- **DIP-1** Klassen höherer Ebenen sollten nicht von Klassen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen.
- **DIP-2** Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

Die besagte Abstraktion wird meist durch Interfaces oder generische Programmierung realisiert.

EVALUATION

https://limesurvey.informatik.uni-bonn.de/index.php/174862?lang=de

UNIVERSITÄT BONN

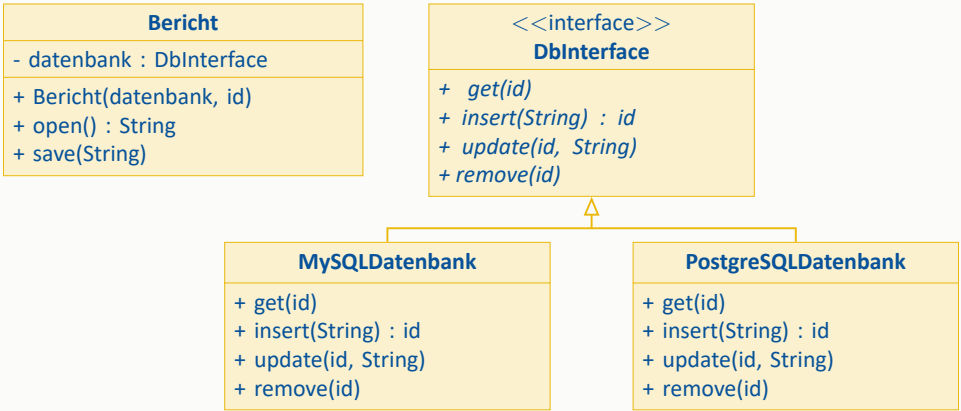
Beispiel (ohne DIP)

```
class MySQLDatenbank {
public:
  std::string get (const index_t id);

  index_t insert (const std::string& s);

  void update (const index_t id,
               const std::string& s);

  void remove (const index_t id);

private:
  /* ... */
};

class Bericht {
public:
  Bericht(MySQLDatenbank& db,
          const index_t id) :
    db(db), id(id) {}
  std::string open() {
    is_open = true;
    return db.get(id);
  }

  int save(const std::string& s) {
    db.update(id, s);
  }

private:
  MySQLDatenbank db;
  index_t id;
};


int main() {
  MySQLDatenbank db;
  Bericht ber(db, 7);
  ber.save("Heut war N1c3R Tag.");
}
```

UNIVERSITÄT BONN

Beispiel (mit DIP)

Das Beispiel verletzt das Dependency Inversion Principle, denn der höher liegende `Bericht` hängt von der tiefer liegenden `MySQLDatenbank` ab. Wir nutzen ein Interface um das Dependency Inversion Principle zu erfüllen.



UNIVERSITÄT BONN

Diskussion des Beispiels

Die erste Implementierung benötigt weniger Code, da der `Bericht` direkt von der `MySQLDatenbank` abhängt. Diese Abhängigkeit ist konkret. Durch diese Abhängigkeit ist es nötig die Implementierung des `Berichts` zu verändern, um eine verwandte Datenbank zu verwenden.

Die zweite Implementierung benötigt mehr Code, da der `Bericht` von dem Interface `DbInterface` abhängt und `MySQLDatenbank` von diesem Interface erbt. Diese Abhängigkeit ist abstrakt. Durch abstrakte Abhängigkeit ist es möglich, eine verwandte Datenbank zu verwenden ohne die Implementierung des `Berichts` zu verändern.

Zusammenfassung

Das Dependency Inversion Principle besagt, (1) dass Klassen auf verschiedenen Ebenen von abstrakten Schnittstellen abhängen sollen und (2) dass Abstraktionen nicht von Details abhängen sollen sondern umgekehrt

Programmcode der das Dependency Inversion Principle respektiert ist wartbarer und kann flexibel erweitert werden ohne dabei bereits bestehende Schnittstellen zu verändern

Haben Sie Fragen?

Entwurfsprinzipien

UNIVERSITÄT BONN

Disclaimer

Die SOLID-Entwurfsprinzipien **unterstützen** die Modellierung von nachvollziehbaren, wartbaren und flexibel erweiterbaren Projekten. Wegen auftretender Zielkonflikte ist es nicht immer möglich jedem Prinzip vollständig gerecht zu werden.

Das Modellieren muss selbstständig einstudiert werden. Das Arbeiten an Projekten mit Modellierungsschwächen (die man selbst oder andere zu verantworten haben) verdeutlicht, dass weitsichtige Modellierungen viel Zeit und Ärger spart.

Objekt- und Typbeziehungen

Ziel

Wir lernen, wie eine Auswahl von Beziehungen zwischen Objekten und Typen kommuniziert werden

UNIVERSITÄT BONN

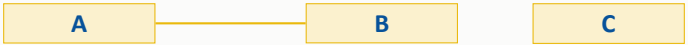
Motivation

Beim Modellieren von einem Projekt mit vielen Objekten und Typen, möchte man natürlich den Überblick behalten. Um in UML hervorzuheben, in welcher Beziehung Objekte oder Typen zueinander stehen, ohne dabei die Objekte oder Typen detailiert zu beschreiben, verwendet man **Associations**.

UNIVERSITÄT BONN

Associations

Mit der einfachsten **Association** drückt man aus, dass zwei Bausteine A und B inhaltlich zusammengehören, aber keine inhaltliche Beziehung zu C haben.



Der inhaltliche Zusammenhang wird durch das Diagram nur angedeutet. Der erweiterte Kontext des Diagrams (z.B. eine Projektdokumentation) muss den inhaltlichen Zusammenhang tiefergehend darstellen.

UNIVERSITÄT BONN

Speziellere Associations

Der UML Standard spezifiziert speziellere Arten von **Associations**. Wir stellen hier nur eine Auswahl dar. Zu den spezielleren Arten gehören unter anderem **Inheritance**, **Dependency**, **Aggregation** und **Composition**. Sie werden wie folgt dargestellt.

UNIVERSITÄT BONN

Inheritance und Dependency

Die **Inheritance** ist uns bereits bekannt. Wir schreiben D $\rightarrow\triangleright$ B, falls D durch öffentliche Subklassenbildung aus B hervorgeht.

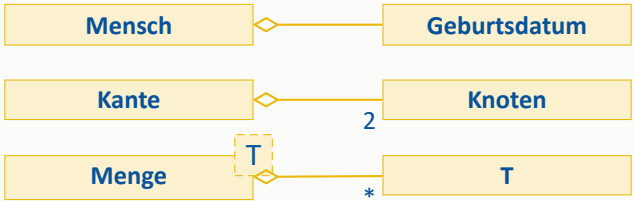
Man verwendet die **Dependency**, um eine inhaltliche Abhängigkeit zwischen Objekten oder Typen auszudrücken. Eine Unterart der **Dependency** ist die **Use-Dependency**. Mit ihr drückt man aus, dass ein Objekt X ein anderes Objekt Y verwendet. Beispielsweise weil Y erzeugt wird oder eine Memberfunktion von Y aufgerufen wird.

UNIVERSITÄT BONN

Aggregation

Wir wissen bereits, dass Objekte zusammengesetzt sind. Das haben wir in Klassendiagrammen durch den Attributblock ausgedrückt.

Bei Übersichtsdiagrammen werden Attribute und Schnittstellenfunktionen nicht genannt. Die Objektzusammensetzung (oder Teile davon) werden durch **Aggregations** ausgedrückt. Man kann dabei auszudrücken, wie oft ein Bestandteil vorkommen darf.



UNIVERSITÄT BONN

Composition

Die **Composition** ist eine spezielle Form der **Aggregation**. Im Allgemeinen drückt die **Aggregation** eine Hat-Ein-Beziehung aus. Falls die schäfere Besteht-Aus-Beziehung oder Ist-Verantwortlich-Für-Beziehung ausgedrückt werden soll, verwendet man die **Composition**.

Mit anderen Worten drückt die **Composition** aus, dass ein Bestandteil das „Eigentum“ von genau einem Objekt ist und die Lebensdauer des Bestandteils direkt an die Lebensdauer des Eigentümerobjekt gebunden ist. Das Eigentumsobjekt ist somit für die Geburt und die Vernichtung des Bestandteil verantwortlich.

UNIVERSITÄT BONN

Beispiele

Jede Datei ist in genau einem Verzeichnis vorhanden. Dateien sind Eigentum eines einzigen Verzeichnis. Wenn das Verzeichnis entfernt wird, muss die Datei ebenfalls entfernt werden.

Jeder Mensch ist Eigentümer seines Geburtsdatums.

Beim Umsetzen des Dependency Inversion Principle, können die referenzierten Interfaces auch ohne das Objekt der oberen Ebene existieren. Hier liegt also keine **Composition** vor.

UNIVERSITÄT BONN

Objekt- und Typbeziehungen in C++

In **C++** wird **Inheritance** und **Aggregation** strikt syntaktisch umgesetzt und erzwungen. Die restlichen Objektbeziehungen werden nicht strikt umgesetzt oder erzwungen.

Ein **Composition** wird oft (aber nicht immer) durch ein direktes Memberobjekt (also keine Referenzen oder Shared Pointer) modelliert.

Eine **Aggregation** die keine **Composition** ist, wird oft (aber nicht immer) durch Referenzen oder Shared Pointer modelliert.

Zusammenfassung

Beziehungen zwischen Objekten oder Typen geben wir durch Associations an

Speziellere Beziehungen beschreiben wir durch Aggregation, Composition, Inheritance und Dependency

Composition ist eine spezielle Aggregation

Haben Sie Fragen?

Entwurfsprinzipien

Ausblick

Ausblick

Wir haben in die grundlegende Entwurfsprinzipien der objektorientierten Modellierung eingeführt. Diese müssen einerseits praktisch einstudiert werden als auch durch fortgeschrittenere Prinzipien und Konzepte ergänzt werden.

Im kommenden Semester nehmen Sie am Modul **Praktikum Objektorientierte Softwareentwicklung** teil. Im Praktikum OOSE arbeiten Sie hier (aber nicht in einem Unternehmen) an drei vierwöchigen Softwareprojekten. Dort setzten Sie das hier Gelernte praktisch um und lernen weitere Konzepte der verteilten Entwicklung kennen.

Im dritten Semester nehmen Sie am Modul **Softwaretechnologie** teil. Dort lernen Sie weitere fortgeschrittenere Prinzipien und Konzepte der objektorientierten Modellierung kennen.

Rückschau und Ausblick

Zusammenfassung

Ab einer gewissen Größe werden Projekte üblicherweise objektorientiert modelliert. Zu den atomaren Modellierungskonzepten gehören **Klassenbildung** (zusammen mit Kapselung), **Vererbung** (zusammen mit Polymorphie, abstrakten Klassen und Interfaces) sowie **generische Programmierung**.

Entwurfsprinzipien (beispielsweise SOLID) setzen sich aus atomaren Konzepten zusammen. Sie helfen dabei die miteinander interagierenden Bestandteile des Projekts **nachvollziehbar**, **wartungsarm** und **flexibel erweiterbar** zu gestalten.

Die Modellierung geschieht im Allgemeinen in UML und wird anschließend konkret umgesetzt (z.B. in **C++**).

UNIVERSITÄT BONN

Ausblick

Die in diesem Modul eingeführten Konzepte sind grundlegend für weiterführende Module, Softwareprojekte und Abschlussarbeiten.

Aufbauend auf diesem Modul werden Sie sowohl Theorie als auch Praxis in anschließenden Modulen tiefergehend diskutieren. Die theorielastigen Konzepte werden stärker in der theoretischen Informatik vertieft und die praxislastigeren Konzepte werden stärker in der angewandten Informatik vertieft.

Um neue Konzepte zu erlernen gibt es viele Methoden. Veruchen Sie gern wie hier vorzugehen. Fragen Sie sich stets:

Welches Problem wird hier gelößt? oder **Welche Frage wird hier beantwortet?**

Am kommenden Mittwoch veranstalten wir eine offene Fragestunde