Entwurfsrichtlinien

by

Dr. Günter Kolousek

Tätigkeiten SW-Entwicklungsprozess

- Anforderungsanalyse (engl. requirements engineering)
  - Erfassung und Beschreibung der Anforderungen
  - Anforderungsdokument oder Lastenheft (Auftraggeber)
  - Pflichtenheft (akkordiert zw. Auftraggeber und Auftragnehmer)
  - Welche Funktionaliät soll das System aufweisen?
- Analyse (engl. analysis)
  - Systemelemente und deren Beziehungen zur Umwelt
  - Ist-Zustand vs. Sollzustand
  - Wie ist die Domäne aufgebaut?
- Entwurf (engl. design)
  - Architekturdokument
  - Wie wird die Software gebaut?
- Implementierung (engl. implementation)
- Testen
- Deployment (dt. Einsatz)

Heuristiken und Richtlinien

- Allgemeine Prinzipien
- Heuristiken des OO-Entwurfs
- SOLID - Prinzipien
- Finden von Klassen, Attributen, Assoziationen

Allgemeine Prinzipien

- KISS - Prinzip
- Vermeiden von Wiederholung
- Prinzip der minimalen Verwunderung

KISS - Prinzip

- Keep it simple and stupid
  - ”Simple is better than complex.”
  - ”Complex is better than complicated.”
  - ”If the implementation is hard to explain, it’s a bad idea.”
  - ”If the implementation is easy to explain, it may be a good idea.”
- Vermeide tiefe Hierarchien und Verschachtelungen
  - ”Flat is better than nested.”
  - ”Sparse is better than dense.”
- Einfachheit vor Allgemeinheit
  - Normale Dinge sollen einfach sein, besondere Dinge möglich…
  - Einfache Lösungen vor allgemeinverwendbaren Lösungen!
- keine Abstraktionen, die nicht benötigt werden
  - ”könnte man noch brachen”

Vermeiden von Wiederholung

- Don’t repeat yourself, DRY
- Keine Wiederholgung von Struktur oder Logik
  - aber: Ausnahmen bestätigen die Regel!
- Trennung der Veranwortlichkeiten
  - engl. Separation of concerns (SoC)
  - Arbeitsablauf um Teile eines Programmes möglichst ohne Überlappungen zu teilen

Pr. der minimalen Verwunderung

- Principle of least surprise (astonishment)
- Ergebnis soll offensichtlich, konsistent und vorhersehbar sein,
  - basierend auf den Namen und anderen Informationen
  - d.h. soll Erwartungen erfüllen

```
def add(a, b):
    return a * b
```

- erstaunliche Lösungen sind meist schwer verständlich
  - ”Explicit is better than implicit.”
  - ”In the face of ambiguity, refuse the temptation to guess.”
  - ”There should be one– and preferably only one –obvious way to do it.”
  - Eleganz oder Schönheit des Codes sind unwichtig!
    - verwendete Sprach/Bibliotheks/Framework-Features…
  - Nicht vergessen:

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. – Brian Kernighan

Heuristiken des OO-Entwurfs

- Object composition over class inheritance
  - verletzt Kapselung: Ober- und Unterklasse eng gekoppelt!
    - neue Methoden der Oberklasse können Unterklasse beeinflussen
- Maximum Cohesion (dt. Kohäsion, Zusammenhang)
  - zusammengehörige Daten und Verhalten in einer Einheit (z.B. Klasse)
- Minimum (Loose) Coupling (dt. Kopplung)
  - minimale Abhängigkeiten einer Klasse
    - einfachere Entwicklung, Wartung, Austauschbarkeit
    - u.U. geringere Performance
- Law of Demeter
  - Sprich nur zu deinen nächsten Freunden!
  - Eine Methode m einer Klasse C darf nur Methoden der folgenden Elemente aufrufen
    - C,
    - ein Objekt, das von m erstellt wird,
    - ein Objekt, das als Argument an m übergeben wird,
    - ein Objekt, das in einer Instanzvariablen von C enthalten ist.

SOLID - Prinzipien

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

- SRP
- Jede Klasse/Funktion hat genau eine Verantwortung und diese gut erledigen
  - und damit gibt es auch nur einen Grund diese zu ändern
    - „There should never be more than one reason for a class to change.“ (Robert C. Martin)
- Beispiel

```
struct GraphicPage {
    vector<Shape> items;
    void add(Shape item) {
        items.push_back(item);
    }
};
```

SRP – 2

- Jetzt soll eine Graphikseite auch gespeichert werden können
- naheliegend:

```
void GraphicPage::save(const string& filename) {
    ofstream fs{filename};
    for (auto& item : items)
        ofs << item << endl;
}
```

  - $\rightarrow$ Jetzt hat die Klasse 2 Verantwortungen!
  - Bei anderen Klassen, die speichern wollen wird genauso verfahren
  - Bei einer Änderung wie abgespeichert werden soll ist **jede** Klasse zu ändern, die Speichern kann!
  - Besser aufteilen auf 2 Klassen, z.B.:

```
struct PersistanceManager {
    static void save(const GraphicPage& gp) {
        ...
    }
}
```

Open-Closed Principle

- Klassen, Funktionen, Module, Packages… sollen offen für Erweiterungen, aber geschlossen für Änderungen sein.
- Beispiel

```
enum class Color { red, green, blue };
enum class LineStyle { dotted, dashed, solid };
struct Shape {
    Color color;
    LineStyle style;
};
struct ShapeFilter {
    typedef vector<Shape*> Items;
    Items by_color(Items items, Color color) {
        Items res;
        for (auto& i : items)
          if (i->color == color)
              result.push_back(i);
    }
};
```

  - …und jetzt `by_style`, `by_color_and_style`…
  - d.h. `ShapeFilter` wird immer *verändert*!

Open-Closed Principle – 2

```
template <typename T> struct Specification {
    virtual bool is_satisfied(T* item);  };
template <typename T> struct Filter {
    virtual vector<T*> filter(vector<T*> items,
                              Specification<T>& spec)=0; };
struct ShapeFilter : Filter<Shape> {
    vector<Shape*> res;
    vector<Shape*> filter(vector<Shape*> items,
                          Specification<T>& spec) {
    for (auto& p : items)
        if (spec.is_satisfied(p))
            res.push_back();
    return res; } };
struct ColorSpecification : Specification<Shape> {
    Color color;
    explicit ColorSpecification(Color color) : color{color} {}
    bool is_satisfied(Shape* shape) {
      return shape->color == color; }
};
```

→ `StyleFilter`, `ColorAndStyleFilter`,... keine Änderungen, statt dessen Erweiterungen!

Liskov'sches Substitutionsprinzip

- Wenn eine Schnittstelle ein Objekt des Typs `Parent` annimmt, dann soll es auch ein Objekt des Typs `Child` annehmen *ohne* den Vertrag des Schnittstelle zu brechen!
  - d.h. Unterklassen: anstelle ihrer Oberklassen einsetzbar!
- Beispiel:

```
class Rectangle : Shape {
  protected:
    int width; int height;
  public:
    int get_width() { return width; }
    virtual void set_width(int width_) { width = width_; }
    int get_heigth() { return height; }
    virtual void height(int height_) { height = height_; }
    int area() { return width * height; }
};
```

LSP – 2

```
class Square : Rectangle {
  public:
    Square(int size) : Rectangle(size, size) {}
    void set_width(width) override {
        this->width = height = width;
    }
};
```

LSP – 2

```
class Square : Rectangle {
  public:
    Square(int size) : Rectangle(size, size) {}
    void set_width(width) override {
        this->width = height = width;
    }
};
void process(Rectangle& r) {
    int w{r.get_width()};
    r.set_height(10);
    cout << "expected area:" << (w * 10)
         << ", got: " << r.area() << '!' << endl;
    // -> expected area:50, got: 25!
    // i.e. interface broken!!!
}
```

Interface Segregation Principle

- Prinzip der Abtrennung von Schnittstellen
  - Clients: nur von Schnittstellen abhängen, die sie benötigten!
- Beispiel:

```
struct MySpecialPrinter {
    void print(Document* doc);
    void fax(Document* doc);
    void scan(Document* doc);
};
```

Interface Segregation Principle

- Prinzip der Abtrennung von Schnittstellen
  - Clients: nur von Schnittstellen abhängen, die sie benötigten!
- Beispiel:

```
struct MySpecialPrinter {
    void print(Document* doc);
    void fax(Document* doc);
    void scan(Document* doc);
};
```

Hmm,... vielleicht doch ein Interface?

```
struct IPrinter {
    virtual void print(Document*)=0;
    virtual void fax(Document*)=0;
    virtual void scan(Document*)=0;
};

struct MySpecialPrinter : IPrinter {
    void print(Document* doc) override;
    void fax(Document* doc) override;
    void scan(Document* doc) override;
};
```

Interface Segregation Principle – 2

- Problem? Jetzt muss jeder der Schnittstelle implementiert die *gesamte* Funktionalität bereitstellen.
  - schlechter Notbehelf: implementieren von no-op Methoden…
- Lösung: Auftrennen des Interfaces!

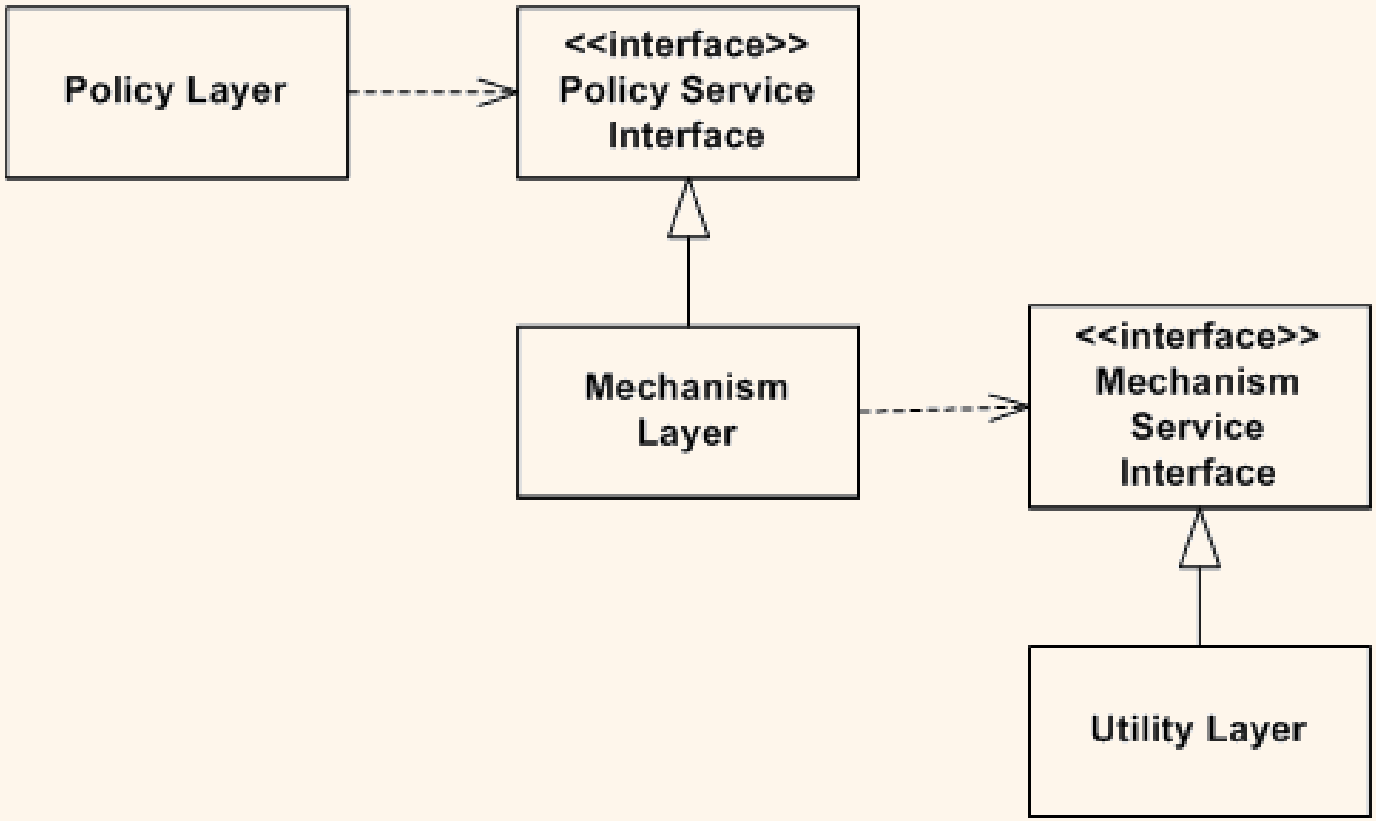
```
struct IPrinter {
  virtual void print(Document*)=0;
};
struct IScanner {
  virtual void scan(Document*)=0;
};
//... IFax
struct Printer : IPrinter {
    void print(Document*) override;
};
struct PrintAndScanMachine : IPrinter, IScanner {
    void print(Document*);
    void scan(Document*);
};
```

Dependency Inversion Principle

- ”Definition” (Martin, Robert C.)
  - Module hoher Ebene sollen nicht von Modulen niedriger Ebene abhängen. Beide sollten von Abstraktionen abhängen.
  - Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.
- Ausgangslage

Quelle: Wikipedia

DIP – 2


Policy Layer
<<interface>>
Policy Service
Interface
Mechanism
Layer
<<interface>>
Mechanism
Service
Interface
Utility Layer

DIP – 3

- Beispiel

```
struct Printer {
    GraphicProcessor* gp;
    ConsoleLogger* log;
    Printer() : gp{new GraphicProcessor()},
                log{new ConsoleLogger()} {}
};
```

  - Problem
    - higher-level Klasse hängt von lower-level Klassen ab
    - hohe Kopplung zwischen `Printer` und `GraphicProcessor` bzw. `ConsoleLogger`
    - Austauschbarkeit nicht gegeben
    - `Printer` legt Instanzen legt selbst an
    - `Printer` sollte eigentlich `GraphicProcessor` beinhalten

DIP – 4

- Verbesserung

```
struct Printer {
    unique_ptr<GraphicProcessor> gp;
    shared_ptr<ConsoleLogger> log;
    Printer(unique_ptr<GraphicProcessor> gp,
            const shared_ptr<ConsoleLogger>& log) :
            gp{move(gp)}, log{log} {}
};
```

  - legt nicht mehr selbst an, keine rohen Pointer mehr, `GraphicProcessor` gehört jetzt zu `Printer`
  - aber beim Anlegen von Printer müssen abhängige Objekte übergeben werden
  - Abhängigkeiten von higher-level zu lower-level noch immer gegeben

DIP – 5

- Verbesserung – 2

```
struct ILogger {
    virtual ~ILogger() {}
    virtual void log(const string& msg)=0;  };
struct ConsoleLogger : ILogger {
    ConsoleLogger() {}
    void log(const string& msg) override {
       /* ... */ } };
struct Printer {
    unique_ptr<GraphicProcessor> gp;
    shared_ptr<ILogger> log;
    Printer(unique_ptr<GraphicProcessor> gp,
            const shared_ptr<ILogger>& log) :
            gp{move(gp)}, log{log} {}  };
```

  - `ILogger` ersetzt `ConsoleLogger` → DIP nicht verletzt!
    - `GraphicProcessor` (absichtlich) nicht durch *interface* ersetzt
  - Abhängige Objekte müssen noch immer übergeben werden
    - was wenn diese wiederum Abhängigkeiten aufweisen?
    - hier hilft ein *Dependency Injection* Framework!

DIP – 6

- Dependency Framework
  - z.B. Boost.DI:

```
auto injector=di::make_injector(
  di::bind<ILogger>().to<ConsoleLogger>()
);

// jetzt kann ein neuer Printer angelegt werden
// ... und alle Abhängigkeiten werden automatisch
//     mit angelegt!
auto printer=injector.create<shared_ptr<Printer>();
```

Finden von Klassen, Attributen,...

- Abstraktionen aus bestehendem System
- Dokumentenanalyse
  - Syntaktische Analyse, d.h. finden von Nomen und Verben
  - Linguistische Analyse
  - Inhaltliche Analyse

Linguistische Analyse

Wortart	Modellelement	Beispiel
Nomen	Typ	Auto, Hund
Namen, konkrete Objekte	Instanz	Peter
intransitives Verb	Methode	laufen, schlafen
transitives Verb	Assoziation	etw. essen, jemanden liebe
Verb ”sein”	Vererbung	ist eine,…
Verb ”haben”	Aggregation	hat ein,…
Modalverb	Zusicherung	müssen, sollen
Adjektiv	Attribut	groß, 3 Jahre alt,…

… als erste Näherung!

Quelle: *Program Design by Informal English Descriptions*, Russel J. Abbott, 1983

Keine Klasse, wenn…

- weder Attribut (damit auch keine Assoziation) noch Operation
  - nur ein Attribut (oder sehr wenige) $\rightarrow$ vielleicht zu anderer Klasse zuordenbar
- gleiche Attribute,… wie andere Klasse
- nur Operationen, die sich anderen Klassen zuordnen lassen

Klassenname?

- Substantiv im Singular
- so konkret wie möglich
- soll die Gesamtheit der Attribute und Operationen darstellen
  - also das was es ausmacht
- soll nicht eine Rolle beschreiben, die diese Klasse in einer Assoziation zu einer anderen Klasse einnimmt

Finden von Assoziationen

- Verben…
  - räumliche Nähe (z.B. wohnt in)
  - Aktionen (z.B. fährt)
  - Kommunikation (z.B. redet mit)
  - Besitz (z.B. hat)
  - allgemeine Beziehungen (z.B. verheiratet)
- Fragestellungen
  - Liegen zwischen Objekten dauerhafte Beziehungen vor?
    - also nicht nur z.B. bei Aufruf von Operationen
  - Sind die beteiligten Klassen gleichrangig?
    - Aggregation vs. Assoziation
  - Assoziation gerichtet?