Die SOLID-Prinzipien

Fünf Grundsätze für bessere Software

Frank Listing
f.listing@microconsult.com

Problemkind Softwarequalität

Die Qualität der Software ist nicht in allen Projekten ideal. Deshalb werden in vielen Bereichen Anstrengungen unternommen, eine Qualitätssteigerung herbeizuführen.

Viele kluge Leute befassen sich mit dem Thema, wie die Codequalität verbessert werden kann.

Doch warum überhaupt guten Code schreiben? – Interessiert doch eh keinen.

Erfahrungswert:

Ein großer Teil der Programmierer schreibt schlechten Quellcode

→ "Was willst Du? Es läuft doch!"

MicroConsult

Schlechter Code verursacht:

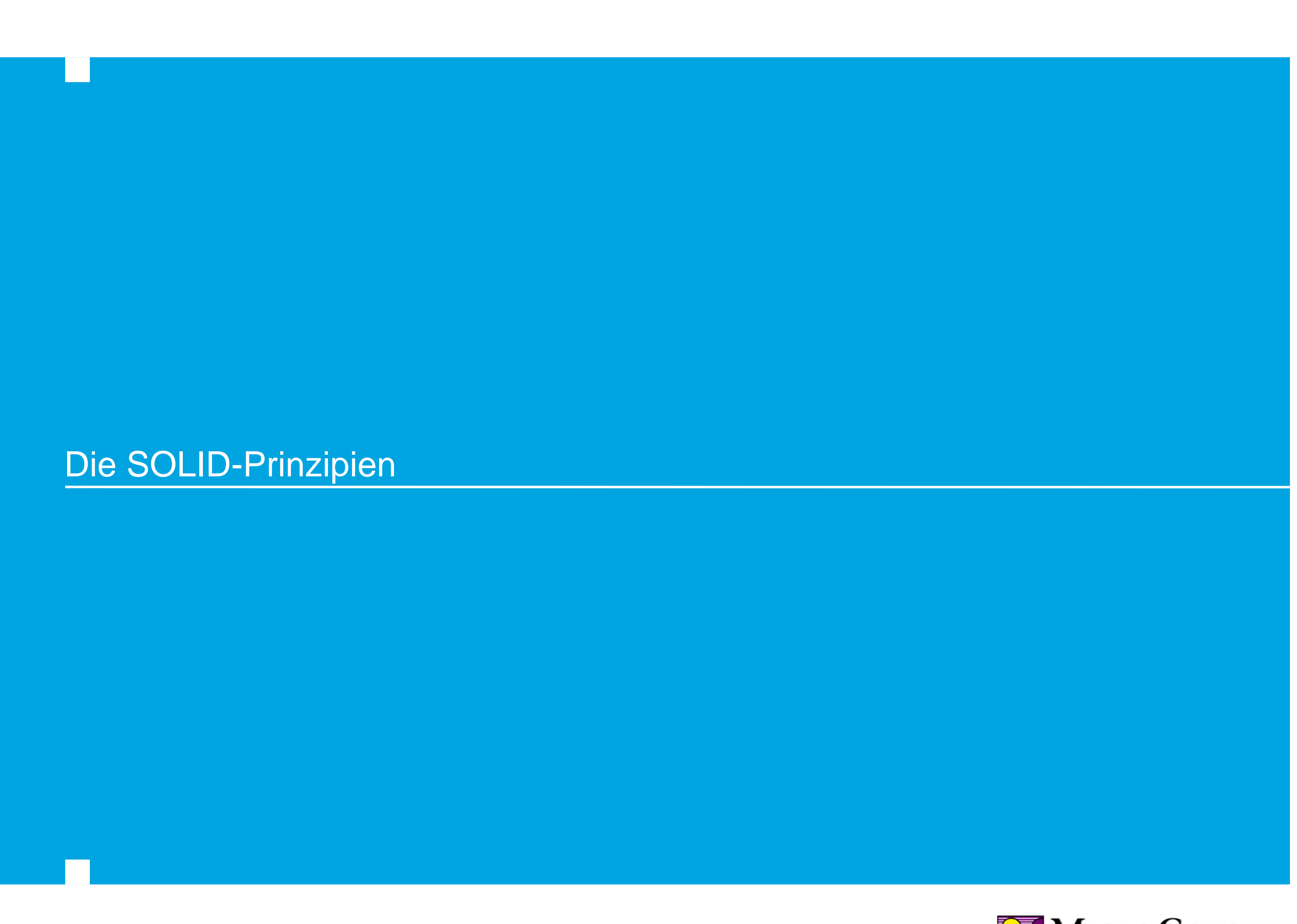
- Hohen Wartungsaufwand
- Hohe Kosten für die Weiterentwicklung
- Aufwändige Fehlersuche

Schlechter Code kostet Geld!

Warum ist guter Code wichtig?

- Code schreiben ist ein relativ kleiner Teil der Software-Entwicklung. Ca. 80% der Kosten sind Wartungskosten.
- Es wird wenig neuer Code geschrieben. Die hauptsächliche Arbeit besteht aus Änderungen im bestehenden Code.
  Der größte Anteil an der Arbeit ist nicht das Codieren, sondern das Verstehen (Lesen) von Code.
- Fehlerbehebungen in unverständlichem Code erzeugen schnell neue Fehler.
- Wenn am Anfang des Projektes auf Kosten der Codequalität Zeit gespart wird, wird das am Ende des Projektes ein Vielfaches der eingesparten Zeit kosten.

Guter Code reduziert unangenehme Arbeit


Die SOLID-Prinzipien

Um die Erstellung guten Codes zu erleichtern, wurden viele Prinzipen für die Softwareentwicklung formuliert.

Prominente Vertreter solcher Prinzipien sind die SOLID-Prinzipien.

SOLID wurde von Robert C. Martin geprägt. Es ist ein Akronym und steht für:

Single-Responsibility-Prinzip

Open-Closed-Prinzip

Liskovsches Substitutionsprinzip

Interface-Segregation-Prinzip

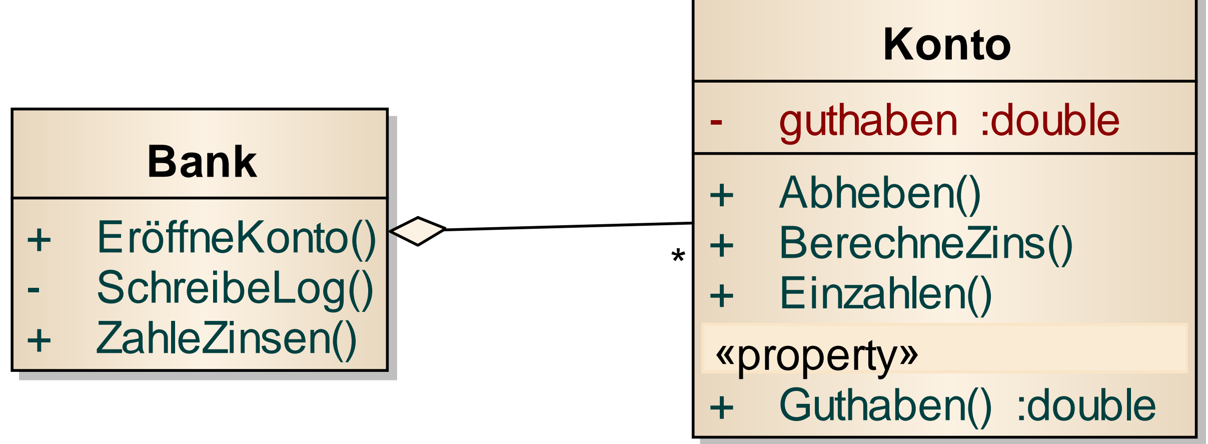
Dependency-Inversion-Prinzip

Single-Responsibility-Prinzip

Aufgabe:

Gegeben ist ein kleines und noch unvollständiges Bankprogramm.

In jeder Methode der Klasse Bank wird eine Logging-Ausgabe in eine Datei geschrieben.



Ändern Sie die Ausgabe des Loggings so, dass die Texte auf die Konsole ausgegeben werden.

Problem:

Das Logging ist fester Bestandteil der Klasse `Bank` – die Klasse `Bank` erfüllt mehr als eine Aufgabe.

Neben ihrer eigentlichen Tätigkeit kümmert sie sich auch um das Logging.

1. Die "Nebentätigkeit" ist von außen nicht sichtbar.
2. Änderungen im Logging sind äußerst mühsam.

Eine Klasse sollte nur eine Verantwortlichkeit haben.

Änderungen an der Funktionalität sollen nur Auswirkungen auf wenige Klassen haben.

Je mehr Code geändert werden muss, desto höher ist das Fehlerrisiko.

Hält man sich nicht an dieses Prinzip, führt das zu vielen Abhängigkeiten und hoher Vernetzung.

Ab einer bestimmten Größe wird ein Universalwerkzeug unhandlich.

Wie erkennt man, dass eine Klasse mehr als eine Aufgabe erfüllt?

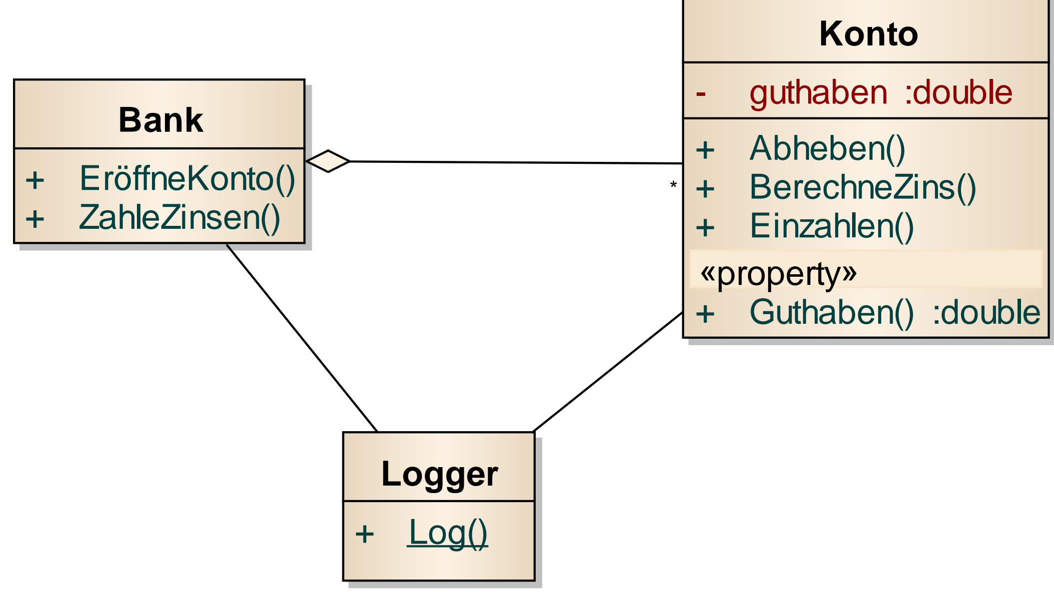
Sie hat mehr als einen Grund zur Änderung – d.h. wenn sich zwei Anforderungen ändern, darf nur eine davon eine Auswirkung auf die Klasse haben.

→ Lieber viele kleine Klassen als wenige Große.
Der Code wird dadurch nicht umfangreicher. Er wird nur anders organisiert.

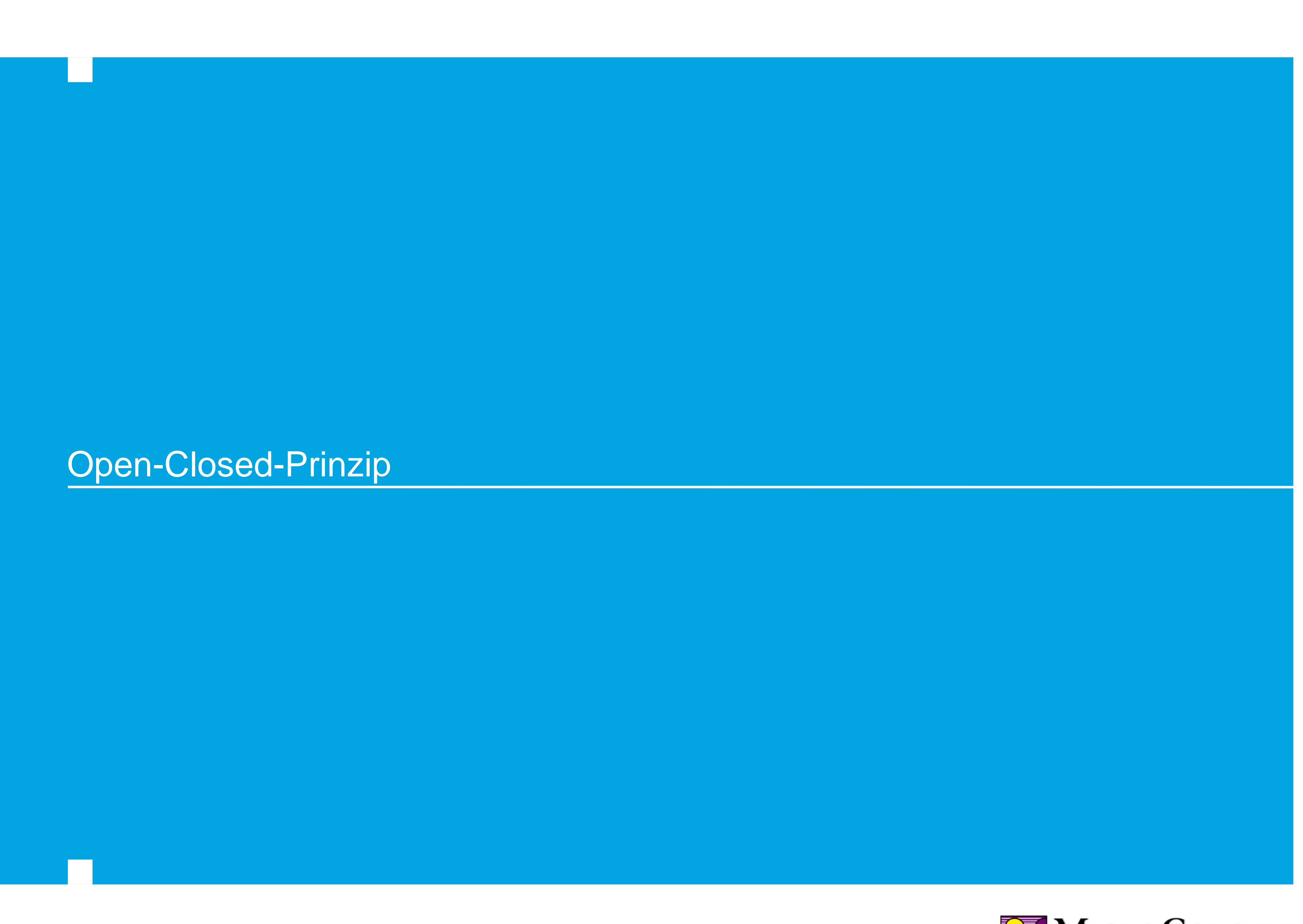
Wenn z.B. alle Schrauben im Bastelkeller in einer Kiste sind, ist es schwer, die Richtige zu finden. Sind sie in mehrere Schachteln sortiert, geht die Suche viel schneller. Genauso verhält es sich mit den Klassen.

Lösung:

Das Logging wird in eine eigene Klasse ausgelagert.



MicroConsult


Open-Closed-Prinzip

Aufgabe:

In dem schon bekannten Bankprogramm sollen für verschiedene Kontoarten angepasste Zinsen gezahlt werden.

Wie kann das erreicht werden?

© MicroConsult - Microelectronics Consulting & Training GmbH

MICROCONSULT

```
public class Konto
{
    ...

    public double BerechneZins()
    {
        double zinssatz = 0.0;

        switch (this.art)
        {
            case KontoArt.Girokonto:
                zinssatz = 0.001;
                break;
            case KontoArt.Sparkonto:
                zinssatz = 0.0075;
                break;
            default:
                break;
        }

        double zins = this.guthaben * zinssatz;
        return zins;
    }
}
```

Versuch einer Lösung

Wo ist das Problem?

Problem

Neue Funktionalität kann nur durch die Änderung einer bestehenden Klasse erreicht werden.

Dadurch entsteht das Risiko, dass bereits bestehende Funktionen nach der Änderung fehlerbehaftet sind.

Nach dem Open-Closed-Prinzip soll eine Klasse offen für Erweiterungen aber geschlossen gegenüber Modifikationen sein.

Das Verhalten einer Klasse darf erweitert aber nicht verändert werden.

Damit sollen Fehler in schon fertigen Codeteilen vermieden werden.

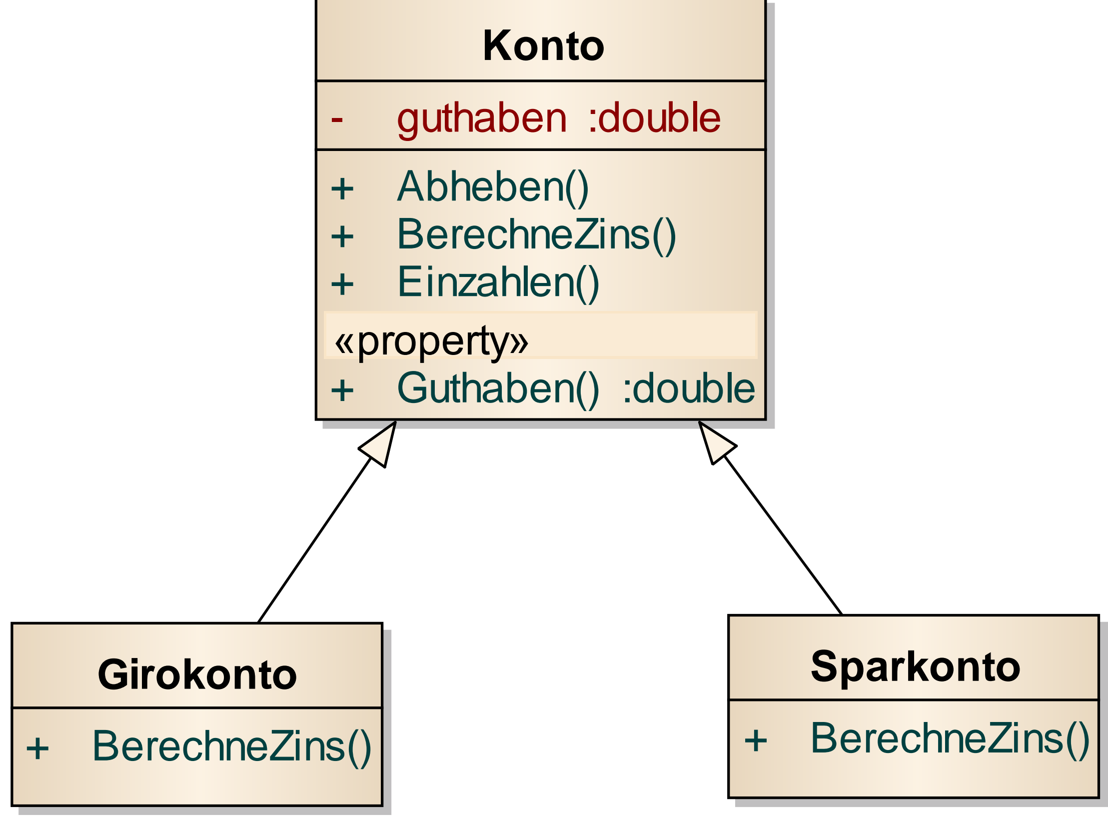
Dieses Prinzip kann normalerweise über zwei Wege erreicht werden:

1. Vererbung
2. Einsatz von Interfaces

Durch die Einhaltung dieses Prinzips können einer Applikation neue Funktionen hinzugefügt werden, ohne bestehende Klassen zu verändern.

Zinszahlung im Bankprogramm

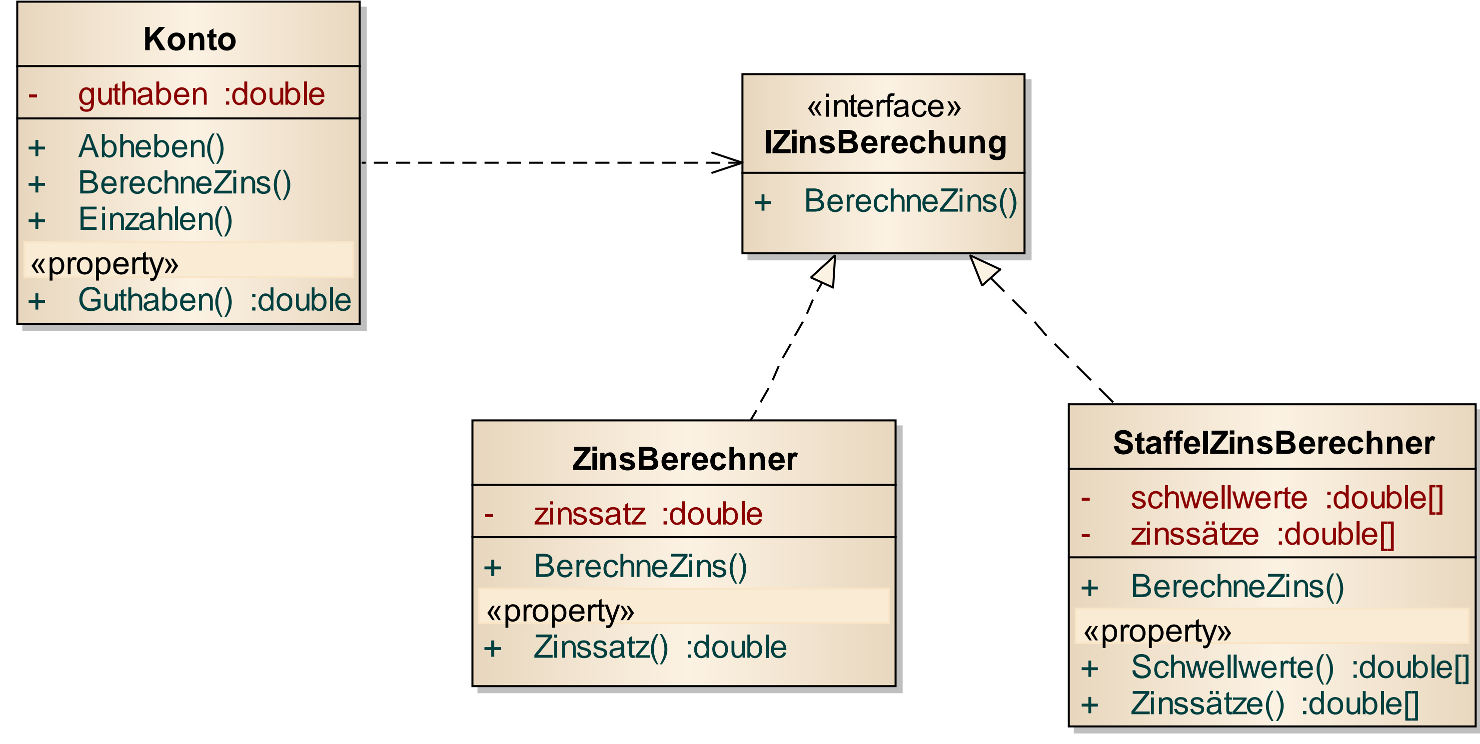
Erster Lösungsvorschlag: Vererbung und virtuelle Methoden.



MicroConsult

Zinszahlung im Bankprogramm

Zweiter Lösungsvorschlag: Nutzung des Strategie-Patterns.



Liskovsches Substitutionsprinzip

Aufgabe:

Die Bank möchte nun auch ein Taschengeldkonto anbieten. Die Kontoklasse ist bereits fertig implementiert.

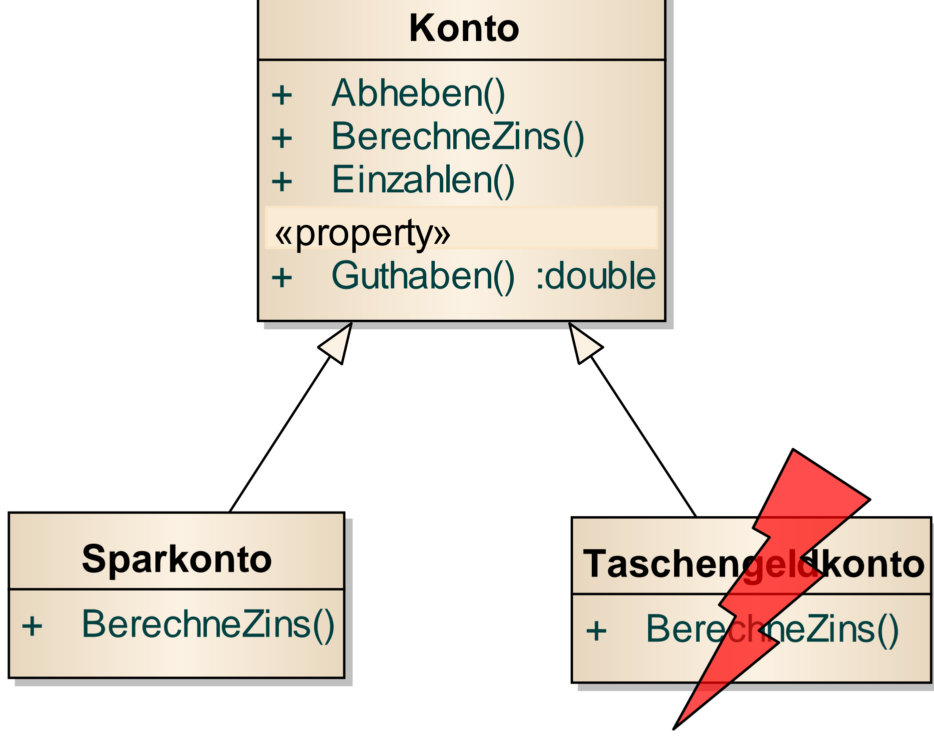
Da für das Taschengeldkonto keine Zinsen gezahlt werden, wird beim Aufruf der Methode `BerechneZins()` eine Exception geworfen.

MICROCONSULT

Problem

Die abgeleitete Klasse verhält sich nicht so, wie es nach dem Verhalten der Basisklasse erwartet wurde.

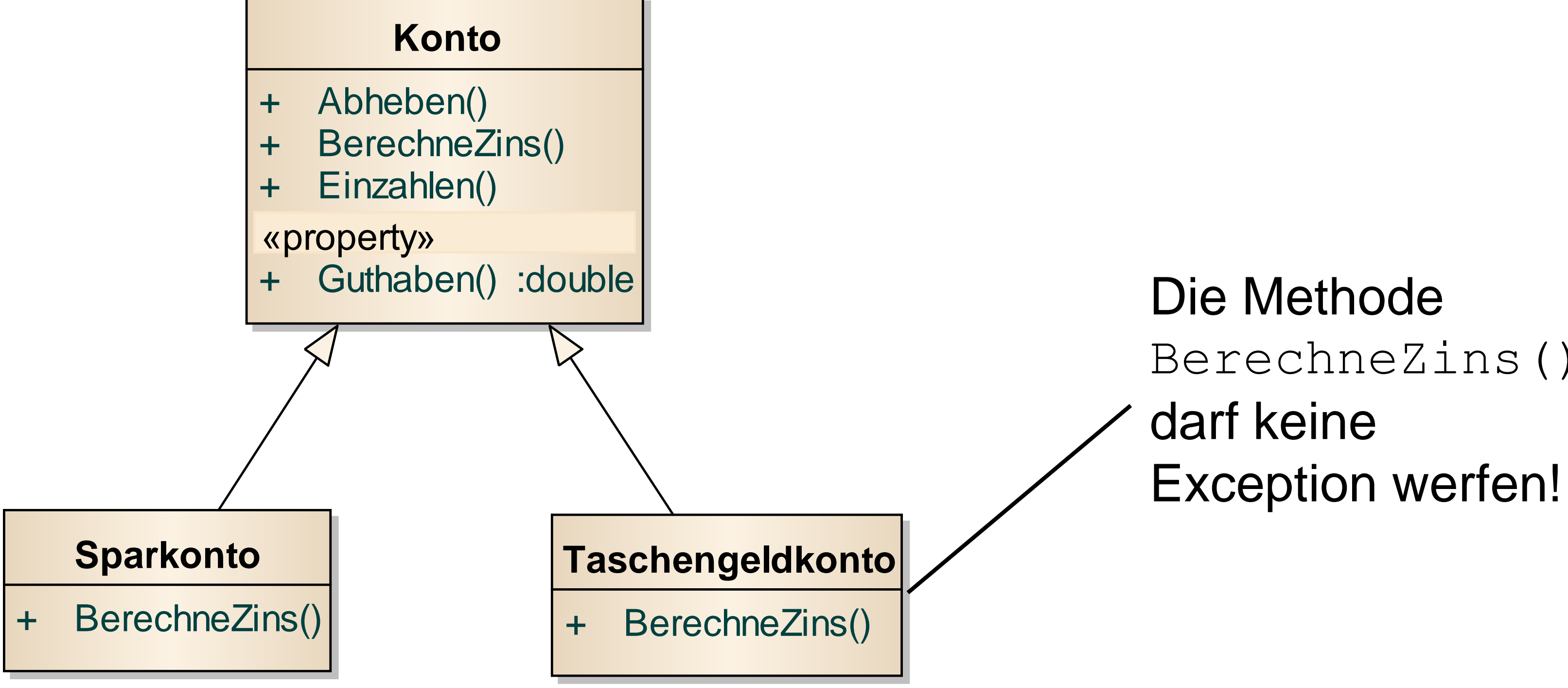
Plötzlich ist es notwendig zu wissen, welche konkrete Klasse hinter einem Objekt steckt.



© MicroConsult - Microelectronics Consulting & Training GmbH

Das Liskovsches Substitutionsprinzip fordert, dass abgeleitete Klassen immer anstelle ihrer Basisklasse einsetzbar sein müssen. → Subtypen müssen sich so verhalten, wie ihr Basistyp.

Eine abgeleitete Klasse darf ihre Basisklasse erweitern aber **nicht einschränken** oder verändern.



© MicroConsult - Microelectronics Consulting & Training GmbH

Interface-Segregation-Prinzip

Das Interface-Segregation-Prinzip besagt, dass ein Client nicht von den Funktionen eines Servers abhängig sein darf, die er gar nicht benötigt.

Ein Interface darf nur die Funktionen enthalten, die auch wirklich eng zusammengehören.

Die Problematik ist, dass durch "fette" Interfaces Kopplungen zwischen den ansonsten unabhängigen Clients entstehen.

Wird ein Aspekt des Interfaces verändert, hat das Auswirkung auf alle Clients. – Selbst wenn sie diesen Aspekt nicht nutzen.

© MicroConsult - Microelectronics Consulting & Training GmbH

MicroConsult

Ein anschauliches Beispiel liefert hier die AWT-Bibliothek von Java.

Soll lediglich auf das Ereignis zum Schließen des Fensters reagiert werden, müssen alle Methoden des Interfaces `WindowListener` implementiert werden.

```
public class Fenster extends Frame implements WindowListener
{
    ...

    // Methoden von WindowListener
    public void windowClosing(WindowEvent event)
    {
        System.exit(0);
    }
    public void windowClosed(WindowEvent event){}
    public void windowDeiconified(WindowEvent event){}
    public void windowIconified(WindowEvent event){}
    public void windowActivated(WindowEvent event){}
    public void windowDeactivated(WindowEvent event){}
    public void windowOpened(WindowEvent event){}
}
```

© MicroConsult - Microelectronics Consulting & Training GmbH

Was kann getan werden, wenn so ein Interface vorliegt und nicht geändert werden kann?

Es kann ein Adapter eingesetzt werden (Adapter-Entwurfsmuster).

Dieser Adapter implementiert alle Methoden des Interfaces mit einer Dummy-Implementierung und stellt diese virtuell zur Verfügung.

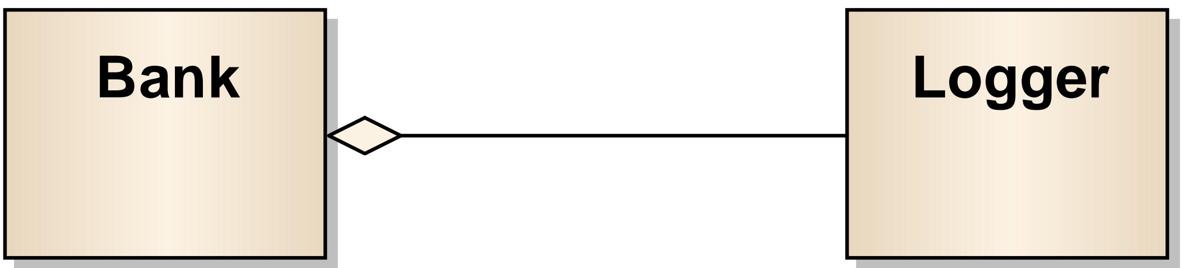
So einen Adapter stellt Java für das vorher gezeigte Beispiel zur Verfügung.

```java
class MyWindowListener extends WindowAdapter
{
    public void windowClosing(WindowEvent event)
    {
        System.exit(0);
    }
}
```

Dependency-Inversion-Prinzip

Aufgabe:

Das Bankprogramm hat einen gewissen Reifegrad erreicht und soll nun getestet werden.



Problem

Die Klassen sind zu stark miteinander verkoppelt.

Die Abhängigkeiten sind so stark, dass ohne Codeänderungen ein **separater Test** einer Klasse nicht möglich ist.

Auch Änderungen in den Anforderungen sind durch diese starke Kopplung schwerer umzusetzen.

Abhilfe schafft hier das Dependency-Inversion-Prinzip.

Dieses Prinzip besagt, dass Klassen auf einem höheren Abstraktionslevel nicht von Klassen auf einem niedrigen Abstraktionslevel abhängig sein sollen.

Dabei geht es aber nicht darum, die Abhängigkeiten einfach umzudrehen.

Abhängigkeiten zwischen Klassen soll es nicht mehr geben, es sollen nur noch Abhängigkeiten zu Interfaces bestehen (beidseitig).

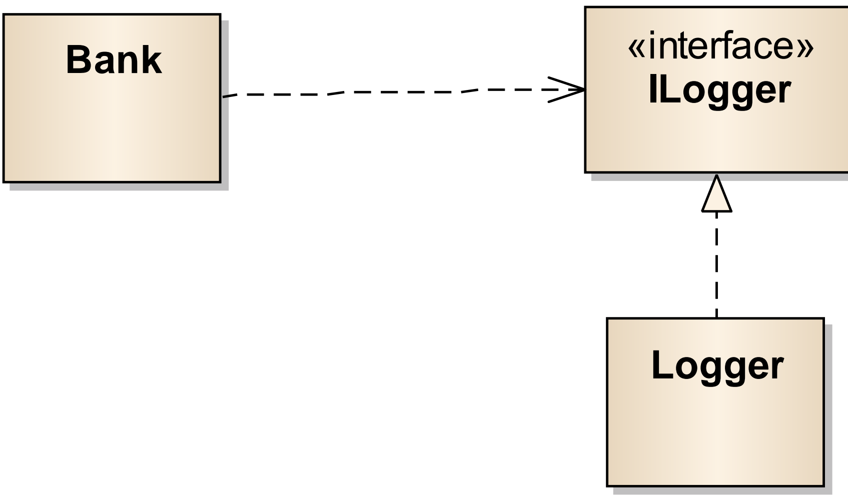
Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces.

Lösungsvorschlag 1 – Konstruktorparameter

Aggregation wird durch Assoziation ersetzt und die Abhängigkeit von einer speziellen Klasse in die Abhängigkeit zu einem Interface geändert.

Das konkrete Objekt (der Klasse `Logger`) wird als Parameter an den Kontruktor der Klasse `Bank` übergeben.

Das ist eine sehr einfache, aber nur bedingt flexible Lösung.



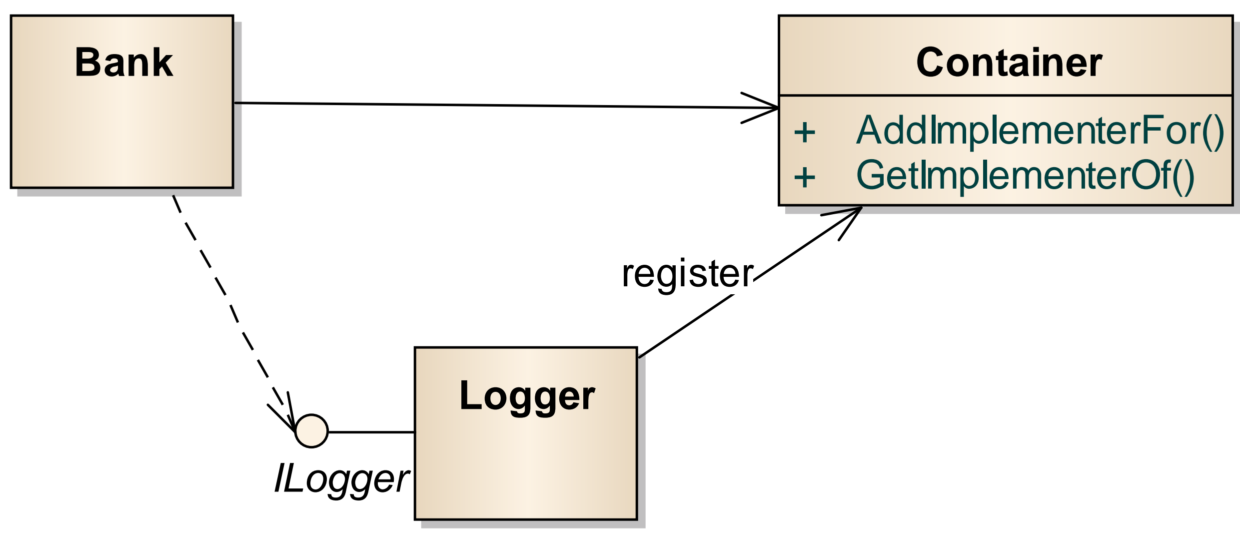
© MicroConsult - Microelectronics Consulting & Training GmbH

Lösungsvorschlag 2 – IoC-Container

Aggregation wird durch Assoziation ersetzt und die Abhängigkeit von einer speziellen Klasse in die Abhängigkeit zu einem Interface geändert.

Die konkrete Implementierung registriert sich beim IoC-Container.

Der Nutzer (die `Bank`) fragt im IoC-Container nach einer Implementierung des benutzten Interfaces (hier `ILogger`).



Die hier gezeigten Prinzipien sind Hinweise, die es einem Entwickler erleichtern, im Alltag die Codequalität zu verbessern.

Die (kleine) Mühe amortisiert sich sehr schnell. Änderungen werden einfacher, und auch Test und Fehlersuche werden beschleunigt.

© MicroConsult - Microelectronics Consulting & Training GmbH