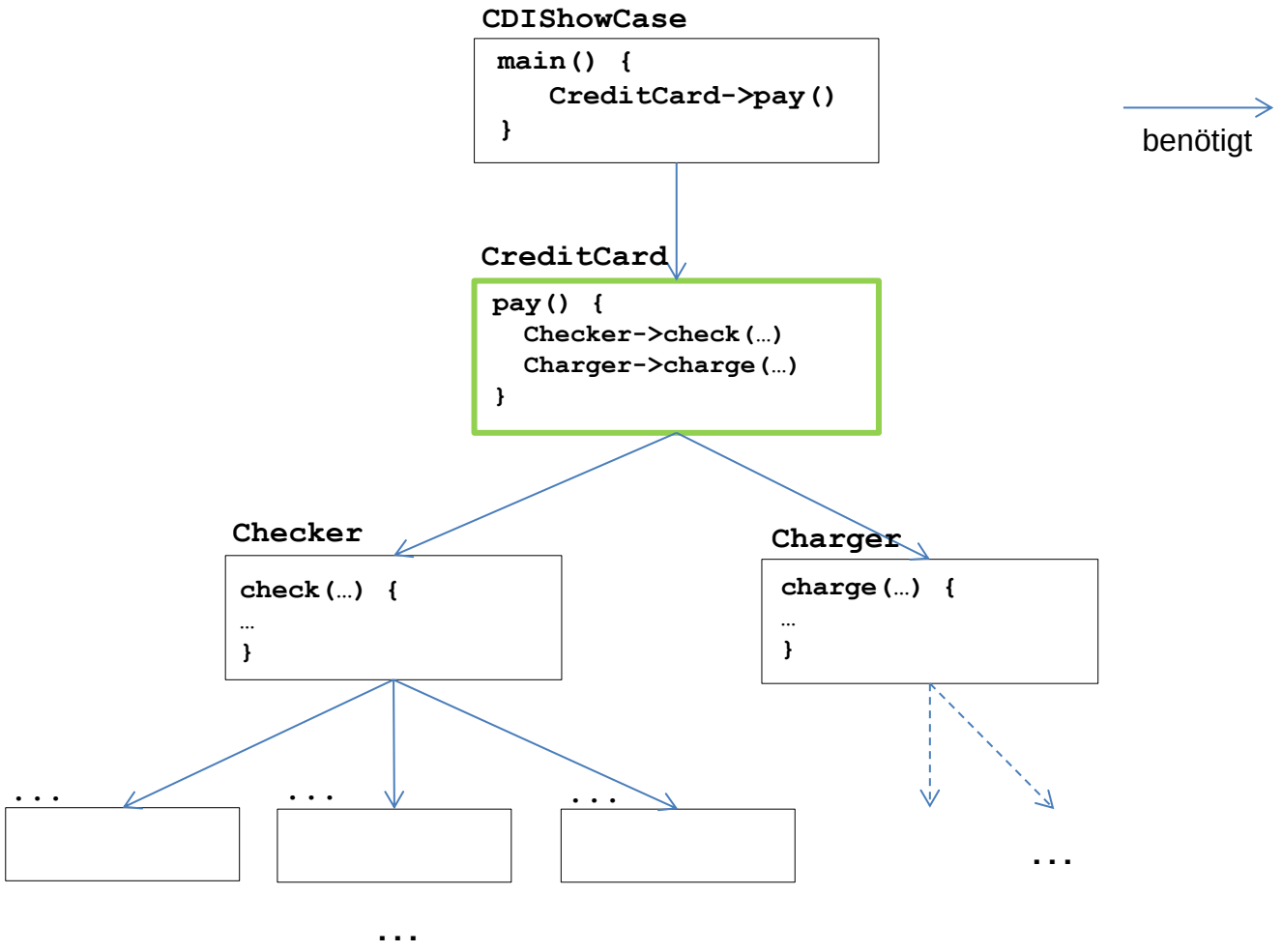
Modul 16
Context and Dependency Injection

CDI

Modul 16
CDI: Motivation



Modul 16
CDI: Motivation

```
public class CDIShowCase01 {
    public static void main(String[] args) {
        CreditCard myCreditCard = new CreditCard();
        myCreditCard.pay("123-98765-214");
    }
}
```

```
public class CreditCard {
    public void pay(String ccNr) {

        Checker myChecker = new Checker();
        if (myChecker.check(ccNr)==true) {
            System.out.println("CreditCard ok!");
        }

        Charger myCharger = new Charger();
        if (myCharger.charge(ccNr)==true) {
            System.out.println("CreditCard was charged!");
        }
    }
}
```

M16_CDIShowCase01

Modul 16
CDI: Motivation

Testen von **`CreditCard->pay()`** funktioniert nicht, weil:

- Jeder Test die Creditkarte belastet
- Die Methode **`Checker->check()`** (vielleicht) nicht funktioniert (?)
- Die Methode **`Charger->charge()`** (vielleicht) nicht funktioniert (?)

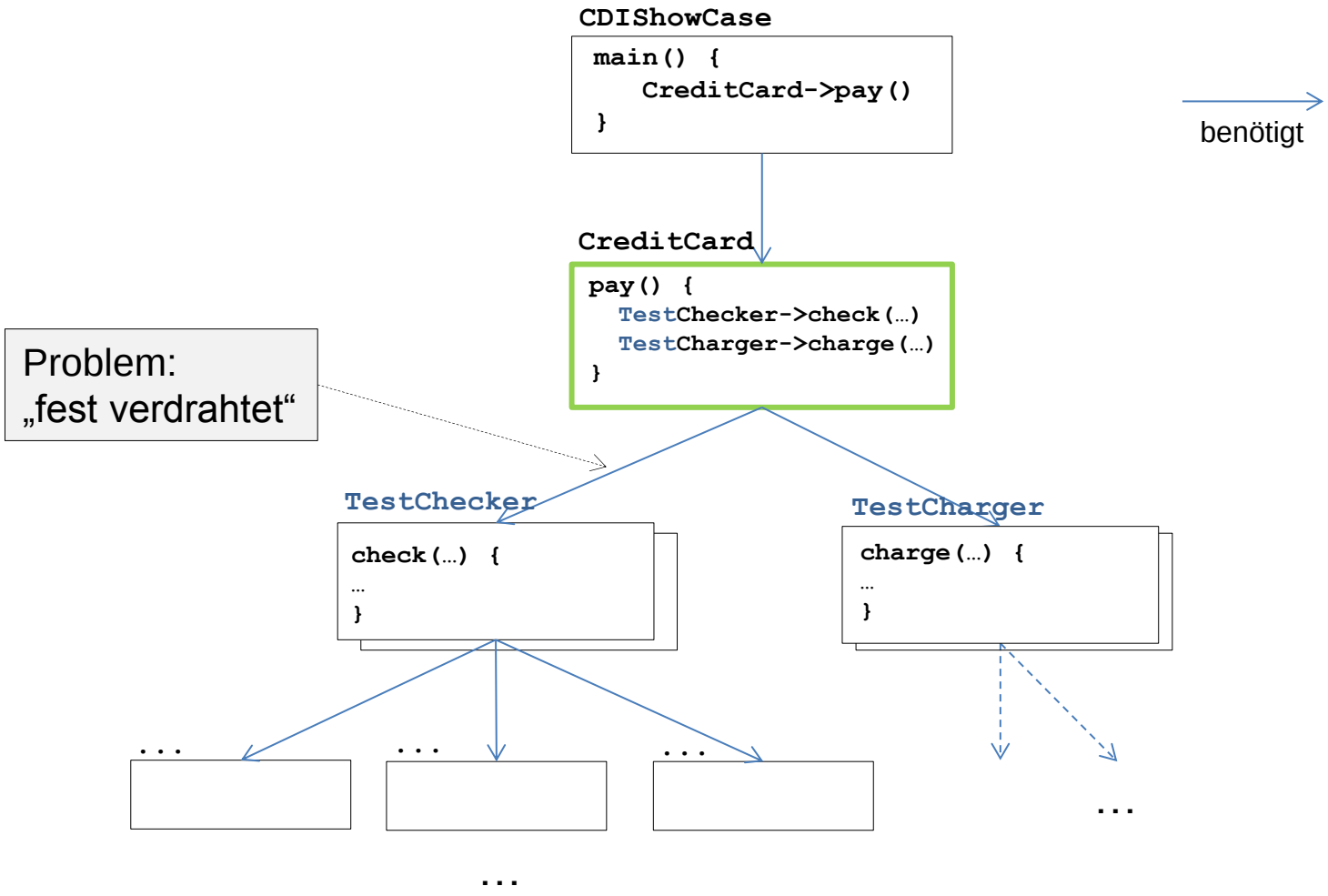
Somit:

- Die Klassen **`Checker`** und **`Charger`** ersetzen („mocken“) durch **`TestChecker`** und **`TestCharger`**

Aber:

- Dann muss der zu testende Quellcode verändert werden.

Modul 16
CDI: Motivation



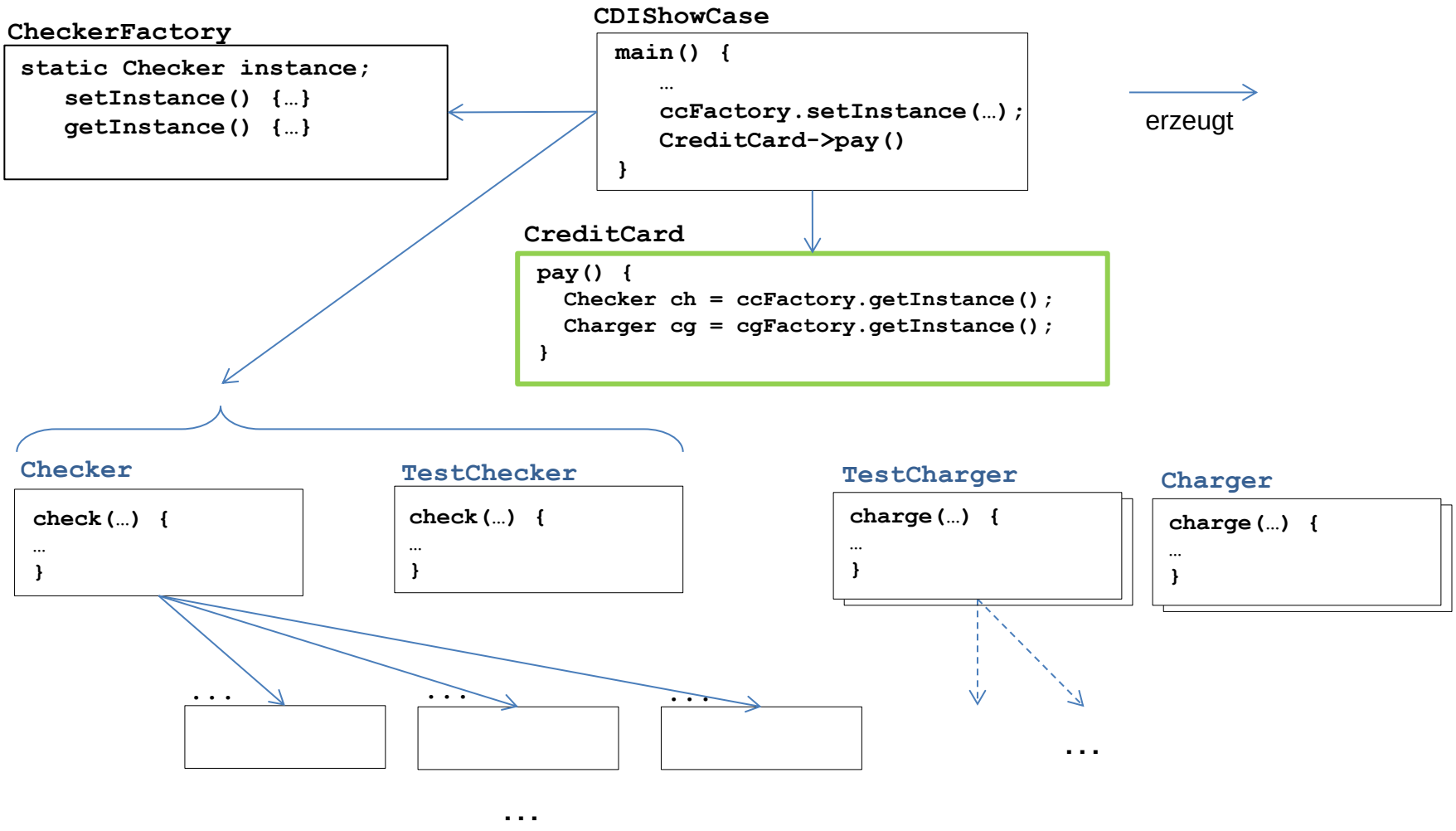
Modul 16
Entkopplung via Factory

Lösung 1 (Factory):

- Verwenden einer Factory
- Die Factory erzeugt zur Laufzeit die benötigten Objekte
  `Checker cc = checkerFactory.getInstance();`
- Die Factory kann zu Beginn so konfiguriert werden, dass sie ein Object der Klasse **`TestChecker`** oder der Klasse **`Checker`** zurückliefert[1].
  `checkerFactory.setInstance(new TestChecker());` [Test]
  `checkerFactory.setInstance(new Checker());` [Wirk]
- In der Klasse **`CreditCard`** wird dann das Objekt nicht erzeugt sondern mittels Factory “zugewiesen“.
  `cc = checkerFactory.getInstance();` statt
  `cc = new Checker();`

*TestChecker und Checker müssen dann von der gleichen Klasse / Interface abgeleitet sein.

Modul 16
Entkopplung via Factory



Modul 16
Entkopplung via Factory

Nachteil:

- Für jede Klasse eine Factory.
- Abhängigkeiten versteckt im Code – wo wird mit **`factory.getInstance(…)`** überall das (globale) Objekt „geholt“?
  Bsp.:
  Wenn wir via **`factory.setInstance(…)`** ein Test-Objekt instanziieren, wo wird es überall verwendet werden?
- Auswirkung (Fehler) erst zur Laufzeit
  Bsp.:
  Wir haben nach dem Testen vergessen, mit **`factory.setInstance(…)`** wieder das korrekte „Wirkbetrieb“-Objekt zu instanziieren. Der Fehler (ggf. eine Meldung) tritt erst zur Laufzeit auf, wenn mit **`factory.getInstance(…)`** zugegriffen wird.

See: http://code.google.com/p/google-guice/wiki/Motivation?tm=6

Modul 16
Dependency Injection by Hand (Interface)

Lösung 2 (DI per Interface):

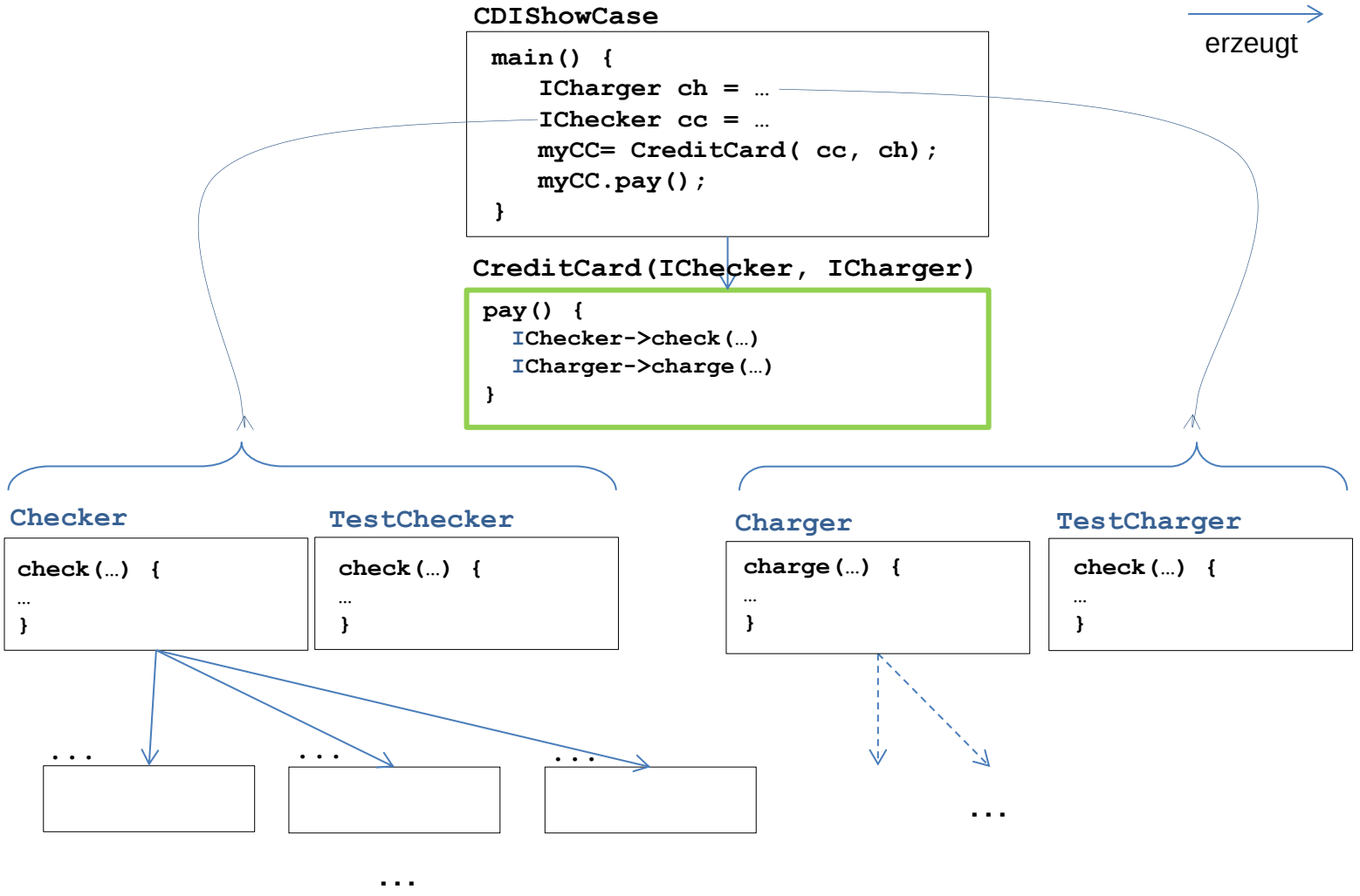
- Einführung eines Interfaces
  (hier Interface für **`Checker`** und **`Charger`**)

- Es gibt jeweils zwei Implementierungen für das Interface
  (eine Testimplementierung und eine Implementierung für den Wirkbetrieb)

- Die Objekte **`myChecker`** und **`myCharger`** werden nicht mehr in der zu testenden Methode erzeugt (kein **`new()`**), sondern über den Konstruktor übergeben.

- Es können nun Objekte der Testklasse oder Objekte der „Wirkbetriebs“-Klasse via Konstruktor übergeben werden.

Modul 16
Dependency Injection by Hand (Interface)



Modul 16
Dependency Injection by Hand (Interface)

```
public class CDIShowCase01 {
    public static void main(String[] args) {
        // For operating mode
        // CreditCard myCreditCard = new CreditCard(new Checker(),
                                                    new Charger());

        // For test mode
        CreditCard myCreditCard = new CreditCard(new TestCheck(),
                                                 new TestCharger());

        myCreditCard.pay("123-98765-213");
    }
}
```

```
public class CreditCard {

    IChecker myChecker;
    ICharger myCharger;

    public CreditCard(IChecker givenChecker, ICharger givenCharger) {
        this.myChecker = givenChecker;
        this.myCharger = givenCharger;
    }

    public void pay(String ccNr) {
          …
    }
}
```

No **new()**–Operator anymore!

M16_CDIShowCase02

Modul 16
Dependency Injection by Hand (Vererbung)

Lösung 3 (DI per Vererbung):

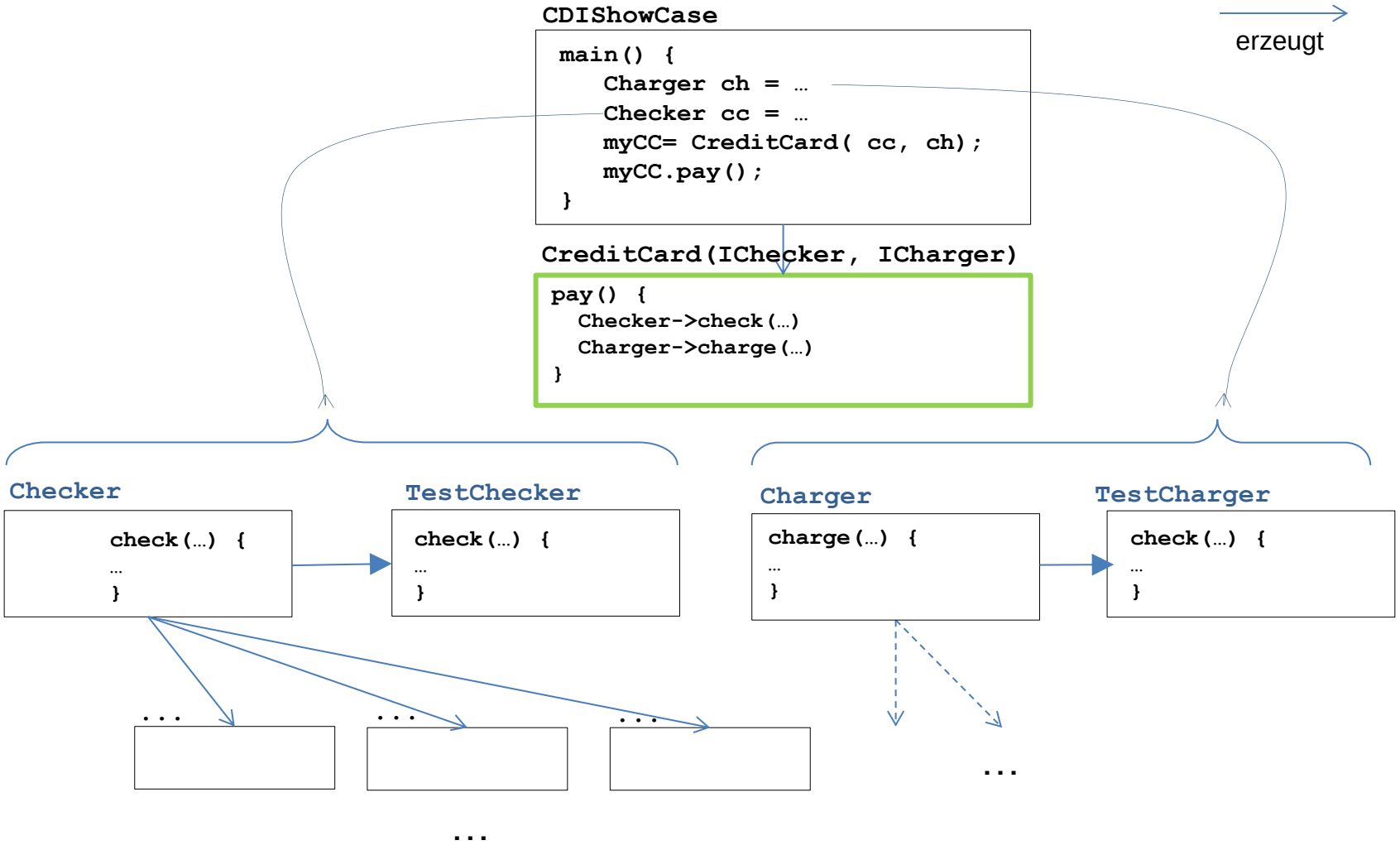
- Einführung einer Kindklassen jeweils für **`Checker`** und **`Charger:`**
  … **`TestChecker extends Checker`** …
  … **`TestCharger extends Charger`** …

- Es können jeweils zwei Objekte vom Type Checker erzeugt werden :
  - Zum Testen: **`Checker myChecker = new TestChecker();`** [upcast]
  - Wirkbetrieb: **`Checker myChecker = new Checker();`**

- Die Objekte **`myChecker`** und **`myCharger`** werden nicht mehr in der zu testenden Methode erzeugt (kein **`new()`**), sondern über den Konstruktor übergeben.

- Es können nun Objekte der Testklasse oder Objekte der „Wirkbetriebs“-Klasse via Konstruktor übergeben werden.

Modul 16
Dependency Injection by Hand (Vererbung)



Modul 16
Dependency Injection by Hand (Vererbung)

```
public class CDIShowCase01 {
    public static void main(String[] args) {
        // For operating mode
        // CreditCard myCreditCard = new CreditCard(new Checker(),
                                                    new Charger());

        Checker cc = new TestChecker();
        Charger cg = new TestCharger();
        CreditCard myCreditCard = new CreditCard(cc, cg);

        myCreditCard.pay("123-98765-213");
    }
}
```

```
public class CreditCard {

    IChecker myChecker;
    ICharger myCharger;

    public CreditCard(Checker givenChecker, Charger givenCharger) {
        this.myChecker = givenChecker;
        this.myCharger = givenCharger;
    }

    public void pay(String ccNr) {
          …
    }
}
```

No **new()**-Operator anymore!

M16_CDIShowCase03

Modul 16
Dependency Injection by Hand (Interface & Vererbung)

Unschön (DI Interface & DI Vererbung):

1. Was, wenn nun alle Klassen ihre “Abhängigkeiten” (Dependencies) in den Konstruktor verlagern? – Rekursive Auflistung von **`new()`**-Operatoren.

```
public class CDIShowCase01 {
  public static void main(String[] args) {
    // For operating mode
    // CreditCard myCreditCard = new CreditCard(new Checker( new CheckService() ),
                                                new Charger( new ChargeService() );

    // For test mode
    CreditCard myCreditCard = new CreditCard(new TestCheck(),
                                             new TestCharger());

    myCreditCard.pay("123-98765-213");
  }
}
```

2. Der Aufrufer muss Klassen kennen, die er nicht benutzt.

Modul 16
Dependency Injection with DI-Framework

DI-Framework („IoC-Container“):

1. Das Framework übernimmt die rekursive Auflistung / Ausführung der `new()`-Operatoren (Bootstrap-Phase).

2. Somit muss der Aufrufer nur Klassen kennen, die er auch benutzt.
   (Für den Test-Mode müssen natürlich alle Klassen bekannt sein, da diese ja durch Test-Klassen ersetzt werden.)

Modul 16
Dependency Injection with DI-Framework

```
public class CreditCard {

    public Checker myChecker;
    public Charger myCharger;

    @Inject
    public CreditCard(Checker givenChecker, Charger givenCharger) {
        this.myChecker = givenChecker;
        this.myCharger = givenCharger;
    }

    public void pay(String ccNr) {
          …
    }
}
```

Constructor-Injection

Constructor (triggered)-Injection:

Wird ein neues Objekt (via **`@Inject`**) erzeugt, wird überprüft, ob ein Konstruktor existiert, der mit **`@Inject`** annotiert wurde.
Wenn ja, werden die in dem Konstruktor enthaltenen Parameter (hier: **`givenChecker`** und **`givenCharger`**) vom Container erzeugt. Enthalten diese wiederum Konstuktoren mit **`@Inject`**, werden diese rekursiv aufgerufen.

Der Konstruktor kann natürlich auch explizit aufgerufen werden und es können “von außen” die gewünschten (Test-)Objekte übergeben werden.

Modul 16
Dependency Injection with DI-Framework

```
public class CDIShowCase01 {

    // For operation mode ----------------------------------------
    // @Inject
    // CreditCard myCreditCard;

    // For test mode ---------------------------------------------
    // Upcast requiers no explicit casting
    // Charger cg = new TestCharger();
    // Checker cc = new TestChecker();

    public void start(String[] args) {
        // For test-mode
        // CreditCard myCreditCard = new CreditCard(cc, cg);
        myCreditCard.pay("123-98765-213");
    }
}
```

Field-Injection

Field-Injection:

Um für ein Objekt die (rekursive) Constuctor-Injection „loszutreten“, muss es via **`@Inject`** erzeugt werden.

Das Objekt kann natürlich auch “konventionell” via new() erzeugt werden und es können “von außen” die gewünschten (Test-)Objekte an den Konstruktor übergeben werden.

M16_CDIShowCase04a

Modul 16
Dependency Injection with DI-Framework

```
public class CreditCard {

    public Checker myChecker;
    public Charger myCharger;

    @Inject
    public void setMyChecker(Checker myChecker) {
        this.myChecker = myChecker;
    }

    @Inject
    public void setMyCharger(Charger myCharger) {
        this.myCharger = myCharger;
    }

    public void pay(String ccNr) {
          …
    }
}
```

Setter-Injection / Method-Injection

Setter (triggered)-Injection / Method Injection:

Wird ein neues Objekt (via **`@Inject`**) erzeugt, werden alle Methoden ausgeführt, die mit **`@Inject`** annotiert wurde.
Die in den Methoden enthaltenen Parameter (hier: **`myChecker`** und **`myCharger`**) werden vom Container erzeugt. Enthalten diese wiederum Konstuktoren mit **`@Inject`**, werden diese rekursiv aufgerufen.

Die Methoden können natürlich auch explizit aufgerufen werden und es können “von außen” die gewünschten (Test-)Objekte übergeben werden.

M16_CDIShowCase04b

Modul 16
Dependency Injection with DI-Framework: Qualifier

Qualifier:
Gibt es mehrere Implementierungen einer Klasse (z.B. via Interfaces **`implements`** oder via Vererbung **`extends`**), so kann eine Implementierung durch einen s.g. *Qualifier* eindeutig gekennzeichnet werden.

Definition

```
@Qualifier
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ElementType.METHOD, ElementType.FIELD,
             ElementType.TYPE, ElementType.PARAMETER})
    public @interface Testing2 {}
```

```
@Testing2
public class TestChecker extends Checker {
    @Override
    public boolean check(String ccNr) {
        return true;
    }
}
```

Aufruf

```
@Inject @Testing2 Checker cc;
@Inject @Testing2 Charger cg;
```

M16_CDIShowCase04b

Modul 16
Context and Dependency Injection

Context:

Objekte die „injiziert“ wurden, werden vom Container verwaltet. Ihr Lebenszyklus (scope) kann somit vom Container verwaltet werden.

`@ApplicationScoped`

Das Objekt existiert nur einmal in der gesamten Anwendung, auch wenn es mehrmals mittels `@Inject` an verschiedenen Stellten „erzeugt“ wird. (Ersatz für globale Variablen).

`@Dependent`

Das Objekt wird mittels `@Inject` an verschieden Stellen neu „erzeugt“.
(Ersatz für `new()`).

In JEE noch: `@RequestScoped` `@ViewScoped` `@SessionScoped` `@ViewScoped`

```
@ApplicationScoped
public class Globals {
    private int globCounter = 0;

    public int getGlobCounter() {
        return globCounter;
    }

    public void setGlobCounter(int globCounter) {
        this.globCounter = globCounter;
    }
}
```

M16_CDIShowCase05

Modul 16
Dependency Injection - Summary

Modul 16
Dependency Injection - Injection points

Bean constructor parameter injection:

```
public class Checkout {

  private final ShoppingCart cart;

  @Inject
  public Checkout(ShoppingCart cart) {
    this.cart = cart;
  }

}
```

Quelle:http://docs.jboss.org/weld/reference/latest/en-US/html/injection.html

Modul 16
Dependency Injection - Injection points

Initializer method parameter injection:

```
public class Checkout {

  private ShoppingCart cart;

  @Inject
  void setShoppingCart(ShoppingCart cart) {
    this.cart = cart;
  }

}
```

Quelle:http://docs.jboss.org/weld/reference/latest/en-US/html/injection.html

Modul 16
Dependency Injection - Injection points

And direct field injection:

```
public class Checkout {

  private @Inject ShoppingCart cart;


}
```


Quelle:http://docs.jboss.org/weld/reference/latest/en-US/html/injection.html

Modul 16
Dependency Injection - Qualifiers

There might be two implementations of PaymentProcessor:

```
@Synchronous
public class SynchronousPaymentProcessor implements PaymentProcessor {
  public void process(Payment payment) { ... }
}
```

```
@Asynchronous
public class AsynchronousPaymentProcessor implements PaymentProcessor {
  public void process(Payment payment) { ... }
}
```

Where @Synchronous and @Asynchronous are qualifier annotations:

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Synchronous {}
```

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Asynchronous {}
```

Quelle:http://docs.jboss.org/weld/reference/latest/en-US/html/injection.html

Modul 16
Dependency Injection - Qualifiers

A client bean developer uses the qualifier annotation to specify exactly which bean should be injected.

Using field injection:

```
@Inject @Synchronous PaymentProcessor syncPaymentProcessor;
@Inject @Asynchronous PaymentProcessor asyncPaymentProcessor;
```

Using initializer method injection:

```
@Inject
public void setPaymentProcessors(@Synchronous PaymentProcessor syncPaymentProcessor,
                        @Asynchronous PaymentProcessor asyncPaymentProcessor) {
  this.syncPaymentProcessor = syncPaymentProcessor;
  this.asyncPaymentProcessor = asyncPaymentProcessor;
}
```

Using constructor injection:

```
@Inject
public Checkout(@Synchronous PaymentProcessor syncPaymentProcessor,
            @Asynchronous PaymentProcessor asyncPaymentProcessor) {
  this.syncPaymentProcessor = syncPaymentProcessor;
  this.asyncPaymentProcessor = asyncPaymentProcessor;
}
```

Quelle:http://docs.jboss.org/weld/reference/latest/en-US/html/injection.html

Modul 16
Dependency Injection “by idiots”

Austausch aller **`new()`**-Operatoren durch Field Injection

Vorher

```
public class Checkout {

        private ShoppingCart cart = new ShoppingCart();

        void doSomeThing() {}
}
```

Nachher

```
public class Checkout {

        private @Inject ShoppingCart cart;

        void doSomeThing() {}
}
```

Nutzen: ?

DI-Frameworks:

- Weld 2.0 (JEE Referenzimplementierung, auch für Java SE verwendbar)
- Spring-Framework
- Google Guice
- Pico-Container
- …