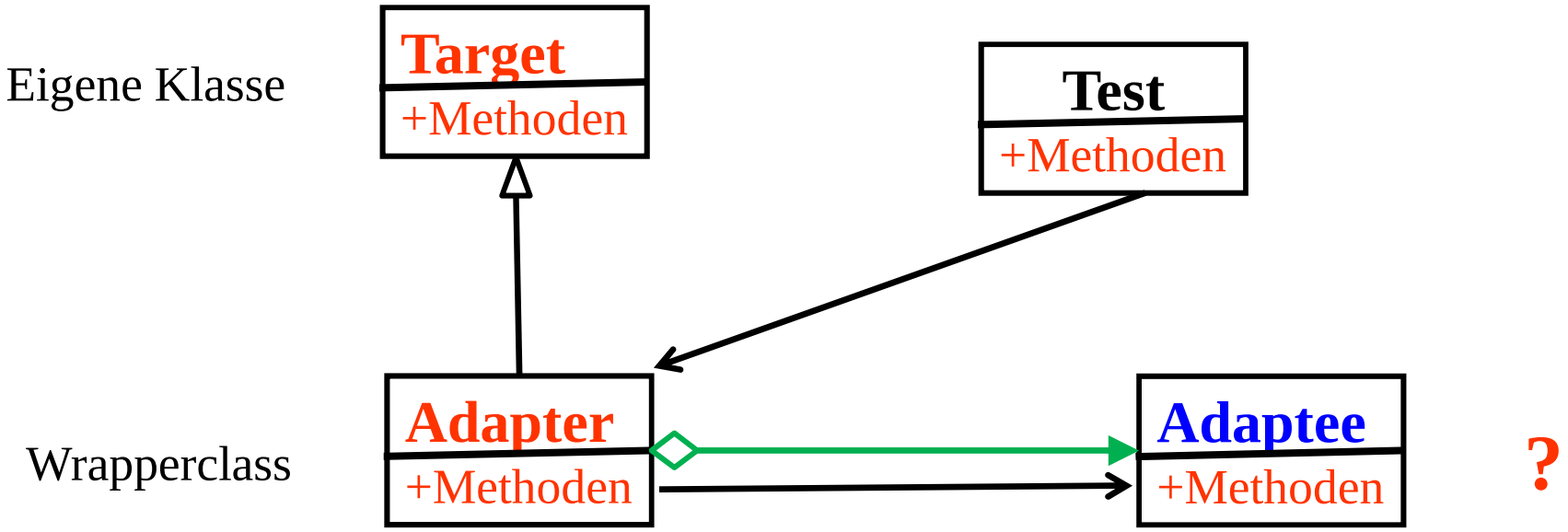
Wahlpflichtfach Design Pattern

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- miwilhelm@hs-harz.de
- http://www.miwilhelm.de
- Raum 2.202
- Tel. 03943 / 659 338

Inhalt

1. Einleitung
2. Singleton
3. Observer
4. Decorator
5. Abstract Factory
6. Adapter
7. Facade
8. Mediator
9. **Bridge**
10. MVWM
11. Java Collection Framework
12. Command / Befehl

1: „Adapter“ mit Ableitung: EINE SEITE



Adapter-Pattern

- Die Klasse Adapter wird von der Target-Klasse abgeleitet und hat damit Zugriff auf die „Problemklasse“.
- Mittels einer Komposition hat man nun Zugriff auf die Transformation.
- Einbau aller Methoden der Adapter-Klasse um die Adaptee-Methoden aufrufen zu können.
- **Adaptee muss flexibler sein**

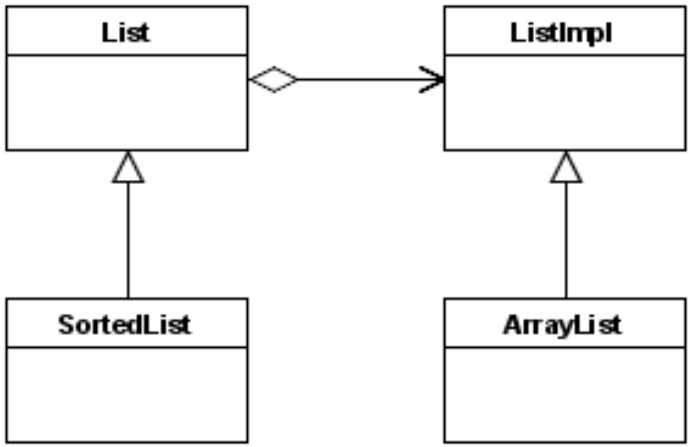
Entwurfsmuster: „Bridge“

- Im Adapter-Pattern hat man nur die eigene Klasse gekapselt.
- Möchte man nun die „fremde Instanz“ austauschen, benutzt man das Bridge-Pattern. Dort können Klassen à la Decorator hinzugefügt werden.
- Im Pattern Bridge werden beide also beide Seite gekapselt.
- Man ist dadurch etwas flexibler.
- Durch Angabe eines Parameters bei der Erzeugung einer Abstraktion kann die Implementierung gewählt werden (Fabrik).
- **Bridge-Pattern**
  - Es existieren zwei Klassenfamilien:
    - Abstrakte Klasse
    - Abgeleitete Instanz (hat Instanz des Interfaces)

    - Interface
    - Instanz(en) mit Interface-Methoden

Beispiele „Bridge“

Abstraktion Implementierer



- Die Abstraktion (im Beispiel: List) definiert einerseits die Schnittstelle der Abstraktion, andererseits hält sie eine **Referenz** zu einem Implementierer.
- Die SpezAbstraktion (im Beispiel: SortedList) erweitert die Schnittstelle.
- Der Implementierer (im Beispiel: ListImpl) definiert die Schnittstelle der Implementierung. Er kann sich dabei von der Schnittstelle der Abstraktion erheblich unterscheiden.
- Der KonkrImplementierer (im Beispiel: ArrayList) enthält eine konkrete Implementierung durch Implementierung der Schnittstelle.

https://de.wikipedia.org/wiki/Brücke_(Entwurfsmuster)

Entwurfsmuster: „Bridge“

Wann benutzen:

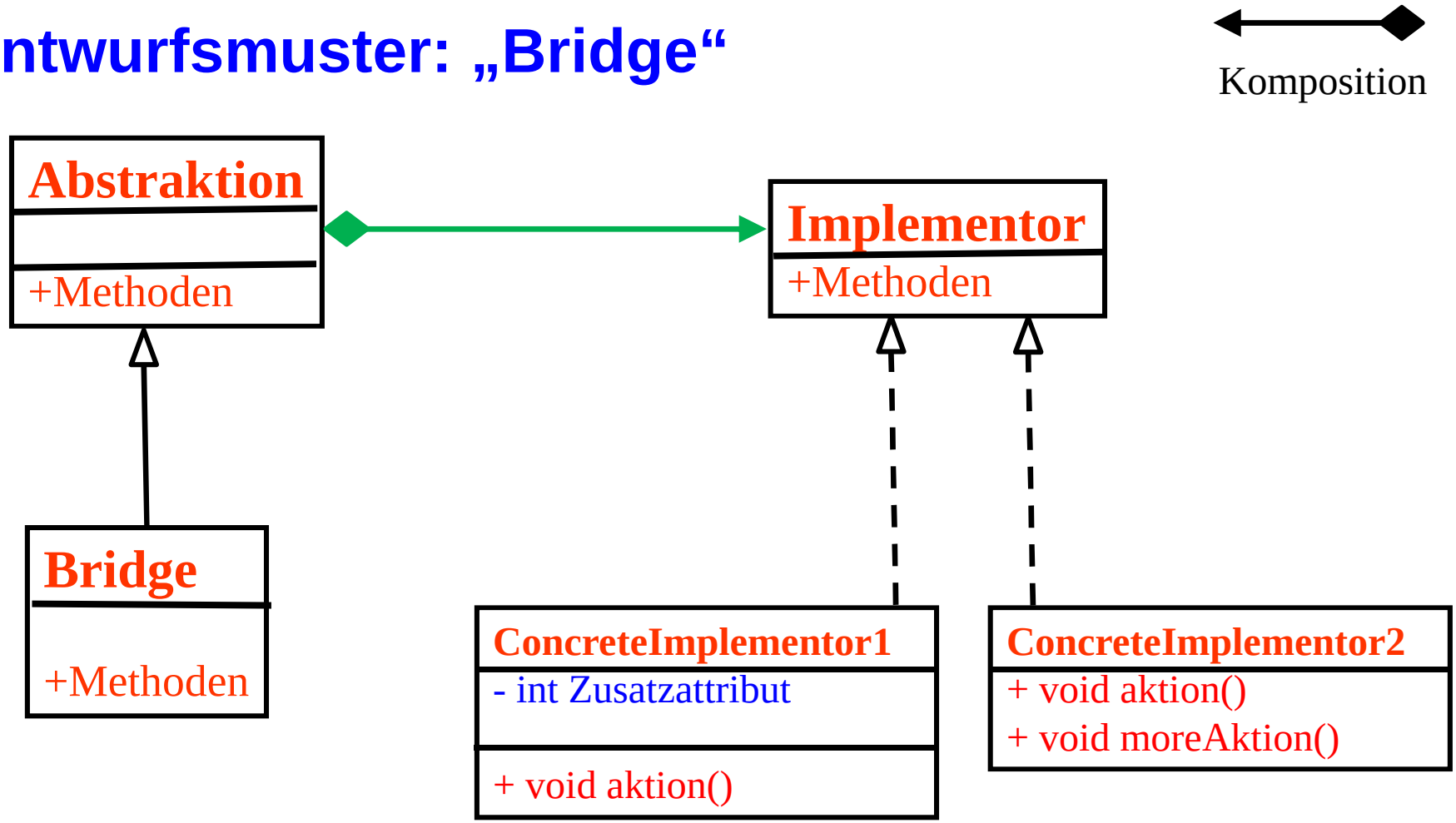
- Eine Brücke findet Anwendung, wenn sowohl **Abstraktion** als auch **Implementierung** erweiterbar sein sollen und eine dauerhafte Verbindung zwischen Abstraktion und Implementierung verhindert werden soll.
- Eine Änderungen der Implementierung **ohne Auswirkungen** für den Klienten sein sollen
- die Implementierung vor dem Klienten verborgen bleiben soll
- die Implementierung von verschiedenen Klassen gleichzeitig genutzt werden soll.
- In der Praxis kommt es oft vor, dass die Abstraktionsseite nicht so feingranular untergliedert wird wie die Implementierungs-seite. Man spricht von einer degenerierten Brücke.

Entwurfsmuster: „Bridge“

- Erzeugt **zwei unterschiedliche** Klassenhierarchien.
- Verhindert eine permanente Abhängigkeit.
- Die abstrakte Klasse und die Schnittstelle kann **separat weiter entwickelt** werden.
- Die Implementation kann zur **Laufzeit ausgewechselt** werden.
- Die abstrakte Klasse wird von Änderungen in der Schnittstelle nicht betroffen (Implementation).
- Nützlich bei mehrfachen Schnittstellen.

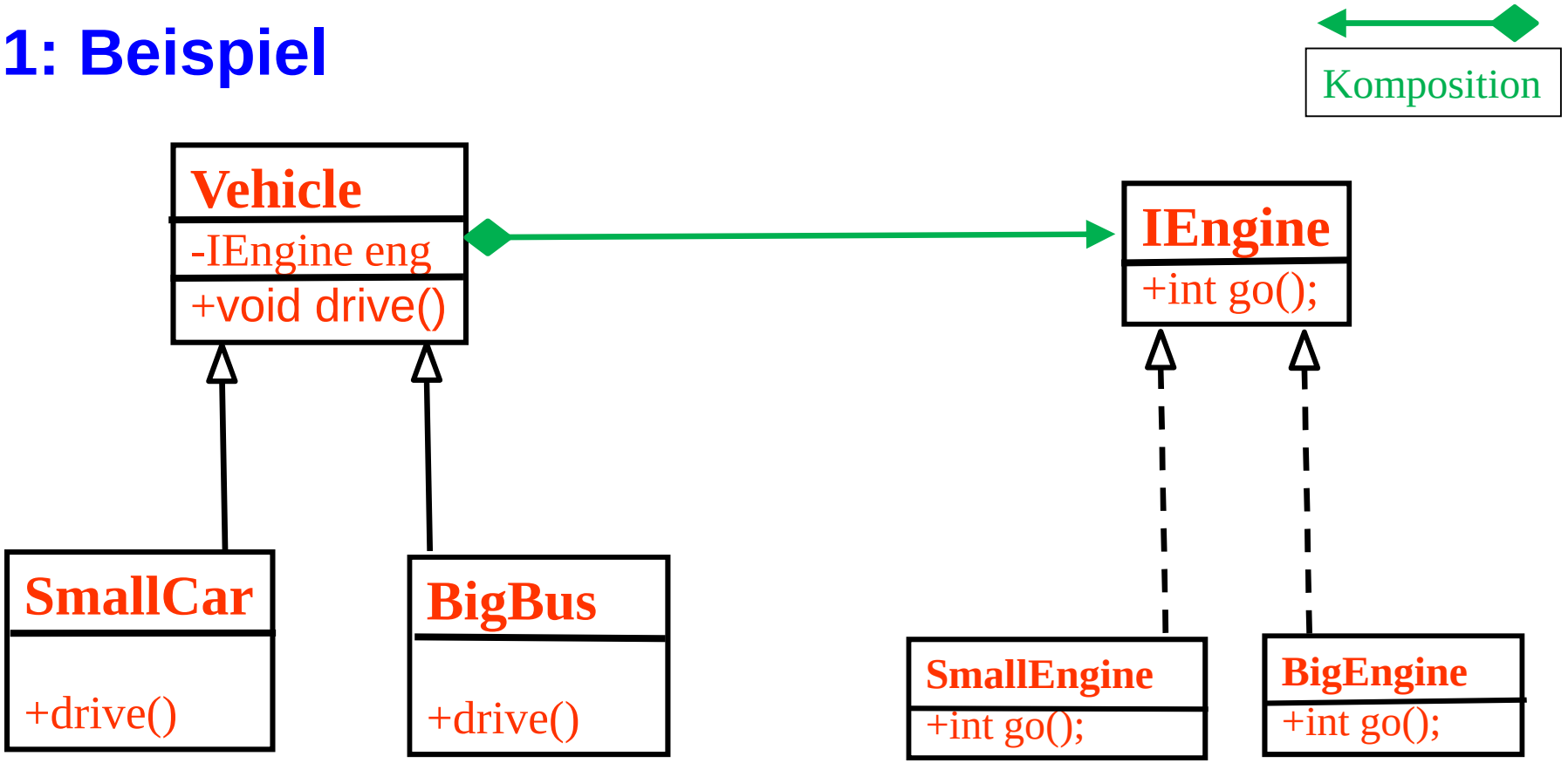
- Das Adapter-Pattern hilft, dass zwei inkompatible Klassen zusammenzufügen.
- Das Bridge-Pattern entkoppelt die **Abstraktion** und die Schnittstelle, indem es zwei hierarchische „Klassen“ erstellt.

Entwurfsmuster: „Bridge“



- Die abstrakte Klasse hat eine Komposition mit dem Interface.
- Die konkreten „Schnittstellen“ implementieren UND erweitern die Schnittstelle.

1: Beispiel



- Das „Interface“ wird per Konstruktor gesetzt.
- Vertauschung ist möglich.
- Abhilfe der Vertauschung: Abstrakte Fabrik (Generator)

1. Beispiel: http://www.avajava.com/tutorials/lessons/bridge-pattern.html?page=2

```
public abstract class Vehicle {
        protected IEngine engine;
        protected int weightInKilos;
        public abstract void drive();
        public void setEngine(IEngine engine) {
                this.engine = engine;
        }
        public void reportOnSpeed(int horsepower) {
                int ratio = weightInKilos / horsepower;
                if (ratio < 3) {
                   syso("Könnte ein Veyron sein.");
                } else if ((ratio >= 3) && (ratio < 8)) {
                   syso("Jaguar, Ferrari oder AMG?");
                } else {
                     syso("Citroen 2CV ?");
                }
        }
}
```

Linke Seite

```
public class SmallCar extends Vehicle {

        public SmallCar(IEngine engine) {
                this.weightInKilos = 600;
                this.engine = engine;
        }

        @Override
        public void drive() {
                System.out.println("\nDas Auto fährt");
                int horsepower = engine.go();
                reportOnSpeed(horsepower);
        }
}
```

Linke Seite

```
public class BigBus extends Vehicle {

        public BigBus(IEngine engine) {
                this.weightInKilos = 14000;
                this.engine = engine;
        }

        @Override
        public void drive() {
                System.out.println("\nDer Bus fährt");
                int horsepower = engine.go();
                reportOnSpeed(horsepower);
        }
}
```

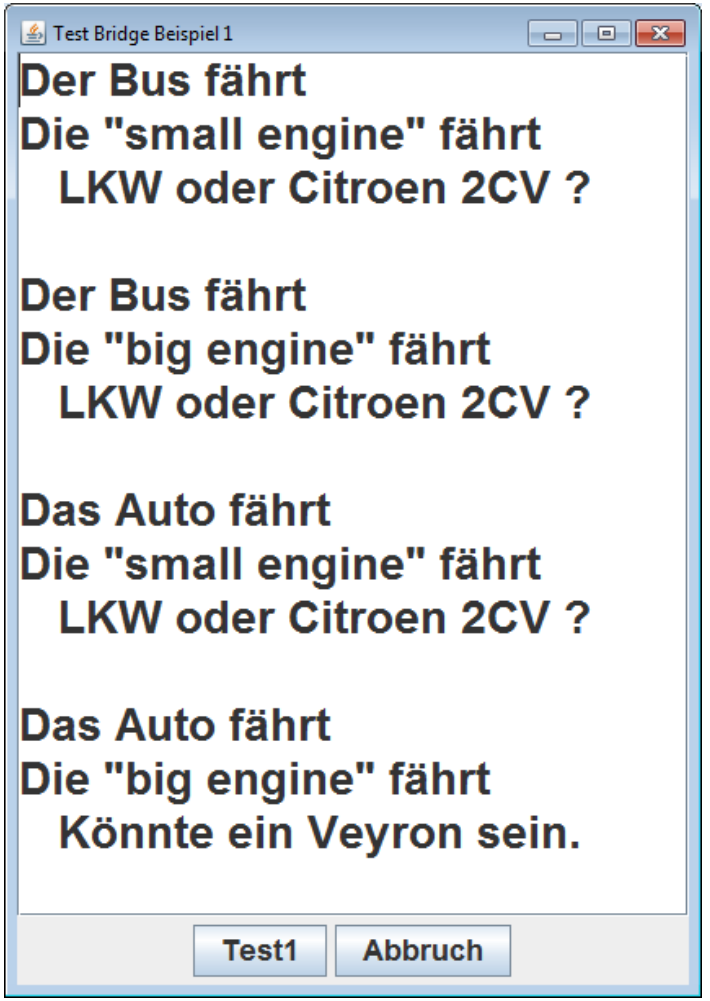
Rechte Seite

```
public interface IEngine {
        public int go();
}
public class BigEngine implements IEngine {
        int horsepower;
        public BigEngine() {
                horsepower = 900;
        }
        @Override
        public int go() {
                System.out.println("\nDie \"big engine\" fährt");
                return horsepower;
        }
}
```

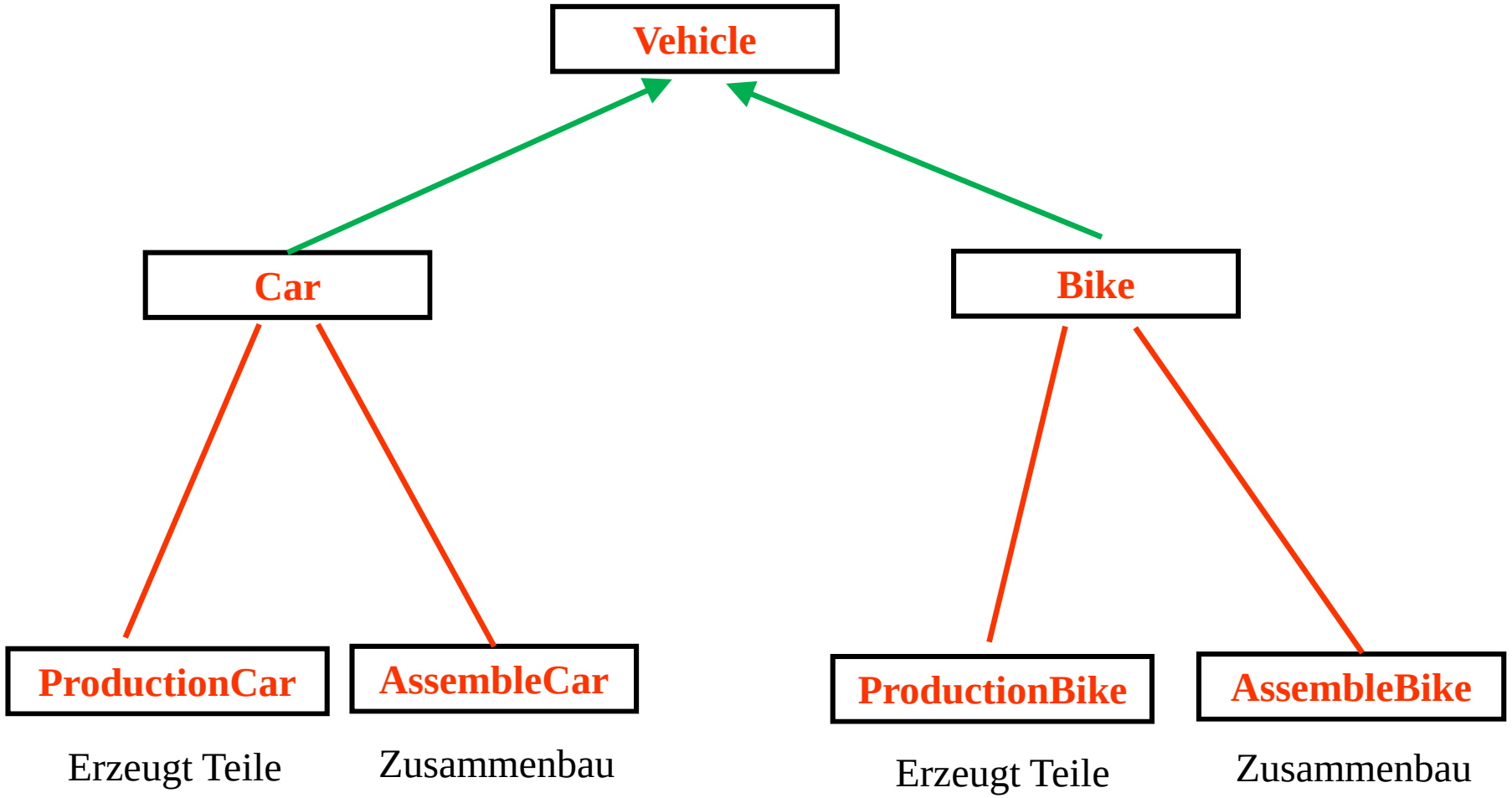
Rechte Seite

```
public interface IEngine {
        public int go();
}
public class SmallEngine implements IEngine {
        int horsepower;
        public SmallEngine() {
                horsepower = 100;
        }
        @Override
        public int go() {
                System.out.println("\nDie \“small engine\" fährt");
                return horsepower;
        }
}
```

```
public static void main(String[] args) {

  Vehicle vehicle = new BigBus(new SmallEngine());
  vehicle.drive();
  vehicle.setEngine(new BigEngine());
  vehicle.drive();

  vehicle = new SmallCar(new SmallEngine());
  vehicle.drive();
  vehicle.setEngine(new BigEngine());
 vehicle.drive();

}
```

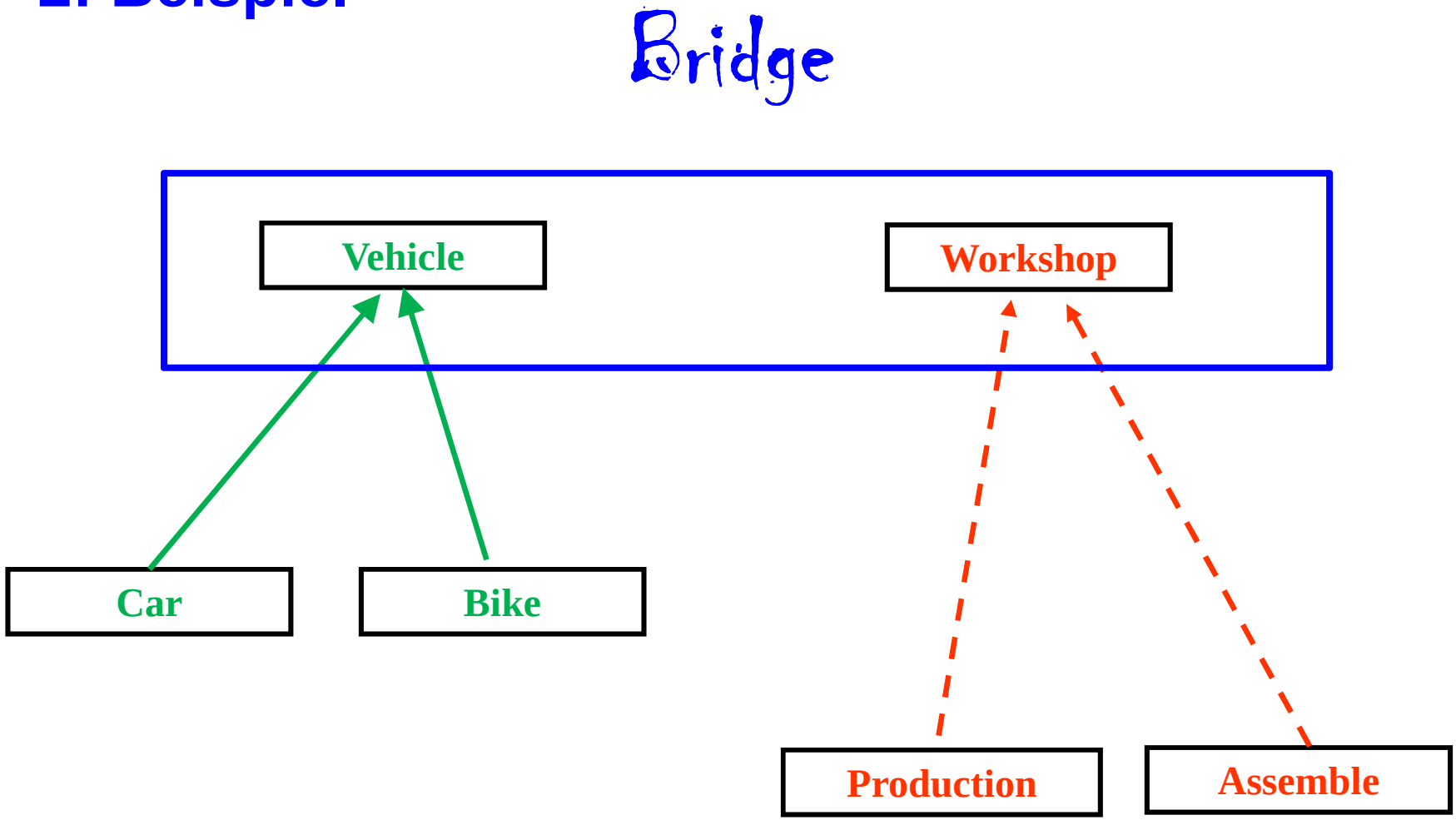


2: Beispiel



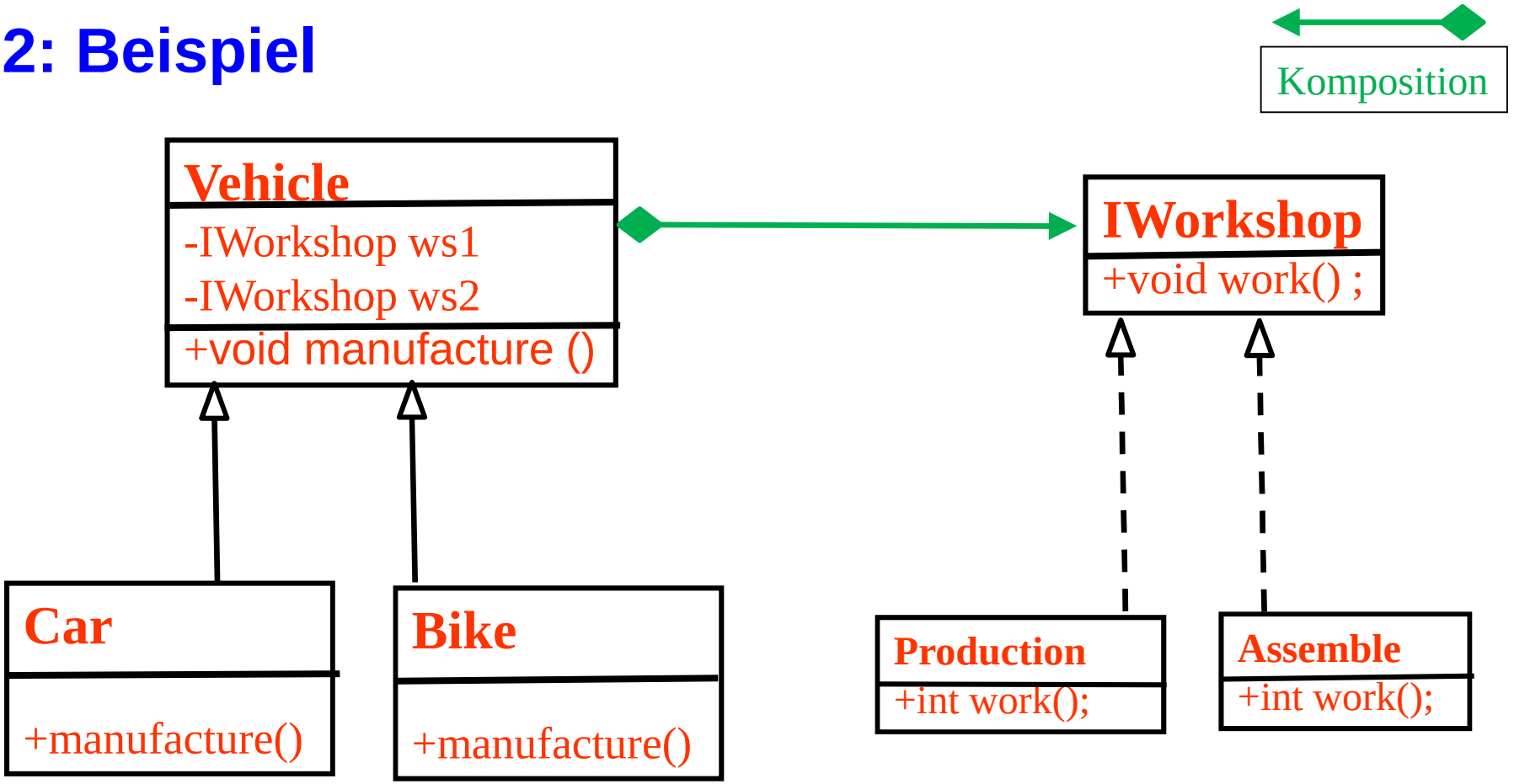
Austausch der Produktion oder Assemble ??

2: Beispiel

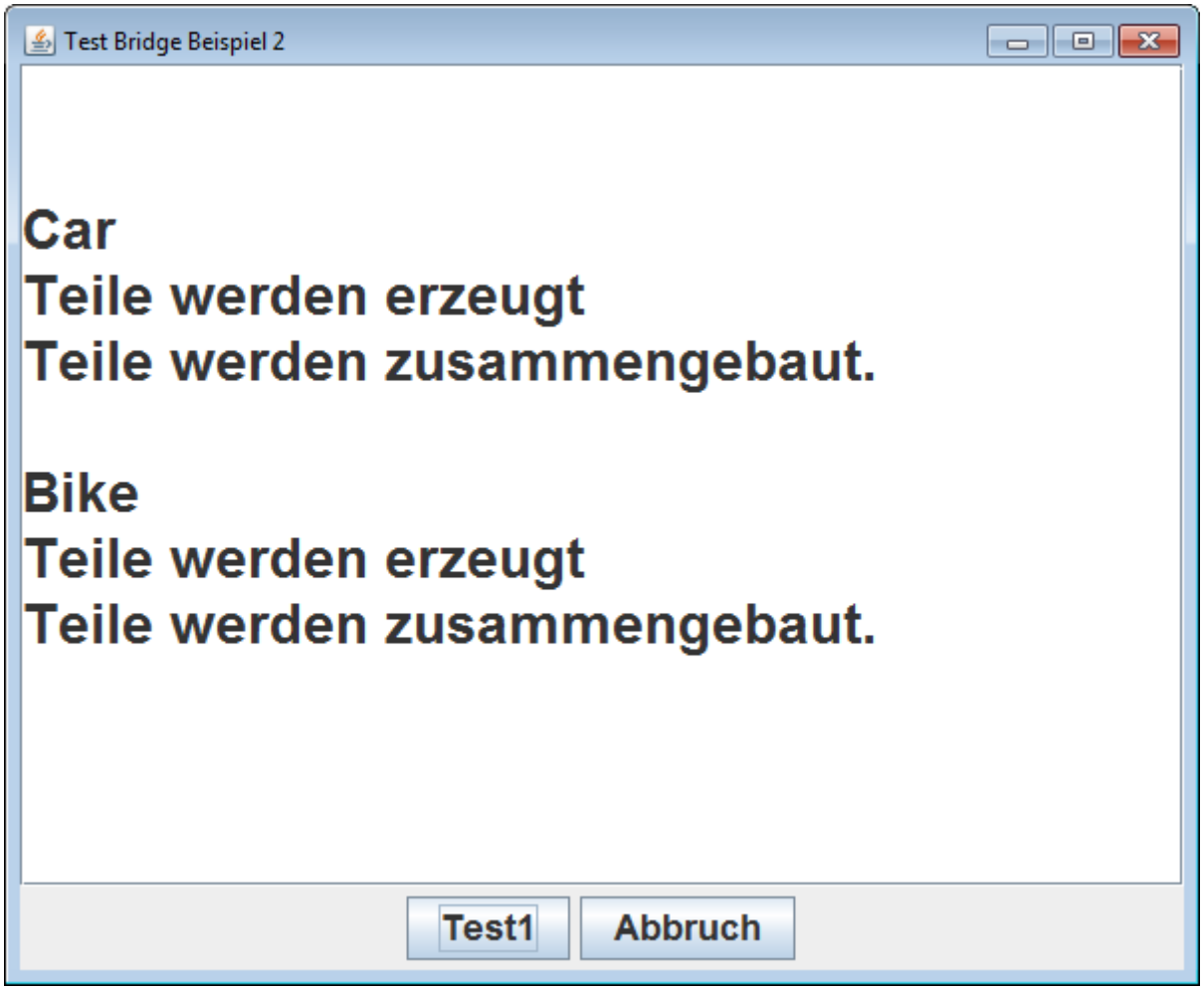


http://javapapers.com/design-patterns/bridge-design-pattern/

2: Beispiel


Komposition
Vehicle
-IWorkshop ws1
-IWorkshop ws2
+void manufacture ()
IWorkshop
+void work() ;
Car
+manufacture()
Bike
+manufacture()
Production
+int work();
Assemble
+int work();

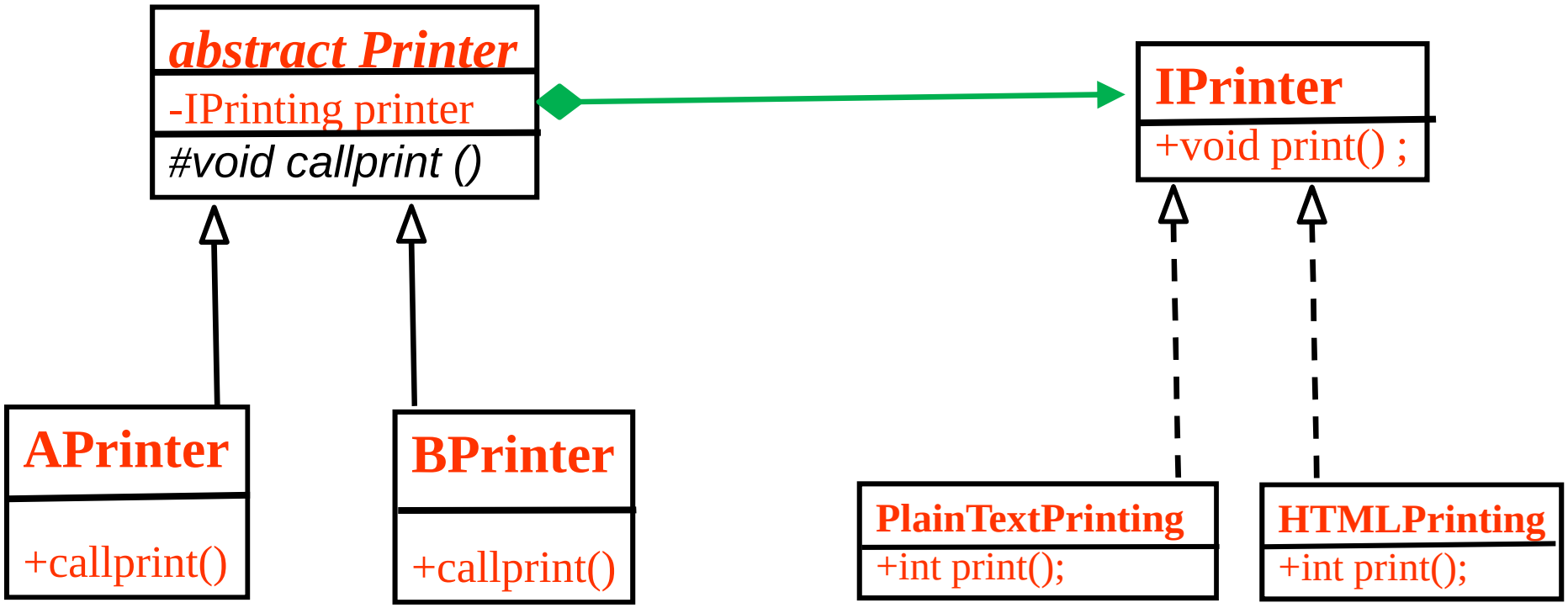
2: Beispiel

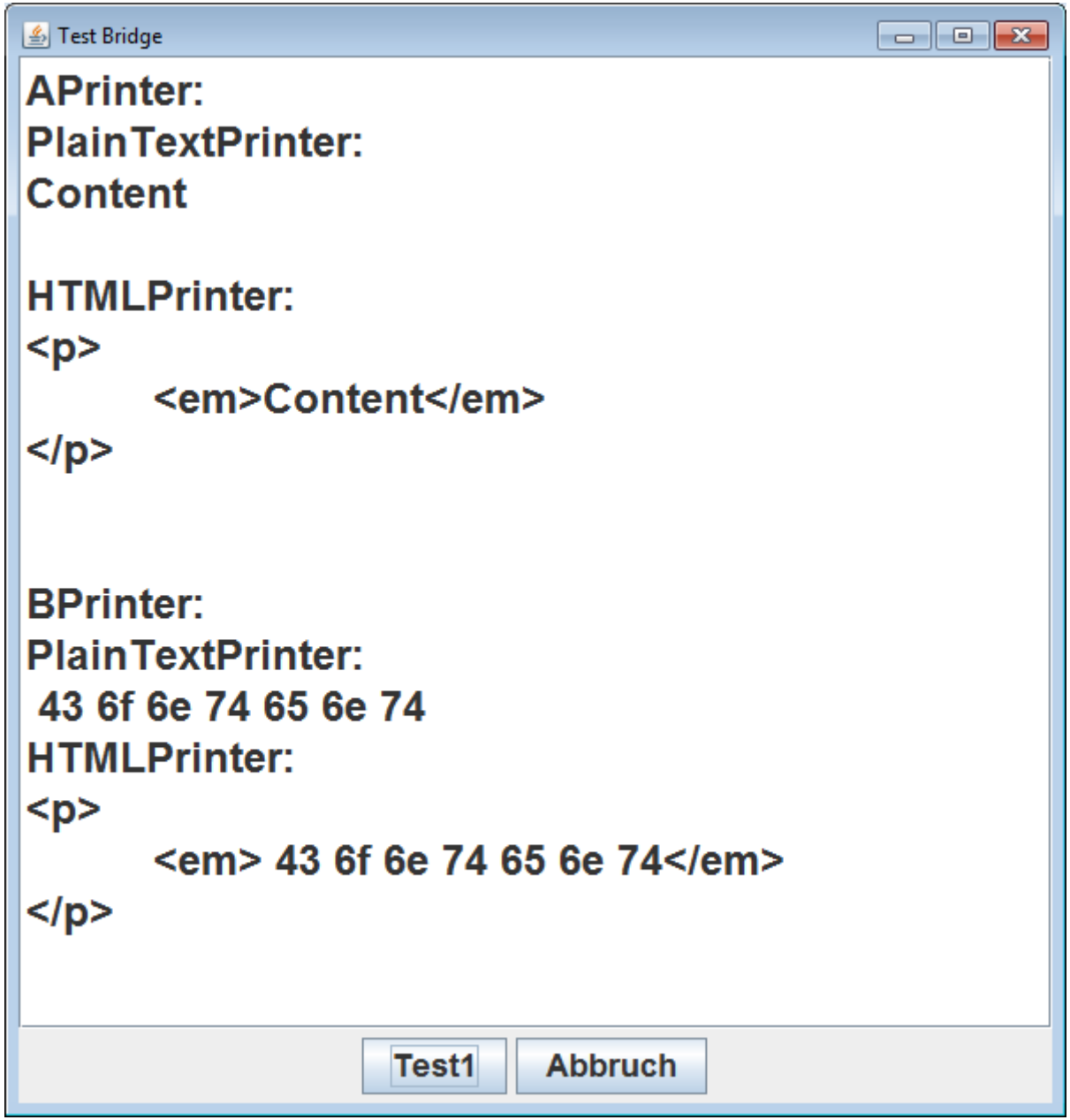

Test Bridge Beispiel 2
Car
Teile werden erzeugt
Teile werden zusammengebaut.
Bike
Teile werden erzeugt
Teile werden zusammengebaut.
Test1
Abbruch

3: Beispiel

 # abstract

Komposition




Test Bridge
APrinter:
PlainTextPrinter:
Content
HTMLPrinter:

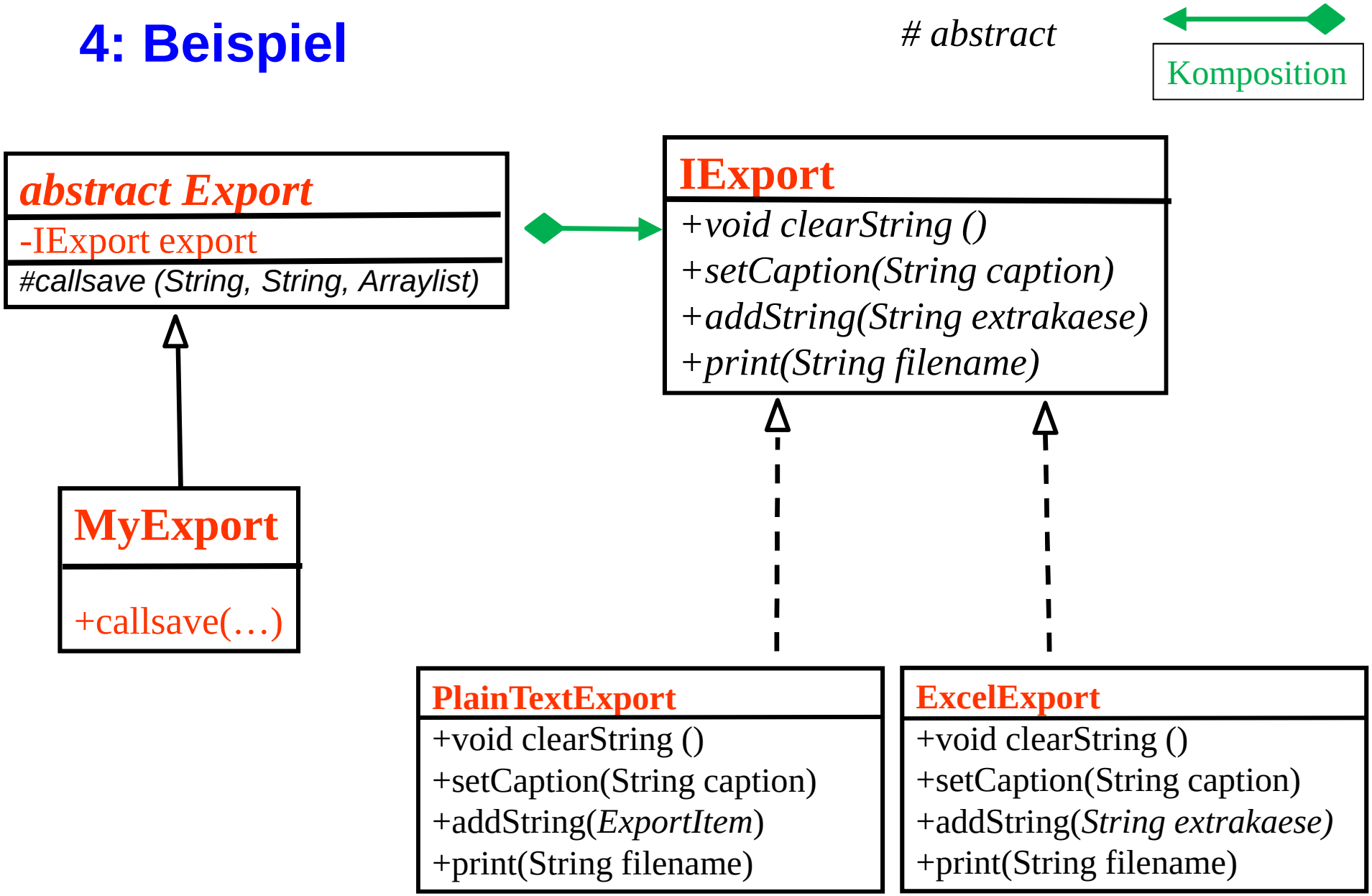
Content

BPrinter:
PlainTextPrinter:
43 6f 6e 74 65 6e 74
HTMLPrinter:

 43 6f 6e 74 65 6e 74

Test1
Abbruch

4: Beispiel


abstract
Komposition
abstract Export
-IExport export
#callsave (String, String, Arraylist)
IExport
+void clearString ()
+setCaption(String caption)
+addString(String extrakaese)
+print(String filename)
MyExport
+callsave(…)
PlainTextExport
+void clearString ()
+setCaption(String caption)
+addString(ExportItem)
+print(String filename)
ExcelExport
+void clearString ()
+setCaption(String caption)
+addString(String extrakaese)
+print(String filename)

```
private void  test1() {
   Export export;  // abstrakte Klasse

   IExport plainExport = new PlainTextExport();  // zwei „Exporte“
   IExport excelExport = new ExcelExport();

   export = new MyExport(plainExport);

    ArrayList<ExportItem> liste = new ArrayList<ExportItem>();
   liste.add( new ExportItem("Hamburg",1234567) );
   liste.add( new ExportItem("Wernigerode",34667) );
   liste.add( new ExportItem("Magdeburg",190567) );

   export.callsave( "einwohner.txt", "Einwohner"  , liste);

   export.setImpl(excelExport);
   export.callsave( "einwohner.xls", "Einwohner"  ,  liste);
}
```