Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/2015

Fliegengewicht (*flyweight*)

Das Fliegengewicht (*flyweight*) ist ein Strukturmustur, dass dazu dient, bei einer Vielzahl von Objekten die Informationen, die vielen Objekten gemeinsam sind, per Referenz auf gemeinsam verwendete Objekte speichersparend zu verwalten.

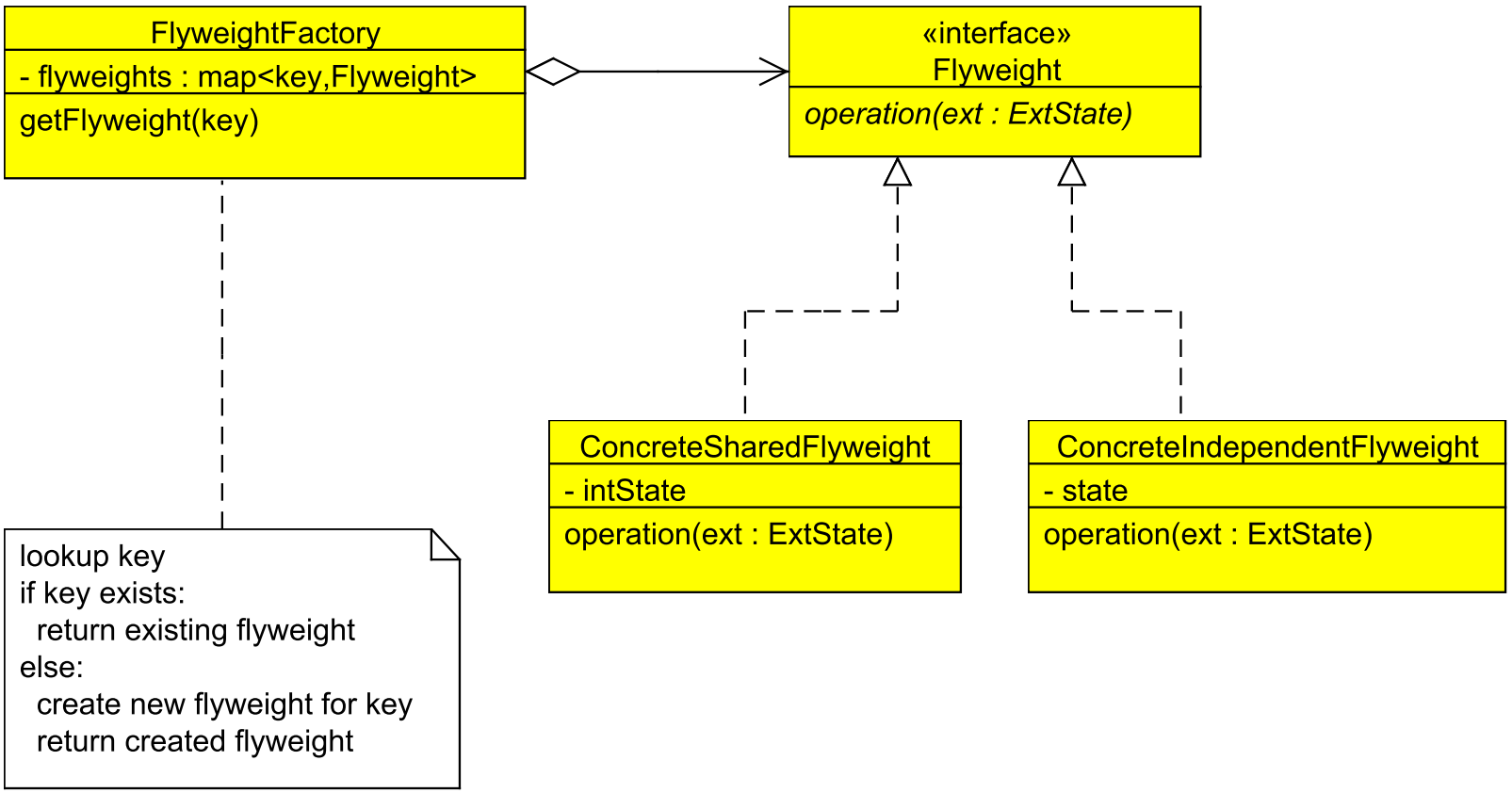
Hierbei wird unterschieden zwischen:

- *intrinsischen* Daten, die individuell je Instanz sind,
- *extrinsischen* Daten, die über ein geteiltes Objekt verwaltet werden und i. A. mehreren Objekten gemeinsam sind.

Beispiele

- Eine Text besteht aus Zeichen, jedem Zeichen entspricht (für eine festgelegte Schriftart) eine jedesmal identische geometrische Kurve. Gespeichert wird in der Praxis die Liste der Character-Codes, die dann bei der Anzeige bzw. beim Ausdruck mit der entsprechenden Kurve ausgegeben wird.
- Eine Firma besteht aus vielen Angestellten in mehreren Abteilungen an mehreren Standorten. In der Personalakte stehen neben den intrinsischen Daten (Name, Personalnr.) jeweils nur die Abkürzung der Abteilung, der ausführliche Name und der Standort der Abteilung sind extrinsische Informationen, die über die Abkürzung eindeutig zugeordnet sind.

Klassendiagramm



Grundlegendes Schema

- Identifizieren: Intrinsische Daten, individuell je Instanz; extrinsische Daten, bei vielen Instanzen gemeinsam.
- Extrinsische Daten sollten weitgehend unveränderlich sein!
- Entwurf des Fliegengewichts.
- FliegengewichtFabrik: Einem Schlüssel eine (ggfs. neu erstellte) Instanz zuordnen.
- Konsequenz: Die Zahl der Instanzen des Fliegengewichts ist durch die Zahl der Schlüssel bestimmt, nicht dadurch, wie oft jeder Schlüssel verwendet wurde.

Beispiel: Ausgabe einer Zeile

Fliegengewicht-Interface:

```
public interface Content {
    public void print(int row, int col);
}
```

Fliegengewicht-Implementierung:

```
public class Glyph implements Content {
    private char c;

    public Glyph(char c) {
        this.c = c;
    }

    @Override
    public void print(int row, int col) {
        System.out.format("Print glyph %c at row %d, col %d\n",
                          c, row, col);
    }
}
```

Beispiel: Ausgabe einer Zeile (Forts.)

Fliegengewicht-Fabrik:

```
public class GlyphFactory {
    Map<Character,Glyph> charmap = new HashMap<Character, Glyph>();

    public Glyph getGlyph(char c) {
        Glyph g = charmap.get(c);
        if (g==null) {
            g = new Glyph(c);
            charmap.put(c, g);
        }
        return g;
    }

    public int size() {
        return charmap.size();
    }
}
```

Beispiel: Ausgabe einer Zeile (Forts.)

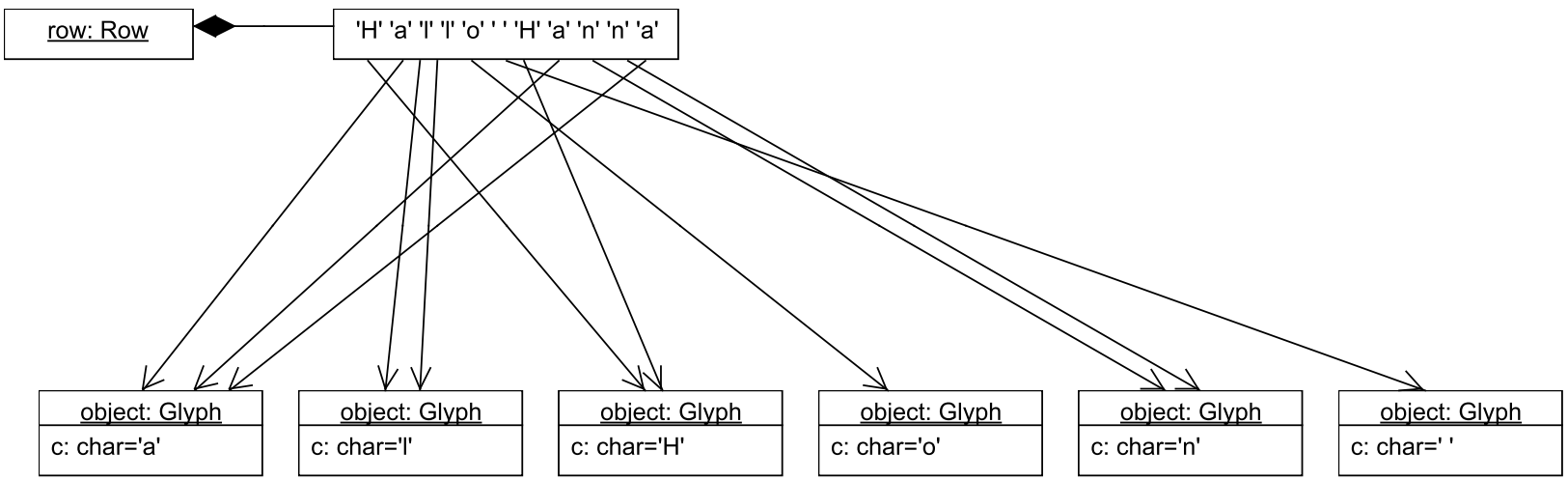
```
public class Row implements Content {
    GlyphFactory f = new GlyphFactory();
    List<Glyph> glyphs = new ArrayList<Glyph>();

    public void add(char c) {
        glyphs.add(f.getGlyph(c));
    }

    public void add(String s) {
        for (char c : s.toCharArray())
            add(c);
    }

    @Override
    public void print(int row, int col) {
        int offset = 0;
        for (Glyph g : glyphs) {
            g.print(row, col+offset++);
        }
    }
}
```

Schaubild


row: Row
'H' 'a' 'l' 'l' 'o' ' ' 'H' 'a' 'n' 'n' 'a'
object: Glyph
c: char='a'
object: Glyph
c: char='l'
object: Glyph
c: char='H'
object: Glyph
c: char='o'
object: Glyph
c: char='n'
object: Glyph
c: char=' '

Analyse

Der Quelltext lässt eine enge Verwandschaft zum Prototypmuster erkennen.

Während der Prototyp jedoch dem Erzeugen (anhand des *Kopierens* eine Vorlage) dient und daher bei jedem neuen Aufruf der Fabrikmethode des Prototypmusters eine neue Instanz entsteht, reduziert das Fliegengewicht durch das Teilen der extrinsischen Daten die Zahl der Instanzen.

Im folgenden Beispiel wird von Javas Möglichkeit Gebrauch gemacht, für ein Mapping eine `enum` zu verwenden, wobei hier die Fliegengewichtsinstanzen schon direkt zu Beginn erzeugt werden.

Beispiel: Chemische Formel

Mapping Elementkürzel–Chemische Eigenschaften

```
public enum Element {
    H(1.008, 1),
    C(12.0, 6),
    N(14.003,7),
    O(15.995, 8),
    Na(22.990,11),
    Cl(34.970,17);

    private final double gewicht;
    private final int ladung;
    private Element(double gewicht, int ladung) {
        this.gewicht = gewicht;
        this.ladung = ladung;
    }
    public double getGewicht() {
        return gewicht;
    }
    public int getLadung() {
        return ladung;
    }
}
```

Beispiel: Chemische Formel (Forts.)

Interface

```
public interface Chemical {
    public String getName();
    public int getLadung();
    public double getGewicht();
    public int getAnzahl();
}
```

Fliegengewicht:

Atomzahl intrinsisch, Elementeigenschaften extrinsisch

```
public class ChemicalImpl implements Chemical {
    private int anzahl;
    private Element element;

    public ChemicalImpl(Element element, int anzahl) {
        this.anzahl = anzahl;
        this.element = element;
    }
```

Beispiel: Chemische Formel (Forts.)

```
    @Override
    public String getName() {
        if (getAnzahl()==1)
            return element.toString();
        else
            return element.toString()+getAnzahl();
    }

    @Override
    public int getLadung() {
        return anzahl*element.getLadung();
    }

    @Override
    public double getGewicht() {
        return anzahl*element.getGewicht();
    }

    @Override
    public int getAnzahl() {
        return anzahl;
    }
}
```

Beispiel: Chemische Formel (Forts.)

Eine Chemische Summenformel:

```
public class Structure implements Item {
    List<Item> items = new ArrayList<Item>();

    public void add(String name, int anzahl) {
        items.add(new ItemImpl(Element.valueOf(name), anzahl));
    }

    @Override
    public String getName() {
        StringWriter out = new StringWriter();
        for (Item item : items) {
            out.write(item.getName());
        }
        return out.toString();
    }
```

Beispiel: Chemische Formel (Forts.)

```
    @Override
    public int getLadung() {
        int summe = 0;
        for (Item item : items)
            summe += item.getLadung()*item.getAnzahl();
        return summe;
    }
    @Override
    public double getGewicht() {
        double summe = 0.0;
        for (Item item : items)
            summe += item.getGewicht()*item.getAnzahl();
        return summe;
    }
    @Override
    public int getAnzahl() {
        int summe = 0;
        for (Item item : items)
            summe += item.getAnzahl();
        return summe;
    }
}
```

Stellvertreter (*proxy*)

Das Stellvertretermuster (*proxy pattern*) verwendet zur Implementierung eines Interfaces *zwei* Klassen:

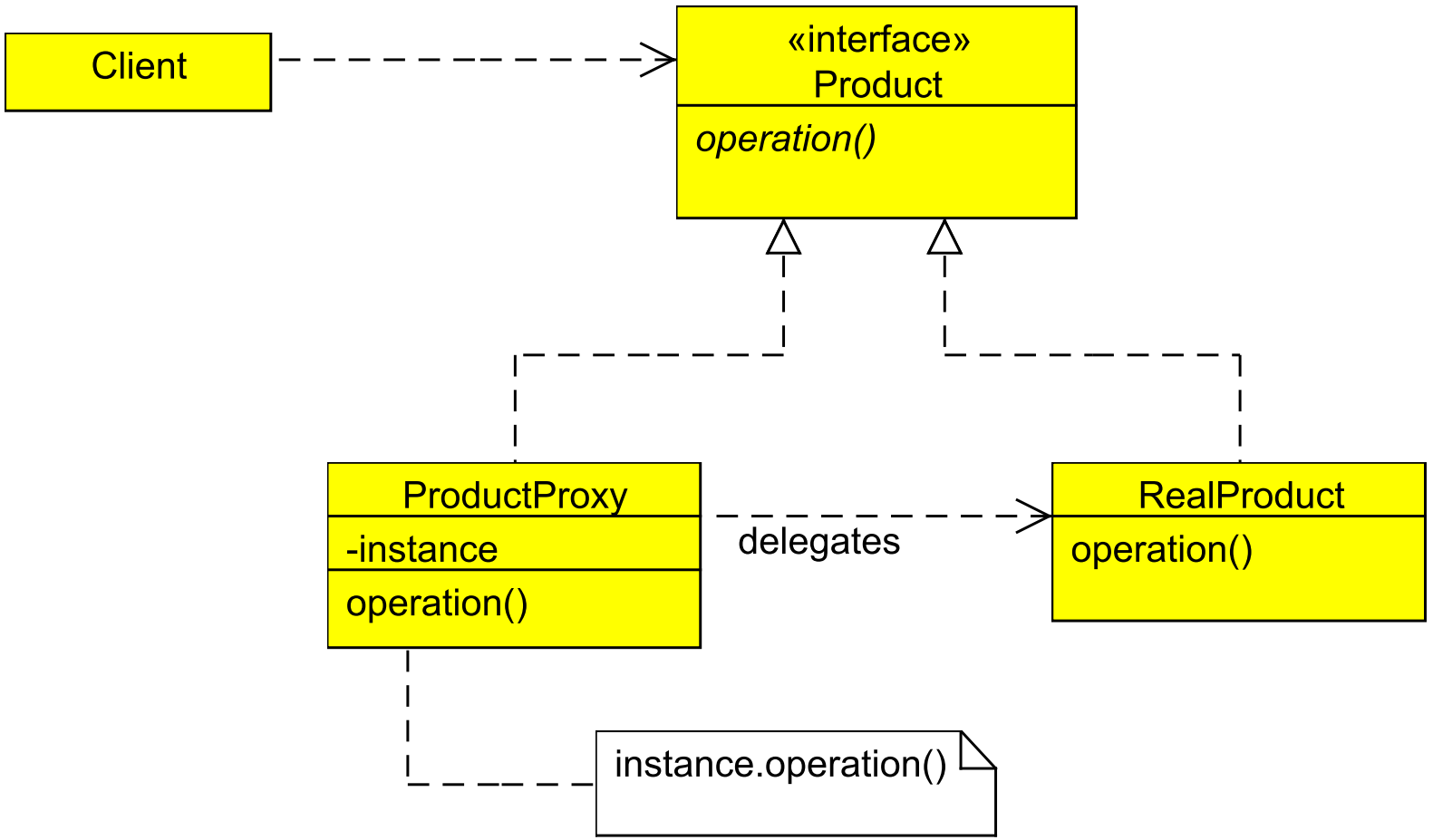
- neben der realen Klasse, die die im Interface definierte Operation ausführt,
- eine Stellvertreterklasse, die das Interface in der Weise implementiert, dass beim Aufruf der Operation diese an eine Instanz der realen Klasse delegiert wird.

Der Stellvertreter ist also eine Spezialisierung des Dekorierers, in dem Sinne, dass die delegierte Operation im Normalfall weder abgeändert noch durch zusätzliche Operationen ergänzt wird.

Anwendungsfälle

- Eine Instanz eines komplexen Objektes soll Klienten gegenüber durch eine Vielzahl von Instanzen der Stellvertreterklasse gegenüber repräsentiert werden, hier wird denn das Proxy-Pattern mit dem Flyweight-Pattern kombiniert.
- Die Instanz der realen Klasse läuft auf einem anderen Rechner, das Proxy-Objekt erledigt den Datenaustausch mit dem realen Objekt über eine Schnittstelle (z. B. CORBA, .NET, SOAP).
- Das Erzeugen des realen Objektes ist aufwändig, die Stellvertreterklasse erledigt somit das Erzeugen bei Bedarf und zwischenspeichert ab hier das Objekt. Dies wird als *virtual proxy* bezeichnet. Prinzip: *lazy loading*.

Klassendiagramm



Beispiel

Interface:

```
public interface Content {
    public String getContent();
}
```

Proxy-Klasse:

```
public class FileContentProxy implements Content {
    private String filename;
    private Content content;

    public FileContentProxy(String filename) {
        this.filename = filename;
    }

    @Override
    public String getContent() {
        if (content == null) {
            content = new FileContent(filename);
        }
        return content.getContent();
    }
}
```

Beispiel (Forts.)

Reale Klasse:

```
public class FileContent implements Content {
    public final int BUFSIZE = 1024;
    private String content;

    public FileContent(String filename) {
        Reader reader = null;
        Writer writer = null;
        try {
            char[] buf = new char[BUFSIZE];
            reader = new FileReader(filename);
            writer = new StringWriter();
            while (true) {
                int count = reader.read(buf);
                if (count < 0) break;
                writer.write(buf, 0, count);
            }
            content = writer.toString();
        } catch (IOException e) {
            // do nothing
```

Beispiel (Forts.)

```
        } finally {
            try {
                reader.close();
                writer.close();
            } catch (IOException e) {
            }
        }
    }

    @Override
    public String getContent() {
        return content;
    }
}
```

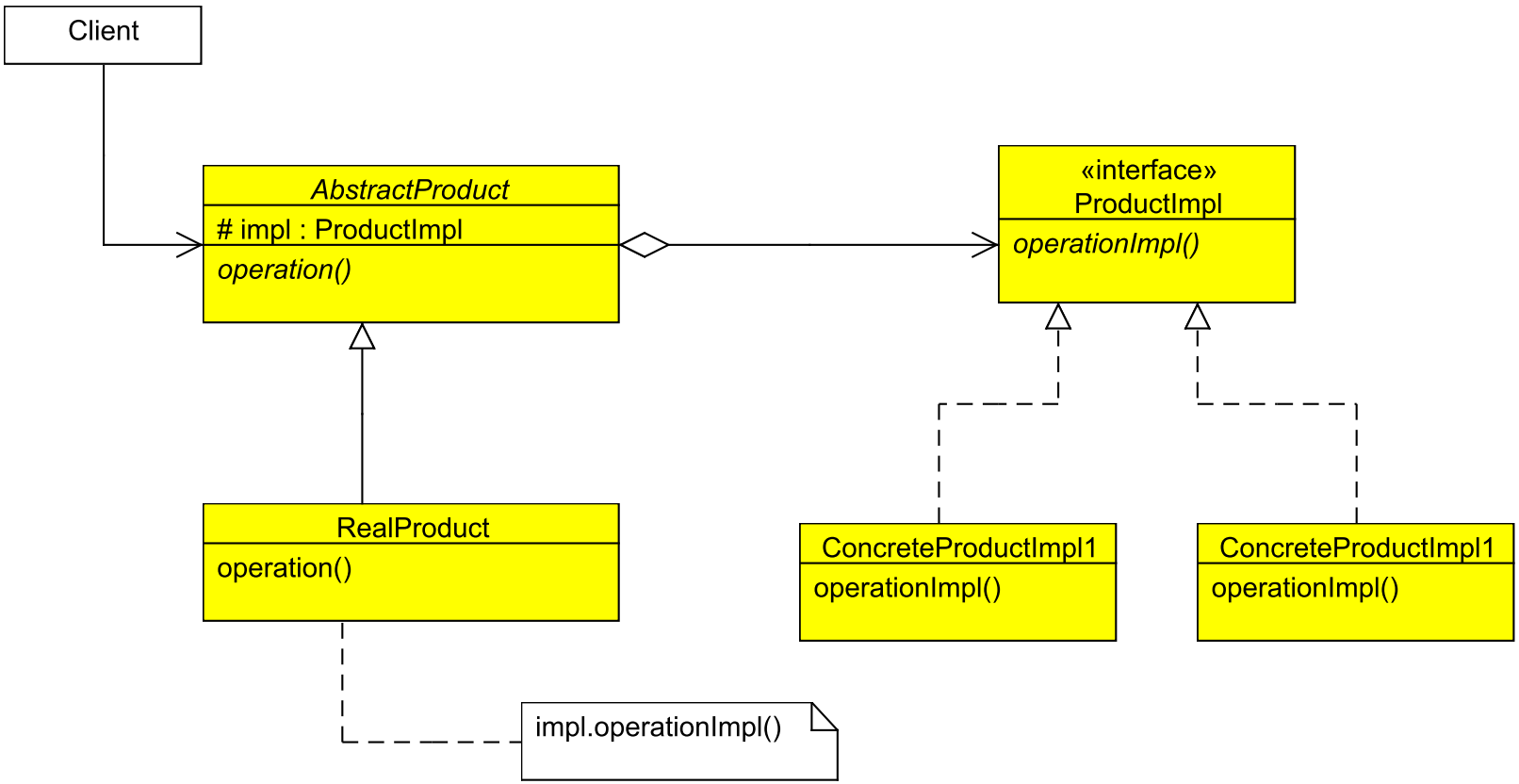
Das Brückenmuster (*bridge*)

Bisher wurde in den vorgestellten Strukturmustern erreicht, dass *ein* Interface durch verschiedene Implementierungen realisiert werden konnte.

Bei der Brücke soll nun für mehrere Spezialisierungen einer Abstraktion die in der Abstraktion definierte Funktionalität auf unterschiedlichen Wegen ausführen.

Beispiel: Kreis und Rechteck sind Spezialisierungen eines abstrakten Geom. Objekts, die mithilfe der Methode `display()` entweder auf dem Monitor oder dem Drucker ausgegeben werden können.

Klassendiagramm



Beispiel

Abstraktes geometrisches Objekt:

```
public abstract class GeoObject {
    protected Presentation presentation;

    public GeoObject(Presentation presentation) {
        this.presentation = presentation;
    }

    public void setPresentation(Presentation presentation) {
        this.presentation = presentation;
    }

    public abstract void display();
}
```

Beispiel (Forts.)

Implementierung: Kreis

```
public class Circle extends GeoObject {
    double x, y, radius;

    public Circle(Presentation presentation,
            double x, double y, double radius) {
        super(presentation);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    @Override
    public void display() {
        presentation.displayCircle(x, y, radius);
    }
}
```

Beispiel (Forts.)

Implementierung: Rechteck

```
public class Rectangle extends GeoObject {
    // lower left x/y, upper right x/y
    double llx, lly, urx, ury;

    public Rectangle(Presentation presentation,
            double llx, double lly,
            double urx, double ury) {
        super(presentation);
        this.llx = llx;
        this.lly = lly;
        this.urx = urx;
        this.ury = ury;
    }

    @Override
    public void display() {
        presentation.displayRectangle(llx, lly, urx, ury);
    }
}
```

Die Präsentationsebene

Interface:

```
public interface Presentation {
    public void displayRectangle(double llx, double lly,
                                 double urx, double ury);
    public void displayCircle(double x, double y, double radius);
}
```

Implementierung:

```
public class Monitor implements Presentation {
    @Override
    public void displayRectangle(double llx, double lly,
                                 double urx, double ury) {
        // draw the rectangle on screen
    }

    @Override
    public void displayCircle(double x, double y, double radius) {
        // draw the circle on screen
    }
}
```

Anwendung

```
Presentation screen = new Monitor();
Presentation printer = new Printer();
GeoObject c = new Circle(new Printer(), 0., 0., 2.5);
GeoObject r = new Rectangle(screen, 0., 0., 1., 2.);
c.display();
r.display();
r.setPresentation(printer);
r.display();
```

Diskussion der Strukturmuster

Bei den Strukturmustern werden über die Komposition von Klassen und/oder Objekten größere, übergeordnete Strukturen geschaffen:

Kompositum, Dekorierer und Proxy haben sehr ähnliche Klassendiagramme. Das Kompositum dient dem Aufbau einer rekursiven Baumstruktur und sorgt so dafür, dass eine beliebig große Menge von Objekten wie ein großes Objekt erscheint. Auch der Dekorierer ist rekursiv, hat aber eine andere Zielsetzung. Er dient nicht der Strukturierung einer Menge von Objekten, sondern der Erweiterung oder Änderung von Funktionalität und erspart das Definieren von Unterklassen.

Diskussion der Strukturmuster (Forts.)

Der Proxy ist ein Stellvertreter für eine konkrete Klasse und unterscheidet sich vom Dekorierer in der Form dadurch, dass er nicht rekursiv ist, in der Zielsetzung dadurch, dass er der Indirektion dient, wenn es nicht erwünscht ist, dass auf Instanzen einer konkreten Klasse direkt zugegriffen wird.

Adapter und Brücke führen beide eine Indirektion auf ein anderes Objekt ein. Der Adapter dient dabei dazu, die Zusammenarbeit von zwei existierenden Schnittstellen zu ermöglichen. Daher tauchen diese in der Software-Entwicklung eher am Ende von Projekten auf. Die Brücke ermöglicht es hingegen, eine Teilfunktionaltität auszulagern und es zu Ermöglichen, verschiedene Implementationen austauschbar zu machen.