Inhalt – Vorlesungsteil Entwurfsmuster

1 Einführung

1.1 Definition

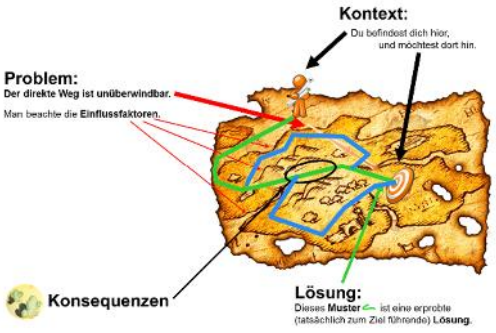
Entwurfsmuster sind erprobte Lösungsansätze für wiederkehrende Fragestellungen und Aufgaben: „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice“ (Alexander, 1977). Es handelt sich also um generative Problem-Lösungs-Paare, die vom Einzelfall abstrahieren und die invarianten Teile einer strukturierten Lösung berücksichtigen.

1.2 Einsatzgebiete

Der Muster-Ansatz wurde von Christopher Alexander entwickelt und stammt ursprünglich aus der Architekturtheorie und soll Individuen dabei helfen, Anforderungen und Bedürfnisse für ihren eigenen Wohn- und Lebensraum zu identifizieren und kommunizieren. Insgesamt werden in dem Buch „A Pattern Language“ (Alexander, 1977) 253 Muster zur Gestaltung von Städten, Gebäuden und Konstruktionen beschrieben. Beck & Cunningham (1987) greifen den Ansatz auf und übertragen ihn auf den Bereich der objektorientierten Softwareentwicklung. Der Durchbruch gelingt 1995 als Gamma et al. mit „Design Patterns. Elements of Reusable Object-Oriented Software“ ein Standardwerk über objektorientierte Softwareentwicklung publizieren und Cunningham das erste Wiki programmiert, um darin kooperativ Entwurfsmuster zu sammeln. Inzwischen gibt es viele Disziplinen, die den Entwurfsmusteransatz aufgegriffen haben, darunter Mensch-Maschine-Interaktion, Webdesign, Communitygestaltung, Organisationsformen, Geschäftspraktiken, nachhaltige Entwicklung, Pädagogik und E-Learning. Im Folgenden werden sich unsere Betrachtungen aber auf Entwurfsmuster im Bereich Softwaredesign konzentrieren.

1.3 Kontext, Problemfeld, Lösung, Konsequenzen

Ein Entwurfsmuster beschreibt eine mehrfach erprobte Lösung in allgemeiner Form, so dass diese für neue Einsatzkontexte angepasst werden kann. Auf den ersten Blick erscheint vor allem die Lösung das Wichtigste an einem Muster zu sein. Tatsächlich verfolgt der Musteransatz aber eine ganzheitliche Sichtweise, bei der die Lösung auch wirklich zu dem jeweiligen Problemfeld, der Zielsetzung und dem Aufgabenfeld passen muss. Eine bestimmte Struktur, also z.B. ein Teil einer Softwarearchitektur, ist nicht automatisch eine gute Lösung, sondern sie muss der Designaufgabe gerecht werden. Dazu gehört auch, dass man mit den Konsequenzen einer Lösung leben kann.



Man kann sich dies gut an einer Wegmetapher vorstellen. Wenn Sie ein Ziel erreichen möchten, dann kann es mehrere Wege geben, die dort hinführen. Jeder Weg hat aber seine eigenen Vor- und Nachteile: ein Weg ist vielleicht schnell, aber langweilig, während der andere Weg schöner aber anstrengender ist und vielleicht besondere Ausrüstung benötigt. In beiden Fällen führt der Weg zum Ziel, aber Sie müssen je nach Situation entscheiden, welcher Weg der bessere ist. Sie müssen also den Kontext im Auge behalten. Kleine Änderungen können auch dazu führen, dass ein Weg nicht mehr passt: wenn eine Passage zugeschneit ist, dann ist vielleicht ein Weg blockiert. Genauso verhält es sich mit Entwurfsmustern im Bereich des Software-Designs. Eine zusätzliche Anforderung (z.B. häufiger

Wechsel zwischen Plattformen) kann dazu führen, dass ein Entwurfsmuster, das sonst sehr gut passt, nicht mehr genutzt werden kann, weil es diese eine Anforderung nicht erfüllt. Dies ist auch mit Ganzheitlichkeit gemeint: ein Entwurfsmuster soll allen Einflussfaktoren der Situation gerecht werden.

1.4 Anpassbare allgemeine Lösungen

Nicht immer führt eine kleine Änderung des Kontexts dazu, dass gleich die gesamte Lösung nicht mehr funktioniert. Wenn ein Baum auf den Weg gefallen ist, dann lässt sich dieser oft leicht umgehen, d.h. Sie ändern den Lösungsweg minimal ab. Genauso ist es mit Software-Mustern. Wir werden etwas später allgemeine Klassenstrukturen kennenlernen, die eine gute Lösung beschreiben. Ein häufiger Fehler ist jedoch, dass man diese Klassenstrukturen zu eng interpretiert und stumpf umsetzt. Tatsächlich ist es erlaubt und empfehlenswert, die Strukturen zu variieren, so dass sie zur eigentlichen Aufgabe passen.

1.5 Begründete Lösungen

Und noch ein letztes Mal möchte ich die Wegmetapher bemühen. Wege haben immer eine allgemeine Strukturform (gut sichtbar als Streckenzug auf einer Landkarte). Dass diese Form eine gute Lösung darstellt ist nicht zufällig. Im Prinzip könnten Sie unendlich viele Streckenzüge auf einer Karte einzeichnen. Die meisten sind aber keine oder keine guten Lösungen. Wenn eine Strecke Sie zu einer unüberwindbaren Schlucht führt dann ist diese sicherlich keine Lösung mehr. Wenn eine Strecke Sie besonders umständlich und über gefährliche Pfade zum Ziel führt dann ist dies sicherlich auch keine gute Lösung. Die Streckenform, die Sie möglichst elegant und sicher zum Ziel führt ist dagegen eine Musterlösung. Und diese Form liegt gerade in den verschiedenen Gegebenheiten des Kontexts begründet: eine unüberwindbare Schlucht oder Felsmauer ändert die Ausprägung der Route. Wenn eine Form alle diese Einflussfaktoren berücksichtig, dann stellte sie eine gute Lösung da. Es muss dabei nicht die einzige gute oder gar die beste Lösung sein (oft ist der beste Weg noch nicht gefunden), aber es ist zumindest keine schlechte Lösung.

1.6 Software Patterns

All dies lässt sich in gleicher Weise auch für Entwurfsmuster im Bereich der Softwareentwicklung sagen (die Wegmetapher veranschaulicht nur die Bedeutung). Eine Lösungsstruktur besteht aus einer bestimmten Konfiguration von Klassen, sowie der dynamischen Struktur der Objektinteraktion zur Laufzeit. Häufig wird in der Lösungsbeschreibung ausführlich auf die einzelnen Rollen, Verantwortlichkeiten und Teilnehmer (=Klassen oder Objekte) eingegangen, es werden Stolpersteine aufgezeigt, und Beispielimplementierungen gezeigt. Dass es sich bei der beschriebenen Lösung aber nicht zufällig um eine gute Struktur handelt liegt – ähnlich wie bei der Wegmetapher – an den Gegebenheiten des Problemfelds und Anwendungskontexts. Um zu verstehen, ob ein Entwurfsmuster zu Ihrer Aufgabe passt müssen Sie dieses Problemfeld und auch die Konsequenzen der gewählten Lösung ebenfalls verstehen. Waren es beim Wandern die Steine, Abhänge, Schluchten und Felswände, die den Weg geprägt haben, so sind es in der Softwareentwicklung Faktoren bzw. Anforderungen wie Kapselung, Granularität, minimale Abhängigkeiten, Flexibilität, Performanz, Wiederverwendbarkeit, Wartbarkeit, Sicherheit, Transparenz, Robustheit, Reduzierung der Komplexität, Austauschbarkeit von Komponenten, Systemkompatibilität, dynamische Konfigurierbarkeit, Fähigkeiten der Entwickler.

Nicht immer spielen in jedem Kontext alle Anforderungen eine Rolle: nicht in jeder Landschaft gibt es tiefe Schluchten und nicht in jedem Softwareprojekt ist Performanz das wichtigste Ziel. Um zu wissen, was Sie erwartet, sollten Sie aber genau wissen, wie Ihre Umgebung aussieht. Vor einer großen Wanderung machen Sie sich auch nicht nur mit den Wegen sondern auch mit der Umgebung vertraut. In gleicher Weise sollten Sie verstehen, in welchem Umfeld ein Software-Entwurfsmuster eingesetzt werden kann.

1.7 Passung zwischen Anwendungskontext, Problemfeld und Lösung

Ich hoffe, dass inzwischen klar geworden ist, dass es bei Entwurfsmuster nicht nur um den Lösungsansatz geht, sondern auch um den Anwendungskontext, das Problemfeld (Problem und „Forces“) und die Konsequenzen einer Lösung. Wenn Sie Beschreibungen von Entwurfsmustern lesen, dann erfahren Sie nicht nur etwas über potentielle Lösungswege, sondern Sie werden auch auf ernste Probleme aufmerksam gemacht. Das gemeine an schweren Problem ist, dass diese sich erst spät in Softwareprojekten zeigen. Denken Sie zum Beispiel daran, wie kompliziert es werden kann wenn Sie tausende Unterklassen bilden müssen („Klassenexplosion“). Das ist ein echtes Problem, aber es zeigt sich nicht wenn Sie die ersten fünf oder zehn Unterklassen ableiten, sondern erste bei zunehmender Komplexität. Um bei der Wegmetapher zu bleiben: Sie laufen glücklich in die richtige Himmelsrichtung los, aber erst viel später merken Sie, dass Sie vor einer Felswand stehen. Hätten Sie die Umgebung besser gekannt, wären Sie vielleicht zunächst einen kleinen Umweg gegangen, aber würden dann nicht auf unüberwindbare Hindernisse stoßen. Auch in Softwareprojekten bedeuten Entwurfsmuster, dass man zu Anfang oft etwas mehr Aufwand betreiben muss (etwa zusätzliche Interfaces einführt), sich dies am Ende aber vielfach auszahlt.

1.8 Beschreibungen von Entwurfsmustern

Wenn Sie sich die Beschreibungen von Entwurfsmustern anschauen, werden Sie immer den Anwendungskontext, das Kernproblem bzw. Problemfeld („Forces“), eine allgemeine Lösung, Konsequenzen der Lösung und Beispiele in der Beschreibung finden. Allerdings verwenden verschiedene Bücher über Entwurfsmuster unterschiedliche Überschriften für diese Perspektiven und untergliedern diese häufig noch weiter.

Einen wichtigen Aspekt von Entwurfsmustern habe ich nun aber noch nicht genannt – die Namensgebung. Entwurfsmuster werden immer mit Namen belegt, die möglichst in das Vokabular des Softwaredesigners einfließen sollen, so dass in der Kommunikation eine komplexe Struktur mit nur einem Wort referenziert werden kann. Im Prinzip ist dies etwas ganz normales, das in jeder natürlichen Sprache geschieht. Statt „ein großes Säugetier mit langer Nase, großen Ohren und dicker, grauer Haut, das im Urwald lebt“ sprechen wir vom „Elefant“. Das Wort „Elefant“ bezeichnet eine sehr komplexe Struktur, nämlich die Gestalt des Elefanten in seiner Gesamtheit. In gleicher Weise bezeichnen Begriffe wie „Strategie“, „Dekorierer“ oder „Abstrakte Fabrik“ komplexe Lösungen im Rahmen der objektorientierten Softwareentwicklung. Und ähnlich wie man den Elefanten in beliebiger Ausführlichkeit beschreiben kann (was ist ein Elefant, wie schnell kann er laufen usw.) können Entwurfsmuster auf unterschiedliche Weise und mit unterschiedlichem Detailgrad beschrieben werden.

Tatsächlich haben Entwurfsmuster stark dazu beigetragen, dass Softwareentwickler besser miteinander kommunizieren können. Wenn Sie alternative Designentscheidungen besprechen, lässt sich mit Hilfe der Musternamen viel effizienter ausdrücken, welche Struktur gemeint es, statt jedes Mal das Klassendiagramm zu zeichnen. Zudem können existierende Architekturen viel schneller verstanden werden, wenn klar ist, welche Klassen ein bestimmtes Entwurfsmuster implementieren. Denn so wie Sie von verschiedenen Tieren (Elefanten, Hunde, Katzen) unterschiedliche Verhaltensweisen erwarten können, können Sie auch von Klassen, die ein bestimmtes Muster implementieren, definierte Verhaltensweisen erwarten.

1.9 Vorteile und Nachteile von Entwurfsmustern

Zusammenfassend möchte ich folgende Vorteile von Entwurfsmustern hervorheben: Entwurfsmuster beschreiben erprobte Lösungen. Die Lösung wird zudem begründet, da sie auf die Einflussfaktoren des Anwendungskontexts passen muss. Muster stellen eine gute Lösungsalternative dar, d.h. aber auch, dass es alternative Lösungen geben kann. Um besser zu verstehen, ob ein Muster die richtige Alternative ist, werden in der Regel die Konsequenzen (Vor- und Nachteile, Stolpersteine, resultierende Verantwortlichkeiten usw.) explizit beschrieben. Die Diskussion des Problemfeldes mit

seinen Einflussfaktoren („Forces“) macht auf verborgene Probleme aufmerksam und hilft dabei, die Umgebung und die Lösung zu verstehen. Entwurfsmuster führen oft zu flexibleren Designs und reduzieren die Komplexität. Und sie tragen erheblich dazu bei, dass Entwickler besser über komplexe Architekturen reden können.

Bevor wir uns einige Beispiele anschauen, sei aber noch erwähnt, dass es auch immer wieder Probleme mit Entwurfsmuster gibt, meistens weil die oben aufgeführten Prinzipien nicht berücksichtigt werden. So werden Entwurfsmuster oft zu starr interpretiert, falsch umgesetzt oder für den falschen Zweck eingesetzt. Ein Muster führt nicht automatisch zu einem guten Design und durch möglichst viele Muster wird eine Architektur nicht besser. Zudem ist nicht jedes Muster wirklich ausreichend erprobt und auch die Frage, was eigentlich eine „gute“ Lösung ist, lässt sich von unterschiedlichen Personen nicht immer eindeutig beantworten. Zudem müssen bereits gute Programmierkenntnisse und ein Verständnis objektorientierter Entwicklung bekannt sein, um die vorgeschlagenen Lösungen und Konsequenzen bewerten zu können.

Muster liefern eben keine Pauschallösungen, sondern Sie müssen selbst nachdenken und entscheiden, was gut für Ihr Softwaredesign ist. Und ein letztes Mal die Wegmetapher: Wegbeschreibungen können Ihnen dabei helfen, den richtigen Pfad zu finden (=Lösungsmuster). Aber Sie müssen den Weg selbst gehen und benötigen z.B. in den Bergen viel Erfahrung (=Programmiererfahrung). Sie müssen auf neue Gegebenheiten reagieren und eventuell den Weg abändern (=Entwurfsmuster anpassen). Sie müssen mit den Konsequenzen Ihrer Wahl leben können, d.h. ist der Weg zu steil bzw. gefährlich? (=Kann ich z.B. mit erhöhter Laufzeit leben wenn ich dadurch flexibler werde? Traue ich meinem Team zu, das Muster konsequent umzusetzen?) Zudem müssen Sie bewerten, ob der Weg zum richtigen Ziele führt und vertrauenswürdig ist, d.h. wie oft ist dort schon jemand erfolgreich lang gewandert ohne abzustürzen (=ist das Muster wirklich aus realen Softwareprojekten empirisch abgleitet, gibt es mindestens „three known uses“)?

2 Beispiele für Entwurfsmuster

Im Folgenden werde ich das Beispiel aus der Vorlesung aufgreifen und anhand einer einfachen Implementierung einer Umgebung mit einer „Schildkröten“-Grafik zeigen, wie sich mit Entwurfsmuster das Softwaredesign verbessern lässt. Wir werden dabei folgende Entwurfsmuster kennenlernen:

- Strategiemuster: Definiert eine Familie von Algorithmen, kapselt die einzelnen Algorithmen und macht sie untereinander austauschbar
- Dekorierermuster: Ermöglicht das dynamische Hinzufügen zusätzlicher Funktionalität zu einer Komponente
- Abstrakte Fabrik Muster: Bereitstellung einer Schnittstelle zum Erzeugen von Objekten einer Familie verwandter oder voneinander abhängiger Objekte ohne bereits die konkreten Klassen zu spezifizieren

Was damit gemeint ist, werden wir im Einzelnen sehen wenn wir uns die Muster genauer anschauen. Sie werden uns aber helfen, unsere Klassenstruktur zu verbessern.

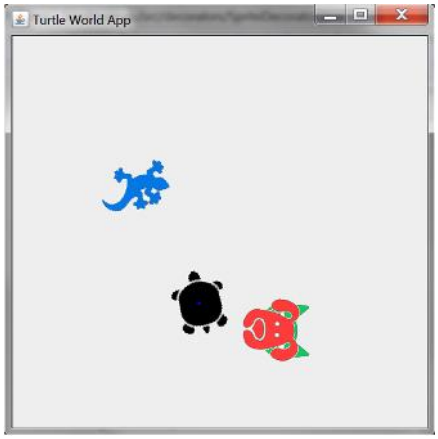
Sie finden die Quelldateien der Beispielanwendungen unter http://www.kohls.de/lehre
Bitte beachten Sie auch die Hinweise am Ende dieses Skriptteils.

2.1 Ausgangslage

Beginnen wir zunächst mit der grundlegenden Infrastruktur. Wir möchten eine graphische Oberfläche haben, auf der sich Sprites unabhängig voneinander bewegen können. Jedes Sprite soll sich auf unterschiedliche Weise bewegen können, einige Sprites bewegen sich aber auf die gleiche Weise. Zudem soll ein Sprite darauf reagieren können, wenn es die Weltgrenzen, also den Rand des Ausgabefensters, erreicht hat.

2.1.1 Beispielanwendung JTurtleWorld

Die Turtle World könnte so aussehen:



Bevor wir uns an die Implementierung der Sprites wagen, müssen wir zunächst unsere virtuelle Welt erschaffen. Hierfür eignet sich das Entwickeln einer neuen Swing Komponente, die wir dann in einem Ausgabefenster platzieren können.

Interessant wird es nachher bei der Implementierung der Sprites. Zum besseren Verständnis wird hier aber auch die Umsetzung der Umgebungs-Komponente erläutert. Diese müssen Sie nicht im Detail nachvollziehen, wichtig ist jedoch das SpriteInterface, auf das wir zu sprechen kommen.

Wir nennen die Swing-Komponente einfach `JTurtleWorld`. Sie erbt von `JComponent`. Wir können die Methode `paintComponent(graphics g)` überschreiben, um die Welt zu zeichnen. Beim Zeichnen sollen zwei Dinge geschehen:

- Für alle Sprites wird eine Bild verwendet und dieses an die Position des Sprites gezeichnet. Das Bild wird entsprechend der Laufrichtung des Sprites bei jedem Zeichnen rotiert.
- Unabhängig von den Sprites wollen wir später Linienzüge mit einem Stift zeichnen können

Wir benötigen also eine Liste von Sprites und eine Liste von Stiften, die jeweils gezeichnet werden können:

```
public class JTurtleWorld extends JComponent {

  ArrayList <SpriteInterface> sprites = new ArrayList<SpriteInterface> ();
  ArrayList <Pen> pens = new ArrayList<Pen> ();
  WorldInfo world = new WorldInfo (600,600);

  …
```

Die `WorldInfo` speichert zum einen die Größe der virtuellen Welt und stellt diese den Sprites zur Verfügung. Wenn sich die Größe der Komponente ändert (Fenster großziehen), wird dies in der

`world` Instanz ebenfalls aktualisiert. Wir erreichen damit später, dass die Sprites nicht den Zugriff auf die gesamte Komponente erhält sondern nur auf die relevanten Weltinformationen. Eine weitere Information, die wir in dieser Welt ablegen werden, ist die Koordinate des letzten Mausklicks. Sprites können diese bei Bedarf als Ziel verwenden.

2.1.2 SpriteInterface für die JTurtleWorld

Wir legen die Sprites in einem Array für `SpriteInterface` Objekte ab. Das `SpriteInterface` ist eine allgemeine Schnittstelle, mit der die `JTurtleWorld` Informationen für die Darstellung eines Sprites einholen kann, ohne zu wissen wie die Sprites tatsächlich implementiert sind. Die Implementierung der Sprites ist also von der `JTurtleWorld` entkoppelt.

Das Interface ist minimal und sieht so aus:

```
public interface SpriteInterface {

	public double getRadianRotation ();
	public Point getLocation ();
	public Point getCenter();
	public BufferedImage getImage ();

	public void setLocation (int x, int y);
	public void setRotation (int degree) ;
	public void setWorldInfo (WorldInfo info);

	public void step ();

}
```

Es gibt ein paar setter und getter Methoden für Positionierung und Laufrichtung, sowie die Übergabe der `WorldInfo`. Die Methode `step()` benachrichtigt eine beliebige Sprite-Implementierung, dass sich das Sprite um einen Schritt bewegen soll.

Die Methode `getImage()` ist für die JTurtleWorld wichtig, um eine bildliche Repräsentation des Sprites zu erhalten. Ohne zu wissen, wie ein Sprite implementiert wird, können wir über diese und die anderen getter-Methoden ein Sprite in unserer Komponente zeichnen.

2.1.3 Ein- und Ausgabe der JTurtleWorld

Die Zeichenarbeit beginnt zunächst in der überschriebenen `paintComponent()` Implementierung. Sie zeichnet zunächst die Weltgrenzen, dann alle Linienzüge der Pens, dann alle Sprites und schließlich noch einen Zielpunkt (=Koordinate des letzten Mausklicks).

```
    public void paintComponent(Graphics g) {
       Graphics2D g2d = (Graphics2D) g;

       // draw world bounds
       Rectangle bounds = world.getBounds();
       g2d.drawRect(bounds.x , bounds.y , bounds.width, bounds.height);

       // draw strokes
       Iterator<Pen>allPens = pens.iterator();
       while (allPens.hasNext()) {
                 allPens.next().paintStrokes(g2d);
       }
```

```
        // draw sprites
        Iterator <SpriteInterface> allSprites = sprites.iterator();
        while (allSprites.hasNext()) {
              drawSprite(g2d, allSprites.next());
        }

          // draw target
        g2d.setColor(Color.blue);
        g2d.fillRect( world.getTarget().x-2, world.getTarget().y-2, 4, 4);


    }
```

Das Zeichnen der Linienzüge und Sprites wird dabei jeweils delegiert. Nur der Vollständigkeit halber hier die Implementierung von `drawSprite()`. Die mathematischen Grundlagen der Zeichentransformation sind hier nicht wichtig.

```
// draw a single sprite
private void drawSprite (Graphics2D g2d ,  SpriteInterface sprite) {

    // get coordinates of sprite
    Point center = sprite.getCenter();
    Point location = sprite.getLocation();

    // Use an affine transform to draw the sprite image rotated
    AffineTransform orig = g2d.getTransform();
    AffineTransform at = new AffineTransform();
    at.setToRotation( sprite.getRadianRotation() ,  center.x ,  center.y );

    g2d.setTransform(at);
    g2d.drawImage(sprite.getImage(), location.x , location.y , null);
    g2d.setTransform(orig);

 }
```

Damit sind wir im Prinzip fast fertig mit der `JTurtleWorld`. Wir benötigen nur noch Event-Listener, um auf Größenänderungen der Komponente und auf Mausklicks zu reagieren. Außerdem brauchen wir einen Timer, der allen Sprites regelmäßig sagt, dass sie einen Schritt weitergehen sollen:

```
this.addMouseListener(new MouseListener() {

            …
            // mouse down sets new target in our world:
            @Override
            public void mousePressed(MouseEvent evt) {

                  world.setTarget(new Point (evt.getX() , evt.getY()));

            }
            …
});


// update word information on component resize
this.addComponentListener(new ComponentListener () {

   …
   @Override
   public void componentResized(ComponentEvent evt) {
      Component c = evt.getComponent();
      world.setBounds(new Rectangle(0,0,600,600));
      world.setBounds(new Rectangle (0 , 0 , c.getWidth(),
                                          c.getHeight()));

   }
     …
});
```

Der Timer stößt für alle Sprites den nächsten Schritt an und ruft `repaint()` auf, damit sich die Komponente neu zeichnet.

```
         Timer t = new Timer(100, new ActionListener () {
            @Override
            public void actionPerformed(ActionEvent arg0) {
             Iterator <SpriteInterface> allSprites = sprites.iterator();
             while (allSprites.hasNext()) {
                   allSprites.next().step();
             }

                  repaint();
            }

         });

         t.start();
```

Damit haben wir unsere JTurtleWorld soweit vorbereitet, dass wir die Sprites implementieren können. Noch ist die Welt sehr langweilig, denn es geschieht darin nichts – es gibt ja noch gar keine Sprites!

Und genau an dieser Stelle wollen wir verschiedene Implementierungsmöglichkeiten betrachten, um zu sehen wo Entwurfsmuster uns gute Lösungen aufzeigen können, um Problemen auszuweichen.

2.1.4 Sprites und Sprite-Familien

Wir wollen für unsere `JTurtleWorld` zunächst vier verschieden Sprites nutzen, die sich in ihrem Wesen (Hund, Katze, Schildkröte, Gekko), ihrer Bewegungsart und der Reaktion auf Kollisionen mit den Weltgrenzen unterscheiden können:

Damit wir Sprites in unserer `JTurtleWorld` platzieren können, müssen wir Objekte von Klassen erzeugen, die das `SpriteInterface` implementieren. Denn nur von diesen Objekten kann unsere Welt Dinge erfahren, wie z.B.: Wo befindet sich das Sprite? In welche Richtung ist es gedreht? Welches Bild soll gezeichnet werden? All diese Informationen werden über die getter-Methoden des `SpriteInterfaces` zugänglich gemacht.

2.2 Implementierungsidee 1 – StupidSprites, eine Sprite-Familie mit nur einer Klasse

Fangen wir mit einer unsauberen Lösung an. Ich spreche daher gleich zu Beginn von dämlichen („stupid“) Sprites. Unsere `JTurtleWorld` Umgebung verlangt ja nur, dass wir Objekte vom Typ `SpriteInterface` erzeugen und dann in die Liste `sprites` einfügen.

Also könnten wir hergehen und eine einzige Klasse schreiben, die das `SpriteInterface` implementiert. Beim Erzeugen von Objektinstanzen werden wir einen Parameter mitgeben, der bestimmt, ob es sich um Hund, Katze, Schildkröte oder einen Gekko handelt.

```
public class StupidSprite implements SpriteInterface {

    // location and and configuration of the sprite
    Point location = new Point (0,0);
    int rotation = 0;
    int stepSize = 12;
    int stepCount = 0;
    BufferedImage img = null;
    WorldInfo worldInfo;
```

```
// in case we want to hunt another sprite
private SpriteInterface targetSprite;

  // different characters:
  public static final int TURTLE  = 0;
  public static final int DOG = 1;
  public static final int CAT = 2;
  public static final int GEKKO = 3;

  private int spriteType = TURTLE;

  public StupidSprite (int type, SpriteInterface targetSprite) {
        this (type);
        this.targetSprite = targetSprite;

  }

  public StupidSprite (int type) {

         spriteType = type;


         String fileName = "joker.gif"; // fallback

         if (type == TURTLE) fileName = "turtleBlack.gif";
         if (type == DOG) fileName = "dogRed.gif";
         if (type == CAT) fileName = "catGreen.gif";
         if (type == GEKKO)  fileName ="gekkoBlue.gif";


           File f = new File("img\\" + fileName);

           try {
               img = ImageIO.read(f);
           } catch (IOException e) {


           }

  }
```

Diese Implementierung speichert für alle Spritearten die gleichen Informationen. Die Unterscheidung zwischen den verschiedenen Arten geschieht über das Feld `spriteType`. Da einige Sprites auch andere verfolgen sollen (Hund jagt Katze) müssen wir noch ein Feld für ein `targetSprite` einführen. Allerdings benötigen wir dieses Feld zurzeit nur für den Hund – für alle anderen Spritearten bleibt es ungenutzt. Dies ist schon einmal nicht so elegant. Zudem sehen wir bereits im Konstruktor, dass für jede Spriteart eine Fallunterscheidung getroffen werden muss (hier wird unterschieden, welche Bilddatei geladen werden soll).

Dies wäre alleine noch nicht weiter schlimm, aber diese Mehrfachverzweigung begegnet uns immer und immer wieder wenn wir die anderen benötigten Methoden des Interfaces implementieren:

```
public void step () {
       // Kollision mit Außenkanten?
      checkCollision();

       if (spriteType == TURTLE) {
              Point target = worldInfo.getTarget();
              rotateToTarget(target);
       }

       if (spriteType == DOG) {
             rotateToTarget(targetSprite.getCenter());
       }

       if (spriteType == CAT) {
             int y = worldInfo.getTarget().y;
             if ( y != location.y) {
                   location.y =  y;
             }
       }

       if (spriteType == GEKKO) {
              stepCount++;
              if (stepCount == 10) {
                    stepCount = 0;
                    rotation += 20;
              }
       }
      // move forward:
       double r = Math.toRadians(Math.abs(rotation)-90);
     location.x += Math.round(stepSize * Math.cos(r));
     location.y += Math.round(stepSize * Math.sin(r));
}
```

Die gleiche Mehrfachverzweigung finden wir wenn wir auf die Kollision mit den Weltgrenzen eingehen, da auch hier immer anders reagiert werden muss, z.B. für die Kollision am linken Bildschirmrand:

```
public void onLeftCollision() {
       if (spriteType == TURTLE || spriteType == DOG) {
              location.x = worldInfo.getCenterX();
       }
       if (spriteType == CAT) {
             location.x = worldInfo.getRight();
       }

       if (spriteType == GEKKO) {
              rotation += 180;
       }
}
```

Diese vielen if-Verzweigungen sind alles andere als schön. Zudem haben wir viele Codeverdoppelungen, da sich die Sprites teils identisch verhalten. Am schlimmsten ist jedoch, dass Code, der eigentlich zusammengehört, quer durch die Klasse verteilt ist. Die spezifischen Verhaltensweisen des Hunde-Sprites sind z.B. an vielen verschiedenen Stellen zu finden.

Stellen Sie sich nun vor, wir möchten auch noch Mäuse, Tiger, Vögel usw. implementieren. Dann hätten wir eine riesige Klasse und jedes Mal wenn ein neues Tier dazu kommt müssen wir quer durch die Klasse die Ergänzungen vornehmen.

2.3 Implementierungsidee 2 – eine alternative Sprite-Familie mit Unterklassen

Gut, dass wir in objektorientierter Programmierung bewandert sind und uns sofort eine Lösung einfällt! Hund und Katze sind ja unterschiedlich, aber haben auch Gemeinsamkeiten. Also benötigen wir eine gemeinsame Oberklasse `Sprite`, in der alle Gemeinsamkeiten zusammengefasst sind und spezielle Unterklassen, in denen die Besonderheiten der verschiedenen Spritearten auftauchen.

Wir implementieren das `SpriteInterface` durch die Klasse `Sprite`. Darin legen wir die allgemeinen Daten und Verhaltensweisen fest. Für die Besonderheiten erweitern wir die Sprite-Klasse durch die Unterklassen `Dog, Cat, Turtle` und `Gekko`.

```
public class Sprite implements SpriteInterface {

    // location and and configuration of the sprite
    Point location = new Point (0,0);
    int rotation = 0;
    int stepSize = 12;
    BufferedImage img = null;
    WorldInfo worldInfo;


    public Sprite () {

              File f = new File("img\\" + fileName());

              try {
                  img = ImageIO.read(f);
              } catch (IOException e) {
                 …
              }
      }
```

Schon beim Konstruktur fällt auf, dass es keine Mehrfachverzweigung gibt. Stattdessen wird `fileName()` aufgerufen und die verschiedenen Unterklassen können diese Methode überschreiben, um den richtigen Dateinamen zu liefern:

```
public class Dog extends Sprite {

      …
      public String fileName() { return "dogRed.gif";}

       …

}
```

Wir können auch alle weiteren allgemeinen Dinge in der Sprite Klasse belassen, z.B. je nach Bewegungsrichtung und Schrittgröße den nächsten Schritt ausführen und die Kollision mit den Weltgrenzen überprüfen:

```
public class Sprite implements SpriteInterface {

	…
	public void step () {
		// Kollision mit Außenkanten?
		checkCollision();
		// move forward:
		double r = Math.toRadians(Math.abs(rotation)-90);
		location.x += Math.round(stepSize * Math.cos(r));
		location.y += Math.round(stepSize * Math.sin(r));

	}

	public void onTopCollision() { }

	public void onBottomCollision() { }

	public void onLeftCollision () { }

	public void onRightCollision () { }

	private void checkCollision() {
		if (worldInfo == null) return;
		if (location.x < worldInfo.getLeft() ) onLeftCollision();
		if (location.x > worldInfo.getRight()) onRightCollision() ;
		if (location.y < worldInfo.getTop() ) onTopCollision();
		if (location.y > worldInfo.getBottom()) onBottomCollision();

	}
}
```

Die Methode `step()` kann von den Unterklassen überschrieben werden, um die Bewegung zu spezialisieren. Hier wird nur ein Schritt in die aktuelle Richtung ausgeführt.

Die Kollisionserkennung mit den Weltgrenzen in `checkCollision()` ist auch für alle Sprites gleich. Allerdings ist die Reaktion auf eine Kollision unterschiedlich. Daher sind in der Sprite-Klasse die Methoden `onTopCollision()`, `onButtomCollision()` usw. leere Implementierung. Per Default wird also gar nicht auf eine Kollision reagiert, aber Unterklassen können die Methoden überschreiben und so das Kollisionsverhalten verändern.

Schauen wir uns also die vollständigen Implementierungen von Dog und Cat an.

```
public class Cat extends Sprite {

	public String fileName()	{ return "catGreen.gif";}

	public void step () {
		int y = worldInfo.getTarget().y;
		if ( y != location.y) {
			location.y =  y;
		}
		super.step();
	}

	public void onTopCollision() {
		location.y = worldInfo.getBottom();
	}

	public void onBottomCollision() {
		location.y = worldInfo.getTop();
	}

	public void onLeftCollision() {
	    location.x = worldInfo.getRight();
	}

	public void onRightCollision() {
		location.x = worldInfo.getLeft();
	}

}
```

Wir sehen hier sehr schön, dass die gesamte Logik, die speziell zur Katze gehört, übersichtlich in einer Klasse ist.

In der Methode `step()` wird das Bewegungsverhalten angepasst. Wenn es ein definiertes Ziel in der Welt gibt, dann springt die Katze vertikal auf die Höhe des Ziels. Ansonsten ändert sie ihre Bewegung nicht und daher wird mit `super.step()` das allgemeine Bewegungsverhalten der Superklasse aufgerufen.

Während `onTopCollision(), onBottomcollision()` usw. in der Sprite-Klasse leere Implementierungen waren, definiert die Unterklasse `Cat` genau, was bei Kollisionen zu tun ist.

Zum Vergleich schauen wir uns noch die Implementierung von Dog an.

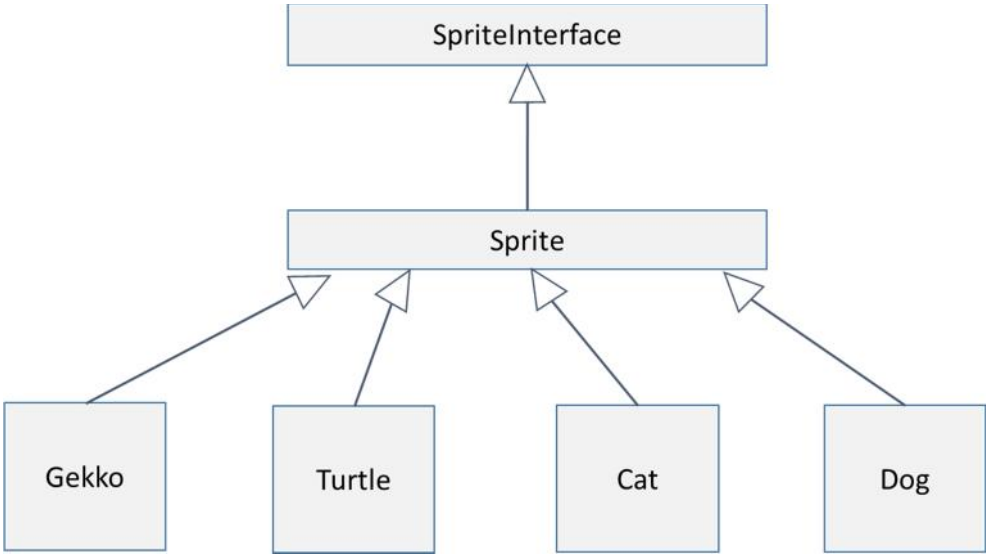
```
public class Dog extends Sprite {

	SpriteInterface huntedSprite;

	public Dog (SpriteInterface target) {

		huntedSprite = target;

	}

	public String fileName() { return "dogRed.gif";}

	public void step () {

	if (huntedSprite != null) {
		this.rotateToTarget(huntedSprite.getCenter());

	}

		super.step();

	}

	public void onTopCollision() {
		location.y = worldInfo.getCenterY();
	}

	public void onBottomCollision() {
		location.y = worldInfo.getCenterY();
	}

	public void onLeftCollision() {
		location.y = worldInfo.getCenterY();
	}

	public void onRightCollision() {
		location.y = worldInfo.getCenterY();
	}
}
```

Zunächst sehen wir, dass Dog ein zusätzliches Feld hat, nämlich das verfolgte Sprite `huntedSprite`. Damit haben wir dieses Feld auch nur für Objekte, die es tatsächlich benötigen.

In der Methode `step()` sehen wir, dass für `Dog` das Bewegungsverhalten anders umgesetzt wird als für `Cat`. Statt lauter if-Verzweigungen zu haben, sind die spezifischen Verhaltensweisen dort, wo sie hingehören.

Prima, jetzt haben wir doch eine hervorragende Lösung für unsere Sprite Implementierung gefunden, oder?

Die allgemeinen Dinge sind in der Sprite-Klasse, die Besonderheiten in den Unterklassen gekapselt und zusammengefasst. So wie es scheint haben wir eine gute Lösung gefunden. Das Klassendiagramm sieht nun so aus:



2.4 Problem der Klassenexplosion

Aber erinnern Sie sich daran, dass Probleme oft nicht offensichtlich sind und erst beim Fortschreiten des Projektes auftreten?

Wenn Sie sich die beiden Unterklassen `Cat` und `Dog` anschauen, dann sehen Sie zum einen, dass wir zusammengehörende Funktionalität auch zusammen innerhalb einer Klasse stehen haben. Das ist etwas Gutes!

Wir können auch sehr leicht neue Tiere hinzufügen. Wenn wir eine Maus dazu nehmen möchten, dann leiten wir eine weitere Unterklasse Mouse ab. Die Implementierung ist sehr viel einfacher: zum einen ist vieles in der `Sprite`-Klasse implementiert, zum anderen sind alle Ergänzungen an einer Stelle durchführbar.

Was passiert nun aber, wenn Sie eine Katze benötigen, die sich so bewegt wie der Hund (weil sie z.B. Mäusen hinterher läuft)? Dann müssten Sie von der Katze zwei Unterklassen bilden:

Ein Katze, die sich wie bisher linear bewegt und nur bei Mausklick auf die Ziellinie springt, z.B. `LinearCat` und eine weiter Katze, die sich in Richtung Maus bewegt, z.B. `HuntingCat`

Und wenn die Katze vor dem Hund fliehen möchte? Dann leiten wir eine Unterklasse `EscapingCat` ab!

So, und nun möchten wir auch noch freundliche Hunde haben, die einfach herumtrollen (sich also wie ein `Gekko` bewegen können) oder sich zum Frauchen/Herrchen bewegen, nachdem diese(r) mit der Maus in die Welt geklickt hat (so wie die `Turtle`). Na, dann führen wir einfach die Unterklassen `HuntingDog, FriendlyDog` und `LoyalDog` ein.

Nun ist es aber so, dass jeder dieser Hunde nicht immer bei einer Kollision mit den Weltgrenzen in die Mitte zurückspringen soll, sondern einfach ihre Richtung ändern (so wie der `Gekko`). Ah, kein Problem! Legen wir also weitere Unterklassen an: `HuntingDogWhoJumpsBack`, `HuntingDogWhoRotates`, `FriendlyDogWhoJumpsBag`, `FriendlyDogWhoRotates`, `LoyahlDogWhoJumpsBack`, `LoyalDogWhoRotates`… Und was Hunde können, können Katzen schon lange – auch hier benötigen wir Unterklassen für die verschiedenen Reaktionen auf die Kollision mit den Weltgrenzen.

Richtig arbeitsintensiv wird es dann, wenn Sie ein neues Tier in Ihre Welt aufnehmen möchten. Auch von der `Mouse` müssten Sie dann zahlreiche Unterklassen implementieren. Das wird nicht nur sehr unübersichtlich, sondern Sie handeln sich jede Menge Codeverdopplungen ein. Denn der `HuntingDogWhoRotates` und die `HuntingCatWhoRotates` sind zwar zwei verschiedene Wesen, aber sowohl ihre Bewegungsstrategie wie auch ihre Kollisionsstrategie sind dieselbe.

Am schlimmsten bei diesem Lösungsansatz sind aber die vielen Unterklassen die entstehen. Hier sieht man sehr deutlich, dass es Probleme gibt, die erst später im Design auftreten können. Insofern ist es wichtig, wenn man sich bereits vorher über mögliche Probleme schlau macht und auch gleich eine passende Lösung findet.

2.5 Das Strategie Muster für die Beispielanwendung oder Eine SmartSprite-Familie mit Strategie

Ein Entwurfsmuster, das uns in unserer Situation helfen kann ist das Strategie-Muster. Beim Strategiemuster lagern Sie das veränderliche Verhalten (bzw. die Strategie oder unterschiedliche Algorithmen) in ein weiteres Objekt aus und delegieren die eigentliche Arbeit an dieses Objekt. Unsere Sprites implementieren dann ihr spezielles Bewegungsverhalten nicht mehr selbst oder in Unterklassen. Stattdessen delegieren Sie die Berechnung des nächsten Schritts an eine von mehreren verschieden Bewegungsstrategien.

Wie bei der letzten Lösung implementieren wir das `SpriteInterface` durch eine allgemeine Klasse, die alle Gemeinsamkeiten implementiert. Wir nennen diese Klasse `SmartSprite`. Auch von `SmartSprite` werden wir die Unterklassen `Cat, Dot, Gekko` und `Turtle` ableiten, um sie zu spezialisieren (um spezielle Eigenschaften der Tiere, in unserem Fall nur verschiedene Bilder, abzubilden).

Der Unterschied liegt aber nun darin, dass wir die Bewegung (und das Kollisionsverhalten) nicht mehr in `SmartSprite` oder in den Unterklassen berechnen, sondern an eine Bewegungsstrategie delegieren.

```
public class SmartSprite implements SpriteInterface {

   StepStrategy stepStrategy;

   OnCollisionStrategyInterface collisionStrategy;

    …


   public void step () {
         // Kollision mit Außenkanten?
         checkCollision();

         stepStrategy.step(this);

         // move forward:
             double r = Math.toRadians(Math.abs(rotation)-90);
           location.x += Math.round(stepSize * Math.cos(r));
           location.y += Math.round(stepSize * Math.sin(r));

   }
```

Wenn wir die `step()` Methode betrachten, dann sehen wir, dass sich nicht viel ändert, außer dass die eigentliche Berechnung des Schritts delegiert wird.

Damit verschiedene Bewegungsstrategien beliebig austauschbar sind, benötigen wir zunächst ein abstraktes Interface:

```
public interface StepStrategy {

      public void step (SmartSprite sprite);

}
```

Die Klasse `SmartSprite` besitzt eine Referenz auf eine konkrete Implementierung dieser Strategie. Bei der Erzeugung der Sprites lässt sich dann festlegen, welche Strategie verwendet wird. Zudem lässt sich die Strategie jederzeit austauschen. Die Bewegungsstrategie, sich zum letzten Mausklick zu bewegen, könnte so aussehen:

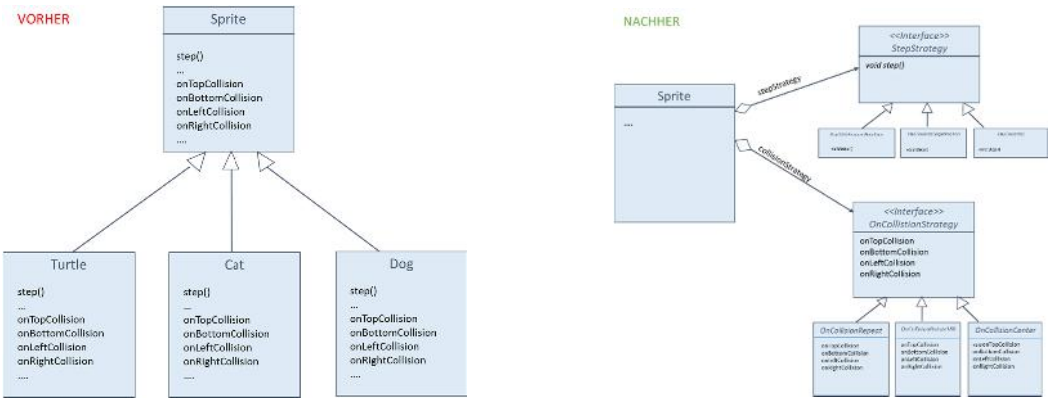
```
public class StepTowardsTargetPosition implements StepStrategy {

      @Override
      public void step (SmartSprite sprite) {

      // rotate towards target:
      Point target = sprite.worldInfo.getTarget();
        sprite.rotateToTarget(target);
      }

}
```

Ein kleiner Nachteil ist, dass Sie der Strategie eine Referenz auf das `SmartSprite` übergeben müssen, damit es dieses auch verändern kann. Bislang waren die `Sprites` selbst dafür verantwortlich, die Richtung zu ändern. Nun ist die Strategie dafür verantwortlich, sie muss also die Daten des `SmartSprites` lesen und verändern können. Dies kann eventuell dazu führen, dass Sie die Kapselung aufbrechen müssen, damit die Strategie an alle benötigten Daten kommt. Dies ist eine Konsequenz, bei der man entscheiden muss, ob man damit leben kann.

Auf der anderen Seite haben Sie aber sehr viel gewonnen, denn jetzt können Sie die `SmartSprites` beliebig mit Strategien für die Bewegung und Kollisionen konfigurieren.

Zum Vergleich: Wenn Sie vier verschiedene Charaktere, drei verschiedene Kollisionsstrategien und vier verschiedene Bewegungsstrategien potentiell benötigen, dann müssten Sie 4 x 3 x 4 = 48 Unterklassen bilden! Wenn Sie eine weitere Bewegungsart hinzufügen wollten müssten Sie diese wiederum für alle Unterklassen umsetzen…

Mit dem Strategiemuster haben Sie diese Probleme nicht. Da für das `SmartSprite` mit jeder `StepStrategy` arbeiten kann, können Sie jederzeit weitere Bewegungsstrategien hinzufügen. Zudem können Sie jede Spriteart (`Dog, Cat, Turtle, Gekko`) mit jeder Bewegungsstrate gie und wiederum mit jeder Kollisionsstrategie kombinieren. Sie kommen somit zu folgendem veränderten Klassendiagramm:



Dies sieht nun zunächst danach aus, dass Sie vielmehr Klassen als vorher haben. Das stimmt! Zunächst haben Sie für die Bewegungsstrategie durch das Interface und das Auslagern der Bewegung in verschiedene Implementierungen mehr Klassen als vorher. Dies zahlt sich aber aus sobald Sie verschiedene Bewegungs- und Kollisionsarten kombinieren möchten, weitere Strategien oder Spritearten hinzufügen. Erinnern Sie sich an das Eingangsbeispiel mit dem Weg: manchmal muss man erst einen kleinen Umweg gehen, um nachher nicht vor der Felswand (hier: der explodierenden Anzahl von Unterklassen) zu stehen.

2.6 Das Strategie Muster allgemein

Dass ein Algorithmus gekapselt und ggf. dynamisch ausgetauscht werden muss, ist kein Spezialfall unserer `JTurtleWorld`. Vielmehr lässt sich das Strategiemuster meist dort gut einsetzen, wo Sie eine Familie von Algorithmen haben, die untereinander austauschbar sein sollen, z.B. alternative Sortieralgorithmen, verschiedene Analysestrategien, unterschiedliche Reaktionen auf Benutzereingaben je nach Arbeitsmodi, verschiedene Validierungsmechanismen für die Dateneingabe, oder der Austausch verschiedener Speichermodelle. Im Prinzip sind die verschiedenen `List` Implementierungen in Java auch alternative Strategien, Daten zu speichern. Client-Code muss nur das

`List`-Interface kennen, um mit Listen arbeiten zu können. Die tatsächliche Implementierung der Liste lässt sich austauschen. Dies geschieht, ohne dass Sie Ihren Code anpassen müssen.

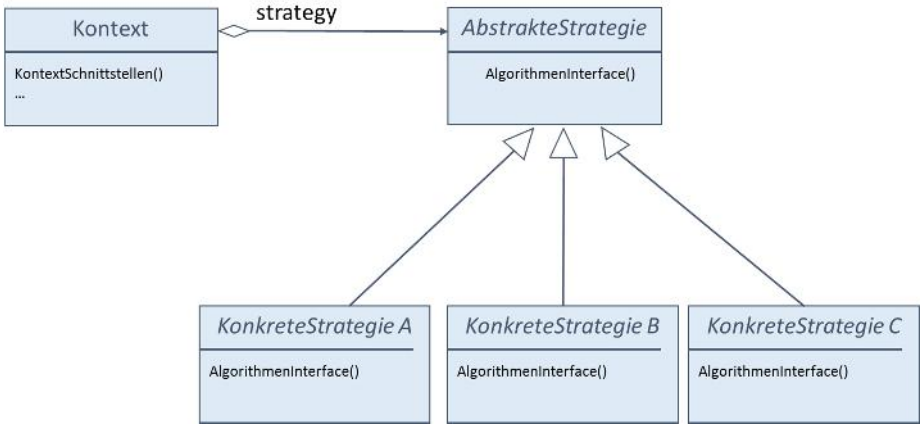
2.6.1 Anwendungskontext und Forces

Nutzen Sie das Strategiemuster dann, wenn Sie mehrere Klassen haben, die sich nur in einem Verhalten unterscheiden. Dieses Verhalten (Algorithmus) kann durch Delegation ausgelagert werden. Würde man für jede Variation des Algorithmus eine Unterklasse erzeugen, hätte man schnell eine unüberschaubare Anzahl an Unterklassen („Klassenexplosion“). Das Muster ist auch nützlich, um Codeverdoppelung zu vermeiden, wenn nämlich verschiedene Klassen dasselbe Verhalten benötigen.

Legt man alle alternativen Berechnungswege in eine Klasse (so wie beim `StupidSprite`), wird diese schnell schwer wartbar, zu groß und unflexibel. Auch eine allgemeine Oberklasse (wie `Sprite`) sollte nicht mit Funktionen überladen werden, da jede Änderungen Auswirkungen auf alle Unterklassen mit sich bringt und dies zu unerwünschten Abhängigkeiten führen kann.

2.6.2 Lösung

Die allgemeine Lösungsstruktur der Strategie sieht so aus:



Der `Kontext` ist der Teil Ihres Systems, der auf einen Algorithmus zurückgreifen möchte. In unserem Beispiel war dies die `SmartSprite` Klasse. Der Kontext kann über beliebig viele eigene Daten und Methoden verfügen, hier mit `KontextSchnittstellen()` angedeutet.

Für die eigentliche Berechnung wird nun auf eine Strategie zugegriffen. Damit der Kontext von konkreten Strategien unabhängig bleibt, nutzen wir ein Interface, die `AbstrakteStrategie`. Die `AbstrakteStrategie` stellt uns Methoden bereit, mit der sich eine Berechnung umsetzen lässt. Dies ist mit `AlgorithmenInterface()` allgemein angedeutet. Bei der `StepStrategy` war dies `step (SmartSprite sprite)`. Bei einer Sortier-Strategie könnte dies eine Methode `suche()` sein.

Das Interface kann aber auch mehrere Methoden bereitstellen wie das Beispiel des `OnCollisonStratgy` Interfaces zeigt. Dieses enthält die Methoden `onTopCollison(), onBottomCollision(), onLeftCollision(), onRightCollision()`. Sie legen also immer gleich die gesamte Strategie fest, wie auf Kollisionen reagiert werden soll.

Der Kontext delegiert nun seine Arbeit, indem er nicht mehr selbst einen Algorithmus implementiert, sondern das `AlgorithmenInterface()` der jeweiligen Strategie aufruft. `SmartSprite` berechnet z.B. nicht mehr selbst die Bewegung sondern delegiert dies an die `StepStrategy`. Der Kontext nutzt nur die `AbstrakteStrategie` und muss somit nicht wissen mit welcher konkreten Strategie gearbeitet wird. Dies kann bei der Objektinstanzierung dynamisch initialisiert und später auch noch geändert werden.

Ein allgemeines Beispiel zeigt das Auslagern des Algorithmus. Ohne Strategiemuster hätten Sie im Code des `Kontexts` vielleicht stehen:

KontextKlasse1	**KontextKlasse2**	**KontextKlasse3**
`Anweisung1`	`Anweisung1`	`Anweisung1`
`Anweisung2`	`Anweisung2`	`Anweisung2`
`SpzAlgorithmusA`	**`SpzAlgorithmusB`**	**`SpzAlgorithmusC`**
`Anweisung3`	`Anweisung3`	`Anweisung3`
`Anweisung4`	`Anweisung4`	`Anweisung4`

Für jeden alternativen Algorithmus müssen Sie eine Unterklasse des Kontexts bilden, die Anweisungen 1-4 wiederholen sich. Wenn Sie das Strategiemuster nutzen, wird im `Kontext` auf den speziellen Algorithmus über eine Strategie zu gegriffen:

```
AllgemeinerAlgorithmus algorithmus = Zuweisung eines speziellen
                                     Algorithmus A,B oder C

Anweisung1
Anweisung2

algorithmus.ausfuehren()

Anweisung3
Anweisung4
```

2.6.3 Konsequenzen

Die Vorteile des Strategiemusters sind:

- Definiert eine Familie von Algorithmen oder Verhalten
- Alternative zur Unterklassenbildung durch Delegation
    - Spezialisierung des Kontexts durch Delegation statt Unterklassen
    - Viele Konfigurationen durch Kombinationen möglich
    - Verhalten kann dynamisch ausgetauscht werden
- Vermeidung von Mehrfachverzweigungen
- Ermöglicht alternative Implementierungen desselben Verhaltens (z.B. Laufzeit vs. Sicherheit)

Die Nachteile und Verantwortlichkeiten dieser Lösung sind:

- Der Kontext bzw. die Klienten müssen die verschiedenen Strategien kennen
- Overhead an Kommunikation zwischen Strategie und Kontext
    - Informationen müssen an die Strategie weitergereicht werden
    - Informationen werden oft gar nicht genutzt
    - Bricht eventuell die Kapselung auf
- Erhöhte Anzahl von Objekten zur Laufzeit

2.7 Neue Anforderungen für die Beispielanwendung

Die Implementierung mit dem SmartSprite sowie verschiedenen Strategien für die Bewegung und Kollision ist bereits eine sehr gute und vor allem flexible Lösung. Das Problem der Klassenexplosion, der Codeverdoppelung, des Auseinanderziehens zusammengehöriger Codeblöcke und die vielen Mehrfachverzweigungen sind wir los!

Nehmen wir an, dass wir unsere Sprite-Implementierung gut ausgebaut haben: mit vielen verschiedenen Tieren, Bewegungsstrategien und Kollisionsstrategien. Unsere bestehende Lösung wollen wir möglichst nicht mehr ändern, außer durch weitere Konfigurationen, Strategien und Tiere erweitern.

Doch nun kommt durch einen Auftraggeber eine weitere Funktionsanforderung hinzu. Unsere `JTurtleWorld` ist bereits darauf ausgelegt, dass Streckenzüge mit einem Stift (Pen) gezeichnet werden können. Nun soll diese Funktion auch genutzt werden, und einzelne Sprites sollen – je nach Bedarf – einen Streckenzug erzeugen wenn sie sich bewegen.

Dies stellt uns vor ein neues Problem. Bisher waren wir sehr flexibel und konnten mit dem Strategiemuster unterschiedliche Verhaltensweisen **austauschen und konfigurieren**. Doch nun geht es darum, Funktionalität dynamisch **zu ergänzen**. Dies ist eine neue und zusätzliche Problemstellung. Häufig benötigen wir mehrere Lösungen für ein komplexes Problem, d.h. wir können dann natürlich auch mehrere Entwurfsmuster einsetzen und miteinander kombinieren.

Das Erzeugen des Streckenzugs soll während der Bewegung geschehen. Eine erste Lösungsidee könnte es sein, Bewegungsstrategien zu implementieren, die auch einen Streckenzug erzeugen. Da ein Sprite sich aber auch bewegen kann, ohne etwa zu zeichnen, müssten wir für jede Bewegungsform wieder zwei Unterklassen der `StepStrategy` erzeugen. Dies würde wiederum zu einer Klassenexplosion führen. Insbesondere wenn man vielleicht noch weitere Funktionen ergänzen möchte (z.B. eine Log-Funktion, die ausgibt in welche Himmelsrichtung das Sprite läuft). Wie können wir nun also die Klassenexplosion umgehen?

Außerdem hatten wir gesagt, dass wir mit unserer `SmartSprite` Implementierung recht zufrieden sind und diese vielleicht schon breit ausgebaut haben (viele Tiere, viele Strategien). Diese wollen wir möglichst nicht ändern.

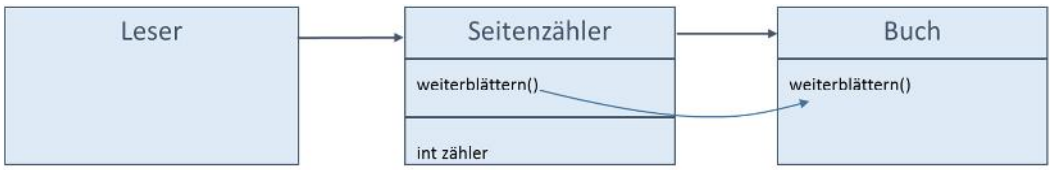
Das heißt, wir möchten unsere bestehende Implementierung nicht anfassen und sie trotzdem dynamisch um Funktionalität erweitern! Unmöglich? Das Dekorierermuster bietet hier einen Lösungweg an.

2.8 Beispiel Dekorierer für Komponenten

Dekorieren bedeutet, dass Sie eine Komponente unverändert lassen, aber mit zusätzlicher Funktionalitäten umhüllen (also dekorieren). Wer auch immer diese Komponente nutzt, sollte die umhüllte Komponente in gleicher Weise nutzen können. Daraus folgt, dass die Komponente und die Dekorationshülle dasselbe Interface bereitstellen müssen.

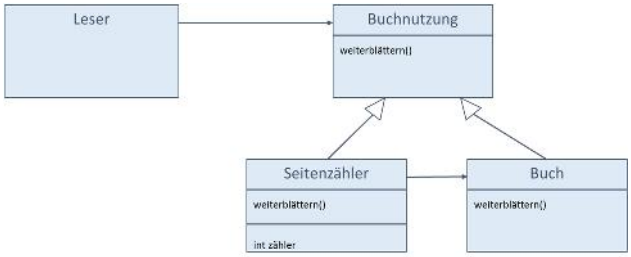
Nehmen wir an, wir haben eine Klasse `Buch`, in der es nur eine Methode gibt, nämlich `weiterblättern()`. Das Buch wird genutzt von einem `Leser`.

Der `Leser` kann nun an verschiedensten Stellen die Methode `weiterblättern()` aufrufen. Nun wollen wir erreichen, dass bei jedem `weiterblättern()` gezählt wird, wie oft weitergeblättert worden ist. Allerdings wollen wir dazu das Buch unverändert lassen und nicht in der Klasse `Leser` zählen, wie oft weitergeblättert worden ist (denn wir rufen `weiterblättern()` an verschiedenen Stellen auf). Die Lösung liegt darin, dass man das Buch mit einem Seitenzähler umhüllt.



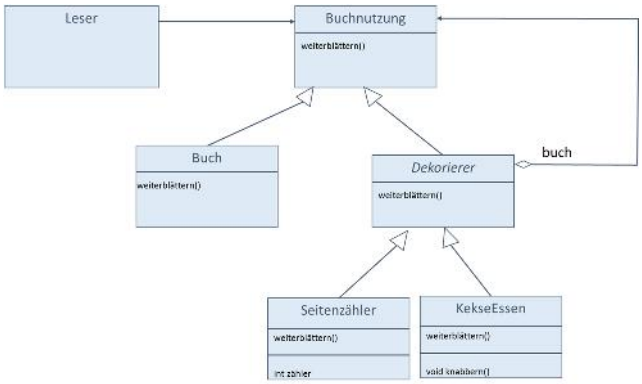
Der Leser nutzt nun nicht mehr direkt das `Buch` zum Weiterblättern sondern den `Seitenzähler`. Jedes Mal wenn der `Leser` die Methode `weiterblättern()` aufruft wird der `Seitenzähler` sein Datenfeld `zähler` erhöhen. Zudem besitzt der `Seitenzähler` eine Referenz auf das `Buch` und ruft danach `weiterblättern()` für das Buch auf. Somit wird das `Buch` wie bisher weitergeblättert und zusätzlich gezählt. Der Leser ruft zwar nicht mehr `weiterblättern()` direkt für das `Buch` auf, aber der `Seitenzähler` übernimmt dies. Wir haben also zusätzliche Funktionalität erreicht, ohne die Komponente, also hier das `Buch`, zu verändern.

Allerdings müssten wir den Client-Code ändern, denn der `Leser` nutzt nun nicht mehr das `Buch` sondern den `Seitenzähler`. Dies ist nicht sonderlich schön, denn vielleicht möchte der Leser gar nicht immer die Seiten zählen. Nun fällt aber gleich auf, dass sowohl `Seitenzähler` wie auch `Buch` die Methode `weiterblättern()` besitzen. Wenn wir nun ein Interface einführen, das die Methode `weiterblättern()` festlegt, dann kann es dem `Leser` egal sein, ob er direkt mit einem `Buch` arbeitet, oder mit einem `Seitenzähler` auf das `Buch` zugreift.

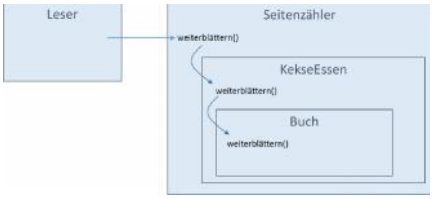


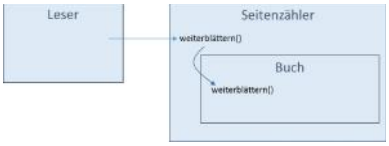
Der Leser kann nun über ein `Buchnutzung` Objekt auf ein `Buch` zugreifen. Dies kann entweder direkt ein `Buch` sein, oder ein `Seitenzähler`, der auf ein `Buch` zeigt. Auf diese Weise lässt sich das Buch bereits gut mit einer Zusatzfunktion dekorieren. Wir gewinnen noch mehr Flexibilität wenn der `Seitenzähler` nicht direkt auf das `Buch` verweisen würde sondern ebenfalls auf die Schnittstelle `Buchnutzung`. Denn dann kann es auch dem `Seitenzähler` egal sein, ob er direkt für ein `Buch` die Funktion `weiterblättern()` aufruft oder wiederum für ein dekorierendes Objekt. Wir könnten einen weiteren Dekorierer schreiben, der bei jedem `weiterblättern()` das Essen von Keksen anstößt.

Allen Dekorierern gemein ist, dass sie eine Referenz auf die `Buchnutzung` besitzen. Daher führen wir eine abstrakte Klasse `Dekorierer` ein, die eine Referenz auf eine `Buchnutzung` besitzt.



Da es sich bei der Buchnutzung um die eigentliche Komponente `Buch` oder um weitere Dekorierer handeln kann, lässt sich die eigentliche Komponente mit beliebig vielen Hüllen dekoriereren. Konkrete Objektkonfigurationen könnten so aussehen:





Für den Leser ist es egal, ob er auf einen `Seitenzähler`, auf `KekseEssen` oder direkt auf ein `Buch` zugreift. Denn statt eine Referenz auf die konkreten Klassen zu nutzen verwendet der Leser eine Referenz auf `Buchnutzung`. `Buchnutzung` kann nun direkt mit einem `Buch` erfolgen oder über einen oder mehrere Dekorierer, die am Ende der Kette `weiterblättern()` für eine konkrete Komonente, also ein echtes `Buch` aufrufen.

2.9 Das Dekorierer Muster für die Beispielanwendung

Doch was hat dies alles mit unserer `JTurtleWorld` und unserem Problem, einige Sprites zeichnen zu lassen, zu tun? Nun, das Problem ist das gleiche und wir können auch die gleiche Lösung nutzen. Für die `JTurtleWorld` gibt es ein `SpriteInterface`. Unsere `SmartSprites`, bzw. die Unterklassen `Dog, Cat, Turtle` und `Gekko`, sind konkrete Komponenten. Diese wollen wir

nach Möglichkeit nicht ändern. Wenn wir nun aber eine Dekorierer-Klasse einführen, dann müsste diese zum einen das `SpriteInterface` implementieren (damit diese genauso wie andere Sprites von der `JTurtleWorld` genutzt werden können), zum anderen muss jeder Dekorierer wiederum auf ein Sprite zugreifen können. Da es sich dabei um beliebige Sprites handeln soll, verwenden wir hierfür ein `SpriteInterface` als Referenz:

```
abstract public class SpriteDecorator implements SpriteInterface {


	protected SpriteInterface decoratedSprite;


	@Override
	public double getRadianRotation() {
		return decoratedSprite.getRadianRotation();

	}

	@Override
	public Point getLocation() {
		return decoratedSprite.getLocation();

	}
	…
}
```

Der `SpriteDecorator` ist eine abstrakte Klasse. Sie erledigt zwei Dinge. Zum einen besitzt sie eine Referenz auf das `decoratedSprite`. Zum anderen implementiert sie die Methoden des `SpriteInterface`, indem sie jede Anfrage an das decoratedSprite weiterleitet. Hier wird also noch keine zusätzliche Funktionalität hinzugefügt.

Eine konkrete Implementierung des SpriteDecorators ist die PenDecoration:

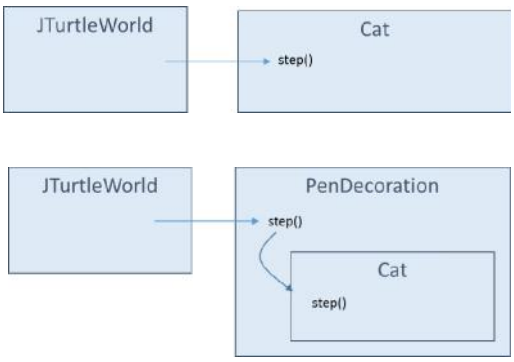
```
public class PenDecoration extends SpriteDecorator implements Pen{


	private Polygon strokes = new Polygon();
	private Color color;

	public PenDecoration (SpriteInterface decoratedSprite , Color c) {
		this.decoratedSprite = decoratedSprite;
		color = c;
	}
```

```
public void step () {

      decoratedSprite.step();

      if (strokes != null) {
            Point center = decoratedSprite.getCenter();
            strokes.addPoint(  center.x  , center.y  );
      }
```

…

Interessant ist hier die `step()`Methode. Auch sie delegiert die Arbeit zunächst an den `decoratedSprite`. So wird sichergestellt, dass auch das echte Sprite (z.B. eine `Cat` Instanz) den Schritt ausführt. Zusätzlich speichert der `PenDecorator` aber noch, wohin sich das Sprite bewegt hat und fügt die neue Position in ein Polygon `strokes` ein. Dieses Polygon wird später dazu verwendet, den Streckzug zu zeichnen (`PenDecoration` implementiert noch ein `Pen`-Interface, darüber kann eine paint-Methode aufgerufen werden.).

`step()` wird nun nicht mehr direkt für die `Cat` sondern für `PenDecoration` Objekte aufgerufen, die dann ihrerseits `step()` für `Cat` aufrufen:



Für die `JTurtleWorld` ist es transparent, ob beim Aufruf von `step()` eine `Cat` Instanz oder eine `PenDecoration` Instanz verwendet wird, da sie ohnehin nur mit dem `SpriteInterface` arbeitet, d.h. sie behandelt alle Implementierungen gleich.

Vorher wurde die Katze der `JTurtleWorld` so bekannt gemacht:

```
SpriteInterface catSprite = new CatSprite();
sprites.add(catSprite)
```

Wenn die Katze nun um die Zeichenfunktionalität erweitert werden soll, ändert sich die Initialisierung wie folgt:

```
SpriteInterface catSprite = new CatSprite();
PenDecoration decoratedCatSprite = new PenDecoration (catSprite ,
                                                      Color.red);
sprites.add(decoratedCatSprite)
```

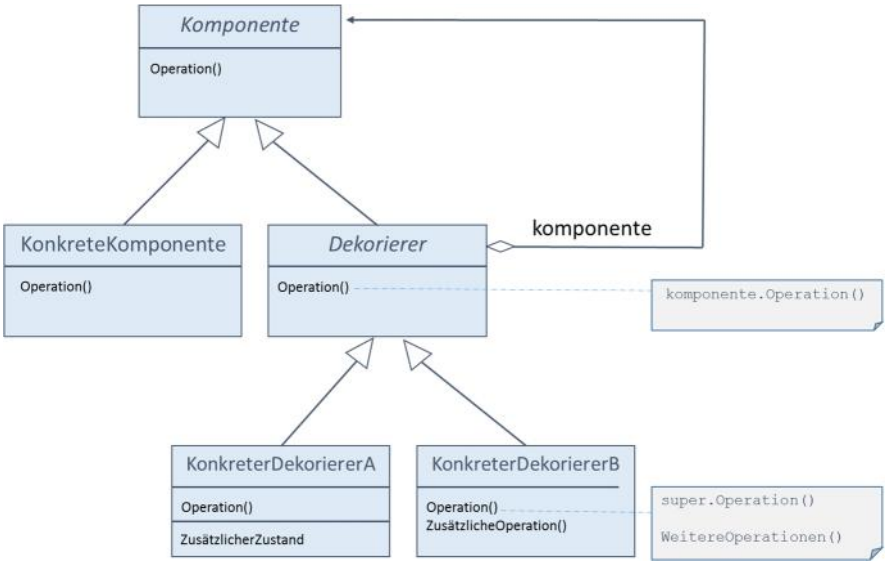
Hier sieht man noch einmal, dass die Klasse `CatSprite` nichts davon mitbekommt, dass sie mit zusätzlicher Funktion dekoriert wird. Auch die `JTurtleWorld` müssen wir nicht ändern. Sie erhält einfach eine anderes `SpriteInterface` Objekt, nämlich die dekorierte Katze statt der Katze selbst.

Das heißt wir müssen unsere bestehende Implementierung nicht anpassen und können diese dennoch um Funktionalität erweitern. Prima, Ziel erreicht!

2.10 Das Dekorierer Muster allgemein

Wir haben gesehen, dass sich die gleiche Lösungsstruktur in verschieden Fällen ähnlicher Situationen einsetzen lässt (sowohl zum Dekorieren von Büchern wie auch von Sprites).

Die allgemeine Klassenstruktur des Dekorierers sieht so aus:



Das Prinzip ist immer dasselbe. Sie haben eine abstrakte Schnittstelle (z.B. `Buchnutzung` oder `SpriteInterface`) und davon gibt es konkrete Umsetzungen (das `Buch`, die verschiedenen Sprites `Cat, Dog`…). Hinzu kommt ein Dekorierer, der die konkreten Umsetzungen umhüllen kann. Der Dekorierer besitzt jeweils eine `komponente`. Dies kann eine konkrete Komponente oder ein weiterer Dekorierer sein.

Die Kernfunktionalität der Komponenten liegt weiterhin in den konkreten Komponenten. Diese wird über verschiedene Methoden zur Verfügung gestellt, allgemein durch `Operation()` dargestellt. Da Dekorierer eine Komponente um Funktionalität erweitern, müssen sie auch die Kernfunktionalität der Komponente aufrufen, d.h. der Dekorierer muss in seiner `Operation()` zwingend `komponente.Operation()` aufrufen.

Die Erweiterung der Funktionalität geschieht in den konkreten Implementierungen des Dekorierers. Ein Dekorierer kann zusätzliche Operationen (Himmelsrichtung des Sprites ausgeben, Kekse essen) und zusätzliche Zustände (Streckenzug speichern, Weiterblättern zählen) abbilden.

Da sowohl die konkreten Komponenten wie auch die konkreten Dekorierer Implementierungen bzw. Ableitungen von `Komponente` sind, ist es für Client-Code (z.B. `JTurtleWorld`, `Leser`) transparent, ob es sich um eine konkrete oder (mehrfach) dekorierte Komponente handelt.

2.10.1 Anwendungskontext und Forces

Generell lassen sich folgende Anwendungskontexte und Einflussfaktoren für den Einsatz des Dekorierers nennen. Er erlaubt das dynamisches Hinzufügen (und Entfernen) von Funktionalität und Zuständigkeiten zu einzelnen Objekten. Er ermöglicht die transparente Funktionalitätserweiterung, so dass andere Objekte nicht davon betroffen sind. Er hilft bei der Vermeidung von Klassenexplosionen wenn viele verschiede Kombinationen von Funktionalität erforderlich sind und ist somit eine flexible Alternative zu Unterklassenbildung. Denn die Funktionalität wird Objekten hinzugefügt und nicht einer Klasse. Dies erlaubt die dynamische statt statischer Konfiguration von Funktionalität. Auch der Dekorierer hilft bei der Vermeidung von unnötiger Vererbung und Codeverdoppelung. Er vermeidet zudem funktionsüberladene Klassen auf hoher Ebene der Klassenhierarchie, da man Funktionalität quasi per Plug&Play hinzufügen kann.

2.10.2 Konsequenzen

Es gibt jedoch auch ein paar negative Konsequenzen auf die man achten muss. Zum einen haben Sie viele kleine Objekte zur Laufzeit und das Setup wird schnell unübersichtlich. Die Reihenfolge der Dekorationen kann eventuell zu verschiedenen Effekten führen, auch unerwünschten Nebeneffekten. Dies ist insbesondere der Fall wenn der Dekorierer das dekorierte Objekt verändert – dies sollte unterlassen werden! Zudem sollte man daran denken, dass die Dekoration und seine Komponente nicht identisch sind und somit ein Vergleich über die Objektidentität nicht möglich ist.

Beispiel: `catSprite` und `decoratedCatSprite` sind natürlich nicht identisch. `catSprite` wird vielleicht schon vorher erzeugt und an anderer Stelle referenziert (z.B. für die direkte Steuerung durch den Nutzer über Tastatureingaben). Der Liste `sprites` zum Zeichnen der Sprites wird dann die `decoratedSprite` übergeben. Jetzt kann man später natürlich nicht mehr über die `catSprite` Referenz das Element aus der `sprite` Liste entfernen, denn diese enthält nur eine Referenz auf das `decoratedSprite`.

2.11 Abstrakte Fabrik

Ein weiteres Beispiel für ein Entwurfsmuster ist die die Abstrakte Fabrik, auf die ich hier nicht ausführlich eingehen werde. Sie kann uns dabei helfen, die verschiedenen Sprite-Familien (`StupidSprite` mit nur einer Klasse, `Sprite` mit Unterklassen und `SmartSprite` mit Dekorierungen) nahtlos auszutauschen.

Die folgende Tabelle zeigt, dass für die Initialisierung der `JTurtleWorld`, also dem Erzeugen und Hinzufügen von Sprites, immer der gleiche Code anfällt, nur dass wir unterschiedliche Objekte erzeugen.

Initialisierung mit StupidSprites	Initialisierung mit Sprite-Unterklassen	Initialisierung mit SmartSprite

`StupidSprite gekko = new StupidSprite(StupidSprite.GEKKO);`
`StupidSprite turtle = new StupidSprite(StupidSprite.TURTLE);`
`StupidSprite cat = new StupidSprite(StupidSprite.CAT);`
`StupidSprite dog = new StupidSprite(StupidSprite.DOG , cat);`
 	
`gekko.setLocation(0, 100);`
`turtle.setLocation(300,300);`
`cat.setLocation(30,0);`
`dog.setLocation(500,500);`
 	
`gekko.setRotation(-30);`
`turtle.setRotation(350);`
`cat.setRotation(90);`
`dog.setRotation(180);`
`gekko.setWorldInfo(world);`
`turtle.setWorldInfo(world);`
`cat.setWorldInfo(world);`
`dog.setWorldInfo(world);`
 	
`sprites.add(gekko);`
`sprites.add(turtle);`
`sprites.add(cat);`
`sprites.add(dog);`
	`Sprite gekko = new Gekko();`
	
	`Sprite turtle = new Turtle();`
	
	`Sprite cat = new Cat();`
	
	`Sprite dog = new Dog(cat);`
	
	`gekko.setLocation(0, 100);`
	`turtle.setLocation(300,300);`
	`cat.setLocation(30,0);`
	`dog.setLocation(500,500);`
	
	`gekko.setRotation(-30);`
	`turtle.setRotation(350);`
	`cat.setRotation(90);`
	`dog.setRotation(180);`
	`gekko.setWorldInfo(world);`
	`turtle.setWorldInfo(world);`
	`cat.setWorldInfo(world);`
	`dog.setWorldInfo(world);`
	
	`sprites.add(gekko);`
	`sprites.add(turtle);`
	`sprites.add(cat);`
	`sprites.add(dog);`
		`SmartSprite gekko = new GekkoSprite();`
		
		`SmartSprite turtle = new TurtleSprite();`
		
		`SmartSprite cat = new CatSprite();`
		
		`SmartSprite dog = new DogSprite(cat);`
		
		`gekko.setLocation(0, 100);`
		`turtle.setLocation(300,300);`
		`cat.setLocation(30,0);`
		`dog.setLocation(500,500);`
		
		`gekko.setRotation(-30);`
		`turtle.setRotation(350);`
		`cat.setRotation(90);`
		`dog.setRotation(180);`
		`gekko.setWorldInfo(world);`
		`turtle.setWorldInfo(world);`
		`cat.setWorldInfo(world);`
		`dog.setWorldInfo(world);`
		
		`sprites.add(gekko);`
		`sprites.add(turtle);`
		`sprites.add(cat);`
		`sprites.add(dog);`

Wenn Sie mit mehreren alternativen Implementierungen (z.B. verschiedene Frameworks) arbeiten und diese bei Bedarf austauschen möchten, müssten Sie für jedes Framework die gleichen Initialisierungsaufrufe schreiben. Zudem müssen Sie dafür Sorge tragen, dass Sie nicht aus Versehen Implementierungen der verschiedenen Frameworks durcheinander mischen.

Die Abstrakte Fabrik hilft hier, da das Erstellen neuer Objekte über ein abstraktes Interface geschieht (die Abstrakte Fabrik). Die eigentliche Objekterzeugung (z.B. das Erstellen von `Gekko, Turtle , Cat` und `Dog`) wird an die Abstrakte Fabrik delegiert. Konkrete Fabriken implementieren die Schnittstelle und erzeugen Objekte der jeweiligen Produktfamilie (also einer unserer Sprite-Implementierungen).

Ein Beispiel dafür wäre die `AbstractSpriteFactory`:

```
abstract public class AbstractSpriteFactory {

  abstract public SpriteInterface createGekko () ;
  abstract public SpriteInterface createTurtle () ;
  abstract public SpriteInterface createCat();
  abstract public SpriteInterface createDog(SpriteInterface huntedSprite);

  …

}
```

Sie enthält Fabrikmethoden für die Erzeugung der verschiedenen Sprite-Arten. Eine konkrete Implementierung dieser `AbstractSpriteFactory` muss alle Fabrikmethoden implementieren und würde die Objekte der jeweiligen Implementierung erzeugen

Die `AnimalFactory` erzeugt z.B. Sprites durch die Unterklassen `Gekko, Turtle, Cat` und `Dog`. Diese implementieren das Bewegungsverhalten und das Kollisionsverhalten direkt.

Die `StupidSpriteFactory` erzeugt die verschiedenen Tierarten unter Verwendung derselben Klasse `StupidSprite` und konfiguriert diese über einen Parameter.

```
public class StupidSpriteFactory extends AbstractSpriteFactory {

      public SpriteInterface createGekko() {
            return new StupidSprite (StupidSprite.GEKKO);
      }

      public SpriteInterface createTurtle() {
            return new StupidSprite (StupidSprite.TURTLE);
      }

      public SpriteInterface createCat() {
            return new StupidSprite (StupidSprite.CAT);
      }

      public SpriteInterface createDog(SpriteInterface huntedSprite) {
            return new StupidSprite (StupidSprite.DOG , huntedSprite);
      }

}
```

Die `AnimalFactory` erzeugt Sprites durch Erzeugen der jeweiligen Unterklassen:

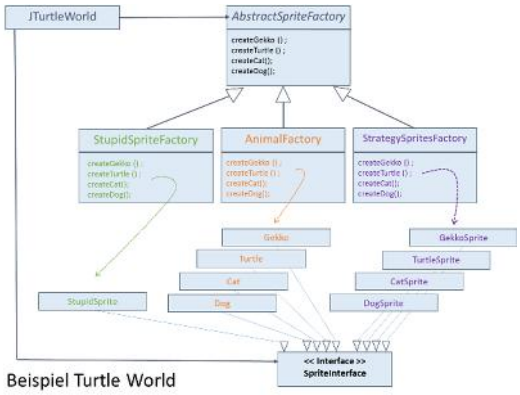
```
public class AnimalFactory extends AbstractSpriteFactory{

	public SpriteInterface createGekko() {
		return new Gekko();
	}

	public SpriteInterface createTurtle() {
		return new Turtle();
	}

	public SpriteInterface createCat() {
		return new Cat();
	}

	public SpriteInterface createDog(SpriteInterface huntedSprite) {
		return new Dog (huntedSprite);
	}

}
```

In allen Fällen liefern die Fabrikmethoden ein Objekt des Typs `SpriteInterface` zurück. So ist sichergestellt, dass die `JTurtleWorld` damit etwas anfangen kann.

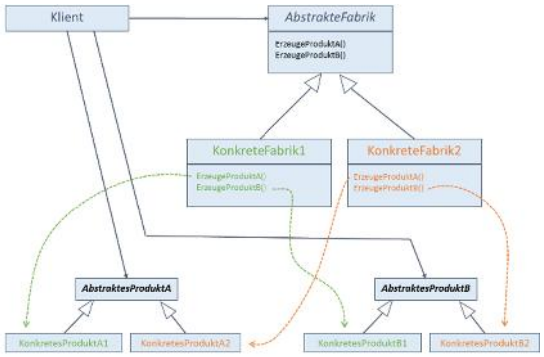
Der Vorteil ist nun, dass die `JTurtleWorld` nur das Interface der abstrakten Fabrik kennen muss, um einmal die Initialisierung der Sprites festzulegen. Aus welcher Sprite-Familie die Objekte erzeugt werden, entscheiden die Implementierungen der Abstrakten Fabrik:

```
AbstractSpriteFactory spriteFactory = new AnimalFactory();

SpriteInterface gekko = spriteFactory.createGekko(0, 100, -30, world);
SpriteInterface turtle = spriteFactory.createTurtle(300,300,350,world);
SpriteInterface cat = spriteFactory.createCat(30,0,90,world);
SpriteInterface dog = spriteFactory.createDog(500,500,180,world,sprite3);

sprites.add(gekko);
sprites.add(turtle);
sprites.add(cat);
sprites.add(dog);
```

Um jetzt zu einer anderen Sprite-Familie zu wechseln müsste man nur `new AnimalFactory()` austauschen, z.B. durch `new StrategySpritesFactory()`. Die Aufrufe zum Erstellen der Sprites bleiben unverändert. Wenn Sie nun eine noch bessere Sprite-Implementierung umsetzen wollten, dann müssten Sie keinen erneuten Initialisierungs-Code schreiben sondern würden einfach die Fabrik austauschen.



Beispiel Turtle World



Struktur-Diagramm des Abstrakte Fabrik Musters

Links: Klassendiagramm der AbstractSpriteFactory
Rechts: Allgemeines Strukturdiagramm einer Abstrakten Fabrik

Allgemein lässt sich für die Abstrakte Fabrik sagen, sie dient der Bereitstellung einer Schnittstelle zum Erzeugen von Objekten einer Familie verwandter oder voneinander abhängiger Objekte ohne bereits die konkreten Klassen zu spezifizieren. Sie erlaubt den Austausch von Frameworks und verschiedenen Implementierungen

Die Abstrakte Fabrik können Sie z.B. einsetzen zum Erzeugen von graphischen Oberflächen unabhängig vom konkreten Framework. Über die Abstrakte Fabrik rufen Sie eine Methode `erzeugeFenster()` oder `erzeugeButton()` auf. Je nach Framework oder Plattform wird eine andere Konkrete Fabrik verwendet. Unter Windows erzeugt `erzeugeFenster()` der `WindowsFabrik` ein Windows-Fenster, unter dem Mac Betriebssystem erzeugt `erzeugeFenster()` der `MacFabrik` ein Mac-Fenster. `Fenster` und `Button` sind ihre abstrakten Produkte. Je nach Plattform gibt es konkrete Produkte: `WindowsFenster`, `WindowsButtons` und `MacFenster`, `MacButtons`. Für jedes Produkt gibt es eine Fabrikmethode in der abstrakten Fabrik.

3 Weiter führende Quellen zu Entwurfsmustern

Man beachte: ich habe die Entwurfsmuster in den Beispielen hier nur angerissen. In Büchern finden Sie ausführlichere Beschreibungen. Das ist wie mit den eingangs erwähnten Elefanten. Sie können einen Elefanten in seinen Grundzügen beschreiben oder ein ganzes Buch über Elefanten verfassen. So ist es auch mit einem Entwurfsmuster: sie können es so wie hier skizzieren und anhand von Beispielen erklären. Eine umfassendere Beschreibung mit vielen weiteren wichtigen Details und weiteren Beispielen finden Sie aber an anderer Stelle. Hier lohnt sich ein Gang in die Bibliothek, dort gibt es zahlreiche Bücher über Entwurfsmuster!

Sie sollten es dagegen meiden, in diesem Fall auf die Wikipediaeinträge zurückzugreifen. Sie finden dort sehr viele Muster beschrieben, aber leider meistens nicht sehr gut. Eine nach wie vor sehr gute Lektüre ist das Standardwerk „Enwurfsmuster – Elemente wiederverwendbarer objektorientierter Software“ von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides („The Gang of Four“ oder GoF).

„The Gang of Four“ haben insgesamt 23 Entwurfsmuster veröffentlicht. Sie unterteilen die Entwurfsmuster in Verhaltensmuster (z.B. die Strategie), in Strukturmuster (z.B. der Dekorierer) und Erzeugungsmuster (z.B. Abstrakte Fabrik). Diese Kategorien sind hilfreich, inzwischen gibt es aber sehr viel mehr Entwurfsmuster als diese 23. So gibt es spezielle Muster für serviceorientierte Architekturen, Sicherheit, verteilte Systeme, Nebenläufigkeit, asynchrone Webkommunikation und viele, viele weitere Bereiche. Es gibt keine genauen Zahlen aber es dürften weit über 10.000 Muster

sein, die beschrieben worden sind. Natürlich muss man nicht alle Muster kennen! Aber es lohnt sich vor dem Start eines Projektes zu schauen, ob es speziell für den Aufgabenbereich bereits Mustersammlungen gibt. Denn wir haben gesehen, wie sich mit Entwurfsmuster Probleme bei der Gestaltung komplexer Architekturen lösen und umgehen lassen. Entwurfsmuster sind meistens dann nützlich wenn man viele Anforderungen zu berücksichtigen hat und für die Weiterentwicklung des Systems ein hohes Maß an Flexibilität benötigt wird. Entwurfsmuster sind nicht immer die beste Wahl! Aber sie helfen uns dabei, über mögliche Probleme zu reflektieren und bieten eine erprobte Lösung an. Deshalb sollte man möglichst viele Muster für das jeweilige Aufgabengebiet kennen.

Empfehlenswert sind die Bücher der Serie „Patterns Oriented Software Architecture“ (POSA). Zum Verstehen der Prinzipien eignen sich auch die comicartig gestalteten „Head First Design Patterns“ (auch in deutscher Sprache als „Entwurfsmuster von Kopf bis Fuß“). Sie gehen zwar nicht in die Tiefe wie GoF oder POSA, aber beinhalten viele anschauliche Beispiele.

Wer sich tiefer mit der Philosophie hinter dem Musteransatz beschäftigen möchte, sollte auch auf die ursprünglichen Arbeiten von Christopher Alexander zurückgreifen, insbesondere „A Pattern Language“ und „The Timeless Way of Building“. Beide Bücher haben nichts mit Softwarearchitektur zu tun, offenbaren aber sehr viel über das holistische Paradigma, das hinter den Entwurfsmustern steht. Darin geht es auch um ästhetische, humane und ethische Aspekte guter Gestaltung, den Prozess der dorthin führt, sowie das Zusammenspiel von Mustern, um ein wohlgeformtes Ganzes zu gestalten: „The structure of the language is created by the network of connections among individual patterns: and the language lives, or not, as a totality to the degree these patterns form a whole” (Alexander, 1977). Beide Bücher haben Teile der Softwarecommunity Ende der 1980er Jahren beeinflusst und dürften auch Ideen des agilen Designs mitgeprägt haben.

Mehr über Muster, Konferenzen, Bücher und Ressourcen finden Sie auch auf http://hillside.net/

4 Hinweise zur Beispielanwendung

Die Beispielanwendung teilt sich in fünf Pakete auf.

Die Umgebung

`swinggui`: Hier wird die Umgebung aufgebaut.

Die `main()` Methode findet sich in der Klasse App. Dort findet sich auch eine Methode `createAppWindow()`, in der Sie durch Ein- und Auskommentieren mit den verschiedenen Spritefamilien experimentieren können.

In diesem Paket befindet sich die Turtleumgebung in der Klasse `JTurtleWorld`. Alle von `JTurtleWorld` benötigten Schnittstellen sind ebenfalls hier zu finden: `SpriteInterface`, `Pen` und `AbstractSpriteFactory`.

Alternative Spritefamilien

`stupid`: Hier befindet sich die erste Implementierung der Spritefamilie in Form einer einzigen Klasse `StupidSprite`. Zudem gibt es hier eine konkrete Fabrik `StupidSpriteFactory` zur Erzeugung von Sprites dieser Familie.

`characters`: Hier befindet sich die zweite Implementierung der Spritefamilie mit einer Oberklasse `Sprite` und den Spezialisierungen `Gekko, Turtle, Cat, Dog`. Für das Erzeugen der verschiedenen Sprits steht ebenfalls eine `AnimalFactory` bereit.

`strategycharacters:` Die dritte Implementierung der Spritefamile mit Strategien. Auch hier gibt es eine Oberklasse SmartSprite und Spezialisierungen `GekkoSprite, TurtleSprite, CatSprite, DogSprite`. Die Spezialisierungen sind aber sehr viel flexibler (und übersichtlicher), da sie die Berechnung des nächsten Schritts sowie die Reaktion auf Kollisionen an Strategien delegieren. In den Konstruktoren der Sprites können Sie ändern welche Strategien eingesetzt werden. Es gibt jeweils zwei Strategien: Implementierungen von `StepStrategy` tragen die Namen `StepXXX`, Implementierungen von `OnCollisionStrategyInterface` tragen die Namen `OnCollissionXXX`. Auch hier gibt es mit `StrategySpritesFactory` eine konkrete Fabrik.

Dekorierer

`decorators:` In diesem Paket befinden sich das `SpriteDecoratorInterface` sowie zwei Implementierungen. `PenDecoration` merkt sich bei jedem Schritt die neue Position. Es implementiert zudem `swinggui.Pen`, und kann von der `JTurleWorld` zum Zeichnen von Streckenzügen genutzt werden. `LogDecoration` gibt auf der Standardausgabe wenn sich die Himmelsrichtung (Nord, Süd, West, Ost) des Sprites geändert hat.

Die Dekorierer funktionieren für alle drei Spritefamilien. Ein Beispiel für die Verwendung finden Sie in `JTurtleWorld.initBestSprites()`.

Sie können die Beispielanwendung hier herunterladen: http://www.kohls.de/lehre