5 Konzepte und Konstruktionsprinzipien für anpassbare Software

Softwareprodukte müssen in der Regel während ihrer Einsatzdauer häufiger als andere Produkte verändert werden. Die Gründe dafür sind vielfältig: zum Beispiel weil die Prozesse, deren Abwicklung sie unterstützen, sich verändern oder sie mit anderen neu hinzugekommenen Softwareprodukten integriert werden müssen. Es ist daher nahe liegend, dass seit Softwareprodukte entwickelt werden, versucht wird, diese so zu gestalten, dass sie auf einfache Weise an spezifische Ausprägungen oder Änderungen des Einsatzkontexts schnell und problemlos angepasst werden können. In diesem Abschnitt werden ausgewählte Konzepte und Konstruktionsprinzipien zur Verbesserung der Änderbarkeit und Anpassbarkeit von Softwaresystemen und -komponenten vorgestellt. Dabei wird insbesondere auf die Parametrierung von Software sowie auf objektorientierte Konstruktionsprinzipien eingegangen, weil diese nach dem derzeitigen Stand des Wissens die am häufigsten eingesetzten Techniken zur Verbesserung der Anpassbarkeit repräsentieren.

5.1 Parameter als Basis für anpassbare Software

Zusammen mit den Sprachkonstrukten Prozedur und Funktion wurde das Parameterkonzept eingeführt. Viele der heute verwendeten Programmiersprachen erlauben es, Eingangs-, Ausgangs- und Übergangsparameter von Prozeduren und Methoden sowie zusätzlich einen Rückgabeparameter bei Funktionen zu definieren. Der Sichtbarkeitsbereich von Variablen definiert, wo und für wen die Variablen sichtbar sind, d.h. wer auf sie zugreifen kann. Beispielsweise können auf eine Instanzvariable einer Klasse alle Methoden einer Klasse zugreifen. Die Instanzvariable kann also von den Methoden wie ein Übergangsparameter verwendet werden kann.

Im folgenden betrachten wir, wie das Parameterkonzept genutzt wird, um Softwaresysteme anzupassen, ohne deren Source-Code ändern zu müssen. Wir nehmen an, dass das Parameterkonzept und die damit verbundenen Begriffe bekannt sind, wie beispielsweise: aktuelle und formale Parameter, Call-by-Reference, Call-by-Value, Sichtbarkeitsbereich.

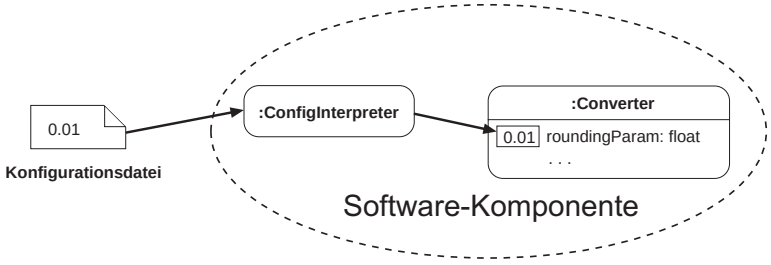

Abbildung 5.1. Anpassung über eine Konfigurationsdatei[1]

Als einfaches Beispiel betrachten wir die Rundung in einer Softwarekomponente zur Umrechnung von Währungen. Eine Klasse Converter dieser Softwarekomponente hat eine Instanzvariable roundingParam, über die definiert wird, auf wieviele Stellen gerundet wird: Beispielsweise drückt der Wert 0.01 aus, dass auf zwei

[1] Da der Begriff Konfigurationsdatei als Terminus Technicus häufig verwendet wird, verwenden wir ihn, obwohl der Begriff Parametrierungsdatei den Sachverhalt besser beschreiben würde.

Nachkommastellen gerundet wird; 0.0001 bewirkt ein Runden auf 4 Nachkommastellen. Der Wert 10 bedeutet, dass auf Zehnerstellen gerundet wird; der Wert 1 bewirkt ein Runden der Einerstelle. Um das Rundungsverhalten einer Converter-Instanz ändern zu können, ohne den Source-Code der Klasse zu ändern, kann eine Konfigurationsdatei vorgesehen werden, die von einem Programmteil ConfigInterpreter verarbeitet wird: Der Wert aus der Konfigurationsdatei wird gelesen und auf Gültigkeit geprüft. Dann wird die Instanzvariable roundingParam des Converter-Objektes entsprechend gesetzt. Abbildung 5.1 illustriert diesen Sachverhalt. Statt das Rundungsverhalten über eine Konfigurationsdatei zu steuern, kann dafür auch ein Interaktionsobjekt (z.B. ein Dialogfenster als Teil einer grafischen Benutzungsschnittstelle) vorgesehen werden: Der Benutzer der Softwarekomponente spezifiziert dann über ein Dialogfenster, wie gerundet werden soll. Eine GUI (Graphical User Interface)-Komponente zur Realisierung des Dialogfensters würde in diesem Fall die Rolle des ConfigInterpreters aus Abbildung 5.1 einnehmen. Statt des Zugriffs auf eine Konfigurationsdatei erfolgt eine Benutzereingabe. Am grundlegenden Prinzip der Parametrierung ändert sich nichts: Werte werden gelesen, interpretiert und verwendet, um das Verhalten von Softwarekomponenten anzupassen.

Im obigen Beispiel hat die Konfigurationsdatei eine einfache Struktur. Bei komplex strukturierten Konfigurationsdateien kann es sinnvoll sein, spezifische Editoren für die Eingabe der Konfiguration bereitzustellen. Als Beispiel betrachten wir die Konfiguration von GUI-Komponenten: Um die Änderung des Source-Codes zu vermeiden, wenn die Benutzungsschnittstelle eines Softwareproduktes in verschiedenen Sprachen verfügbar gemacht werden soll, eignen sich Konfigurationsdateien um die zur Festlegung der gewünschten Ausprägung eines GUI erforderlichen Parametrierungswerte bereitzustellen. Als Terminus technicus für GUI- Konfigurationsdatei wurde der Begriff Ressource-Datei eingeführt. Da eine GUI-Ressource-Datei eine komplexe Struktur aufweist, ist es nützlich Ressourcen-Editoren bereitzustellen, die dafür sorgen, dass Eingaben intuitiv, visuell und interaktiv erfolgen können und das Format der GUI-Beschreibung eingehalten wird. Abbildung 5.2 zeigt schematisch den um einen Editor erweiterten Konfigurationsteil einer Softwarekomponente.

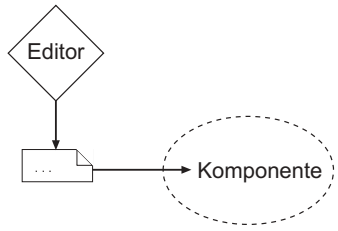

Abbildung 5.2. Editor zum Erstellen einer Konfigurationsdatei, die eine Software-Komponente parametriert

Callback-Style of Programming

In manchen Programmiersprachen wie z.B C, Pascal, Oberon, C# können nicht nur Daten-Objekte sondern auch Funktionen oder Prozeduren Parameter sein. Damit erweitert sich der Spielraum für Anpassungen. Statt Werte interpretieren zu müssen, um Verhalten von Komponenten zu ändern, wird die gewünschte Verhaltensweise in Form eines Prozedur- oder Funktionsparameters selbst bereitgestellt. Das wird mit dem Terminus Technicus Callback-Style of Programming beschrieben. Wenn das Laufzeitsystem das dynamische Laden und Linken von Programmteilen unterstützt, können Prozeduren und Funktionen zu einem in Ausführung befindlichen Softwaresystem hinzugefügt und von diesem verwendet werden.

In Programmiersprachen, die keine objektorientierten Sprachkonstrukte enthalten, ist die Benutzung von Funktions- und Prozedurparametern oft schwierig. Beispielsweise wird in ANSI-C ein Funktionsparameter wie folgt deklariert:

```
void doSomething( int (*compare)(void *, void *), void *elem1, void *elem2 )
```

Das bedeutet, dass die Funktion doSomething() als ersten Parameter einen formalen Funktionsparameter hat, dessen Name compare ist. Da aus Gründen der Verallgemeinerung die Typen der durch die Funktion zu vergleichenden zwei Datenobjekte nicht festgelegt werden sollen, wird der generische C-Typ void* verwendet. Soll nun eine Funktion stringCompare implementiert werden, die als aktueller Parameter an doSomething übergeben werden kann, ergibt sich das Probelm, dass dazu die Typprüfung durch einen sogenannten Type-Cast umgangen werden muss. stringCompare soll zwei Zeichenketten vom Typ char* vergleichen. Da compare als formale Funktionsparameter jedoch void* für die beiden zu vergleichenden Datenobjekte vorsieht, muss sich die Implementierung von stringCompare darauf verlassen, dass die aktuellen Parameter Zeichenketten sind:

```
int stringCompare(void *string1, void *string2) {
  return strcmp(
                    (char *)string1,
                    (char *)string2
            );
}
```

Ein Aufruf von doSomething() könnte beispielsweise folgende aktuellen Parameter mitgeben:

```
doSomething(stringCompare, “first“, “second“);
```

Abbildung 5.3 zeigt schematisch anhand des Beispiels das Konzept von Funktionsparametern, die von einer Funktion aufgerufen werden. Das erklärt, warum dafür der Begriff Callback-Style of Programming eingeführt wurde: Es lässt sich somit begrifflich unterscheiden, ob eine normale Funktion oder Prozedur aufgerufen wird (call) oder ob eine als Parameter übergebene Funktion oder Prozedur aufgerufen wird (callback).

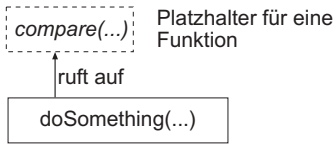

Abbildung 5.3. Beispiel für den Callback-Style of Programming

Objektorientierte Sprachen ermöglichen es, den Callback-Style of Programming mittels Vererbung und dynamischer Bindung syntaktisch eleganter und unter Beibehaltung der Typprüfung zu realisieren. Abbildung 5.4 zeigt das UML-Diagramm einer Klasse AnyClass, deren Methoden doSomething() und compare() den Funktionen des obigen Beispiels entsprechen. Durch Überschreiben der Methode compare() in einer Unterklasse von AnyClass wird quasi eine spezifische Funktionsimplementierung als Parameter von doSomething() – und an andere Methoden, die compare aufrufen – übergeben.

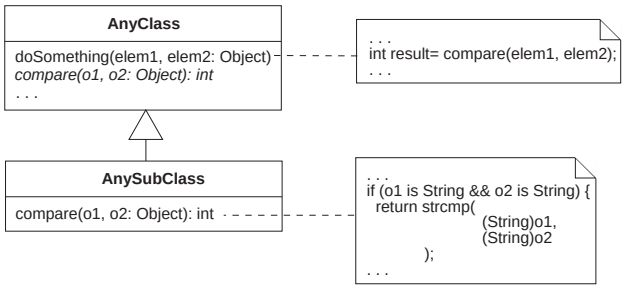

Abbildung 5.4. Callback-Style of Programming durch Vererbung und dynamische Bindung

Dem interessierten Leser wird empfohlen, die objektorientierte Programmiersprache Oberon zu studieren (Wirth und Reiser, 1992). Oberon ist eine Weiterentwicklung von Pascal, die durch erweiterbare Strukturen (Records) und Funktionsparameter zu einer objektorientierten Sprache wird. Das Studium von Oberon macht ersichtlich, wie eine einzige konzeptionelle Ergänzung einer konventionellen Sprache – die erweiterbaren Records – ausreicht, um objektorientiert programmieren zu können.

5.2 Produktfamilien und Entwurfsmuster

In der industriellen Fertigung sind Halbfertigfabrikate weit verbreitet. Beispielsweise sind die Modelle eines Automobilherstellers im Kern weitgehend identisch, sie unterscheiden sich z.B. in fertigungstechnischen Details oder in der Art und Weise, wie durch Kombination von Baugruppen fertige Produkte gebildet werden. Die einzelnen Baugruppen, wie Motoren und Antriebsstränge lassen sich zwischen den Modellen austauschen. Das trägt zu einer Flexibilisierung und Rationalisierung der Produktentwicklung bei.

An einem Beispiel aus der Alltagswelt lässt sich die Idee der halbfertigfabrikats-basierten Produktgestaltung ebenfalls skizzieren: Durch Aufstecken einer bestimmten Komponente wird aus dem Halbfertigfabrikat „elektischer Antrieb für Küchenmaschine“ beispielsweise ein Fruchtmixer oder ein Fleischwolf. Das Halbfertigfabrikat selbst besteht unter anderem aus einem Elektromotor, einem Schalter, einem Gehäuse und einer Antriebswelle, die die Schnittstelle zu den aufsteckbaren Komponenten bildet. Das Halbfertigfabrikat an sich ist für einen bestimmten Benutzer, z.B. einen Koch, nutzlos. Durch Anstecken einer passenden Komponente wird das Halbfertigfabrikat im Handumdrehen zu einer speziellen und nützlichen Küchenmaschine.

Abbildung 5.5. Konfigurierbare Küchenmaschine

Es versteht sich von selbst, dass man bestrebt ist auch die Herstellung von Softwareprodukten durch den Einsatz von Halbfertigfabrikaten zu flexibilisieren und zu rationalisieren. Die von objektorientierten Sprachen angebotenen Sprachkon-strukte eignen sich gut, um sogenannte Software-Halbfertigfabrikate zu realisieren.

In der Softwaretechnik sind objektorientierte Konstruktionsprinzipien zur Herstellung von Halbfertigfabrikaten erstmals bei der Entwicklung von Software zur Simulation diskreter Ereignisse eingesetzt worden; dafür wurde von Dahl und NigardXXXeine eigene Programmiersprache, Simula genannt, entwickelt [XXXRef]. Zu einer weit verbreiteten Technik wurde der Einsatz von Software-Halbfertigfabrikate als man begann, Softwareprodukte mit grafischen Benutzungsschnittstellen zu entwickleln und Objektsammlungen der Smalltalk-Klassenbibliothek trugen ebenfalls zur Verbreitung dieser Technik bei. Die Produkte der Firma SAP stellen Halbfertigfabrikate für betriebliche Anwendungen dar, die durch konventionelle Parametrierung (siehe Abschnitt 5.1) zu kundenspezifischen Produkten maßgeschneidert werden können (dies wird auch „customizing“ genannt). Fayad, Schmidt und Johnson (1999) zeigen typische Anwendungsbereiche von Software-Halbfertigfabrikaten auf.

In den vergangenen Jahren sind diverse Synonyme für den Begriff Software-Halbfertigfabrikat eingeführt worden, zum Beispiel Rahmenwerk als Übersetzung des englischen Begriffes Framework oder Produktfamilie. Da der Begriff Produktfamilie gut ausdrückt, dass aus einem Kern verschiedene Ausprägungen eines Softwaresystems ableitbar sind, verwenden wir bevorzugt diese Begriffsbezeichnung.

Wir definieren eine Produktfamilie als

ein Stück Software, das durch den

Callback-Style-of-Programming erweiterbar ist.

Die Definition schließt Software mit ein, die nach konventionellen, also nicht objektorientierten Prinzipien erstellt wird. Aufgrund der Probleme, mit denen man durch die mangelhafte Typprüfung bei konventionell realisierten Produktfamilien konfrontiert ist, klammern wir solcherart realisierte Produktfamilien aus. Wir behandeln hier ausschließlich Konstruktionsprinzipien für objektorientierte Produktfamilien.

Im folgenden Abschnitt wird skizziert, wie objektorientierte Produktfamilien (in objektorientierten Sprachen) auf Basis der sogenannten abstrakten Kopplung definiert werden. Danach werden die essentiellen Konstruktionsprinzipien für die Entwicklung objektorientierter Produktfamilien erläutert. Die unterschiedlichen Konstruktionsprinzipien erlauben verschiedene Freiheitsgrade bei der Anpassung einer Produktfamilie für einen bestimmten Zweck. Die Anwendung dieser Konstruktionsprinzipien fand auch ihren Niederschlag bei der Entwicklung von Entwurfsmustern durch Gamma et al. (1995). Eine schematische Darstellung der Zusammenhänge zwischen den Konstruktionsprinzipien und Entwurfsmustern soll das Verständnis der Entwurfsmuster erleichtern.

5.2.1 Objektorientierte Produktfamilien

Das Potential von objektorientierten Sprachen in Bezug auf Wiederverwendbarkeit wird oft nicht ausgenutzt, insbesondere wenn sie nur dazu benutzt werden eine modul-orientierte Architektur eines Softwaresystems zu implementierenwenn sie wie verbesserte Modul-orientierte Sprachen verwendet werden: Mit Klassen werden abstrakte Datentypen realisiert. Vererbung unterstützt die Wiederverwendung, da Klassen, die nicht genau passen, eventuell adaptiert werden können, ohne den Source-Code einer wiederverwendeten Klasse verändern zu müssen. Abbildung 5.6 zeigt schematisch, wie die Instanzen von Klassen in solcherart entworfenen Systemen miteinander gekoppelt sind. Als Beispiel betrachten wir ein Softwaresystem zur Steuerung eines autonom, also ohne Piloten fliegenden Helikopters. Das Softwaresystem gliedert sich in ein Steuerungssubsystem, das das Flugverhalten kontrolliert, und ein Navigationssubsystem. Abbildung 5.6 zeigt eine

grobgranulare Modularisierung des Navigationssubsystems, das auf oberster Ebene ein Navigationsmodul und ein Global Positioning System (GPS-)Modul umfasst. Navigationsmodul und GPS-Modul sind miteinander gekoppelt, indem Source-Codeteile des Navigationsmoduls Funktionalität des GPS-Moduls benutzen. Das GPS-Modul liefert aufgrund des empfangenen GPS-Signals die aktuelle Position des Helikopters und die vom GPS gelieferte Zeit. Das Navigationsmodul dient zur Koordination von Flugmustern, um ein bestimmtes Ziel anzufliegen. Beispielsweise soll der Helikopter einen vertikalen Kreis mit einem vorgegebenen Radius fliegen können. Der grau hinterlegte Kreis und der Pfeil im Navigationsmodul stellen schematisch den Source-Codeteil dar, der das Navigationsmodul mit dem GPS-Modul koppelt.

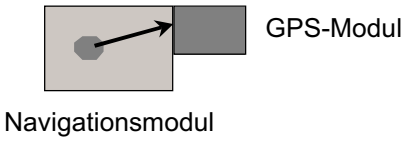

Abbildung 5.6. Kopplung des Navigationsmoduls mit dem GPS-Modul.

Angenommen in Europa wäre zusätzlich zu GPS bereits das Galileo-System verfügbar, und man will das GPS-Modul durch ein Galileo-Modul ersetzen, dann erfordert dies eine entsprechende Änderung des Source-Codes im Navigationsmodul (siehe Abbildung 5.7).

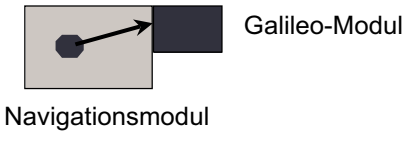

Abbildung 5.7. Kopplung des Navigationsmoduls mit dem Galileo-Modul.

Abstrakte Entitäten als Basis für Produktfamilien

Die Kopplung mit einem bestimmten Modul manifestiert sich hier in der Implementierung des Navigationsmoduls. Sie hängt also davon ab, welches Modul (GPS oder Galileo) verwendet wird. Das ist unerwünscht und es erhebt sich die Frage, wie eine derartige Abhängigkeit verhindert werden kann und wie das Navigationssystem zu einer Produktfamilie werden kann, in der das verwendete Positionierungssystem ausgetauscht wird, ohne dass das Navigationsmodul verändert werden muss? Eine Möglichkeit dazu ist die Definition einer abstrakten Entität.

Zur Beseitigung der oben beschriebenen Nachteile definieren wir eine abstrakte Entität, die wir PosSystem nennen. Objektorientierte Sprachen bieten dafür das Konstrukt der abstrakten Klasse an. Eine weitere, konzeptionell gleichwertige Möglichkeit für die Definition von abstrakten Entitäten bietet das Sprachkonstrukt Schnittstelle (Interface), das beispielsweise die Programmiersprachen Java und C# zur Verfügung stellen. Abbildung 5.8 zeigt schematisch die Umwandlung des Navigationsmoduls zu einer Produktfamilie, die mit jedem Positionierungssystem gekoppelt werden kann, das mit der abstrakten Klasse oder Schnittstelle PosSystem kompatibel ist. Der Polymorphismus der Objektorientierung wird ausgenutzt, um Stecker-kompatible Klassen zu definieren. Die dynamische Bindung ermöglicht es, Source-Code im Navigationsmodul zu schreiben, der nur festlegt, was im Prinzip zu tun ist (zum Beispiel die Abfrage der Positionskoordinaten). Wie die definierten Operationen tatsächlich ausgeführt werden, hängt vom dynamischen Typ des in den Navigationsmodul eingesteckten Objektes ab. Das Einstecken von Objekten, die zur

abstrakten Entität kompatibel sind, erfordert keine Änderung dieses Source-Codes im Navigationsmodul.

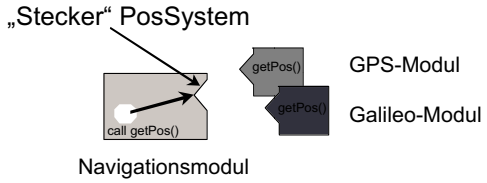

Abbildung 5.8. Abstrakte Kopplung des Navigationsmoduls mit PosSystem.

Abbildung 5.9 zeigt das entsprechende UML-Diagramm. Wir nehmen an, dass die abstrakte Entität PosSystem als abstrakte Klasse definiert wird, die in dieser vereinfachten Darstellung die abstrakten Methoden getPos() und getTime() anbietet. Eine Instanz der Klasse Navigation ist mit einem spezifischen Positionierungssystem abstrakt gekoppelt. Das heißt, dass die Implementierung in der Klasse Navigation auf dem statischen Typ[2] PosSystem basiert. Entsprechende Variablen sind vom Typ PosSystem in der Klasse Navigation deklariert, nicht von einem spezifischen Typ wie GPS oder Galileo. Der Aufruf von getPos() in Methoden der Klasse Navigation drückt aus, was im Prinzip zu tun ist, nämlich die Positionskoordinaten abfragen. Die Methode calcMove() beruht beispielsweise auf einer derartigen Abfrage. Wie die Funktionalität der Bestimmung der Positionskoordinaten zur Laufzeit bereitgestellt wird, hängt vom dynamischen Typ des eingesteckten PosSystem-Objekts ab. Eine Instanz von Galileo liefert auf andere Art die Positionskoordinaten wie eine Instanz der Klasse GPS.

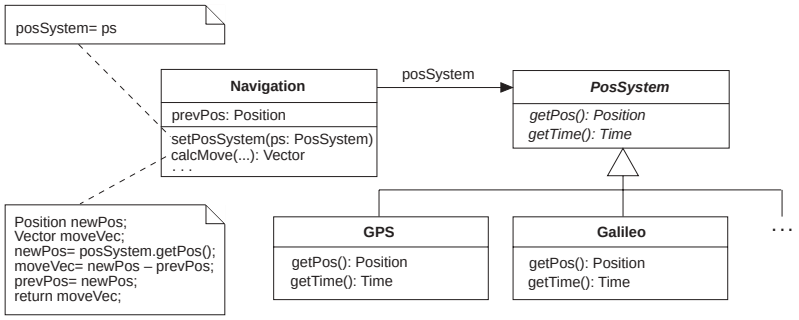

Abbildung 5.9. UML-Diagramm der objektorientierten Produktfamilie Navigationssystem

Durch den zusätzlichen Aufwand, eine abstrakte Entität PosSystem zu definieren, ist eine Produktfamilie entstanden, die den Navigationsmodul auf ein bestimmtes Positionierungssystem anpassen lässt, indem das entsprechende Modul eingesteckt wird. Die abstrakte Entität stellt quasi einen Stecker dar, der mit dem Stecker der eingangs erwähnten Küchenmaschine vergleichbar ist. Komponenten, die zur Produktfamilie gehören und zur Konfiguration des Halbfertigfabrikates verwendet werden, müssen kompatibel zum Stecker sein. Der Stecker definiert einen Standard und muss so gestaltet sein, dass er sowohl für das Halbfertigfabrikat als auch für die spezifischen Komponenten passend ist. Würde beispielsweise bei der Küchenmaschine die Kupplung der Antriebswelle zu schwach dimensioniert sein, würde eine Konfiguration als Fleischwolf vielleicht nicht funktionieren. Hätte man in

[2] Der statische Type einer Variable ist der Typ, der für die Variable in der Deklaration im statischen Programmtext festgelegt wurde. Der dynamische Typ einer Variable ist der Typ des Objektes, auf das die Variable zur Laufzeit zeigt. Eine Variable vom statischen Typ PosSystem kann zum Beispiel auf Instanzen jeder Unterklasse von PosSystem verweisen, da diese Klassen den Typ PosSystem erweitern.

der abstrakten Klasse PosSystem die Methode getTime() nicht vorgesehen, wären Geschwindigkeitsberechnungen in der Klasse Navigation nicht möglich.

Zusammenfassend gilt für abstrakte Entitäten, die als abstrakte Klassen realisiert sind, folgendes:

- Gemeinsame Instanzvariablen und Methoden von ähnlichen spezifischen Klassen sind in einer abstrakten Klasse zusammengefasst, von der die spezifischen Klassen abgeleitet sind.
- Abstrakte Klassen enthalten sowohl Methoden, die nur abstrakt den Namen und die Parameter festlegen, als auch Methoden, die implementiert werden können. Aufgrund der Vererbung verfügen die von der abstrakten Klasse abgeleiteten Klassen über die Methoden, wie sie in der abstrakten Klasse festgelegt wurden. Dadurch wird ein Standard festgelegt, der manchmal als Kontrakt, Signatur oder Protokoll bezeichnet wird. Die Instanzen von Unterklassen der abstrakten Klasse verstehen alle Methodenaufrufe, die in der abstrakten Klasse festgelegt sind.
- Die abstrakten Methoden sind in den spezifischen Unterklassen zu implementieren. Eventuell werden die bereits implementierten Methoden überschrieben oder Methoden hinzugefügt. Idealerweise belassen Unterklassen die Signatur, fügen also keine Methoden hinzu.
- Es macht keinen Sinn, Instanzen einer abstrakten Klasse zu erzeugen. In vielen objektorientierten Sprachen prüft das der Übersetzer. Abstrakte Klassen dienen dazu, dass andere Klassen vorweg auf Basis des Kontrakts der abstrakten Klasse implementiert werden können und somit objektorientierte Produktfamilien entstehen.

Definition einer abstrakten Entität als Schnittstelle (Interface)

Abstrakte Entitäten können auch mit dem Sprachkonstrukt Schnittstelle definiert werden. In statisch typisierten Sprachen wie C++ ist die Typhierarchie mit der Klassenhierarchie identisch: Nur Instanzen von Unterklassen einer Klasse A sind kompatibel mit dem Typ A. Schnittstellen ermöglichen das Aufbrechen dieser rigiden Beziehung.

Syntaktisch ist eine Schnittstelle, wie sie in Java und C# angeboten wird, einer abstrakten Klasse mit ausschließlich abstrakten Methoden ähnlich. Statt des Schlüsselwortes ‘abstract class‘ wird das Schlüsselwort ‘interface‘ verwendet. Es dürfen keine Methodenimplementierungen und keine Instanzvariablen angegeben werden.

Eine Schnittstelle ist vergleichbar mit einem Typ-Stempel: Jede Klasse, egal wo sie sich in der Klassenhierarchie befindet, kann mit dem Typ der Schnittstelle gestempelt werden, also den Typ der Schnittstelle annehmen, indem die Klasse die Schnittstelle implementiert. Das Implementieren einer Schnittstelle seitens einer Klasse bedeutet, dass sämtliche in der Schnittstelle enthaltenen Methoden in der Klasse zu implementieren sind. Eine Klasse kann beliebig viele Schnittstellen implementieren, also mit beliebig vielen Typen gestempelt werden.

Abbildung 5.10 zeigt, wie die Klassen A und SubR1 zusätzlich den Typ I erhalten, indem sie die Schnittstelle I implementieren. Schematisch ist das in Abbildung 5.10 durch den eingekreisten Buchstaben I neben dem jeweiligen Klassennamen dargestellt. Das soll das Stempeln der beiden Klassen mit dem Typ I andeuten. In der UML-Darstellung würde eine zusätzliche Vererbungsbeziehung zwischen der Schnittstelle I und der Klasse, die die Schnittstelle implementiert, gezeichnet werden, wobei statt einer durchgängigen Linie eine gestrichelte Linie die unterschiedliche Semantik ausdrückt.

Variablen vom statischen Typ einer Schnittstelle können auf Objekte von Klassen verweisen, die die Schnittstelle implementiert haben und somit zum Typ der

Schnittstelle kompatibel sind. Das folgende Code-Beispiel zeigt das anhand der Klassen A und SubR1, die die Schnittstelle I implementieren:

```
I iRef= new A();
. . .
iRef= new SubR1();
```

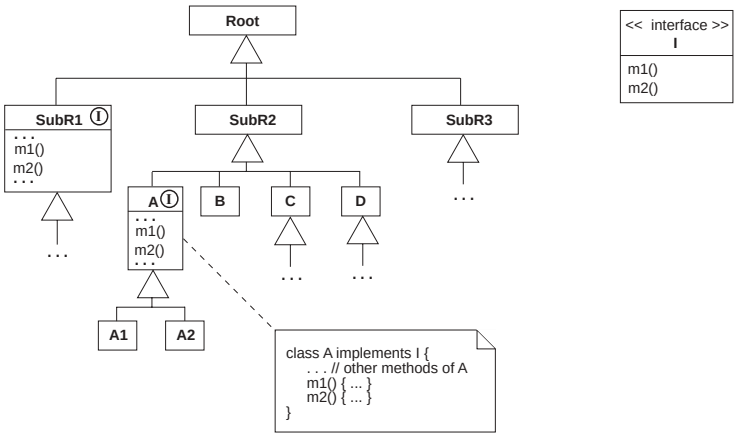

Abbildung 5.10. Das Schnittstellen-Konstrukt als Typ-Stempel

Abbildung 5.11 zeigt das UML-Diagramm des Navigationssystem, wenn eine Schnittstelle PosSystem anstatt einer abstrakten Klasse eingesetzt wird. Ein Vorteil kann sein, dass die Klassen GPS und Galileo nicht Unterklassen von einer abstrakten Klasse PosSystem sein müssen.

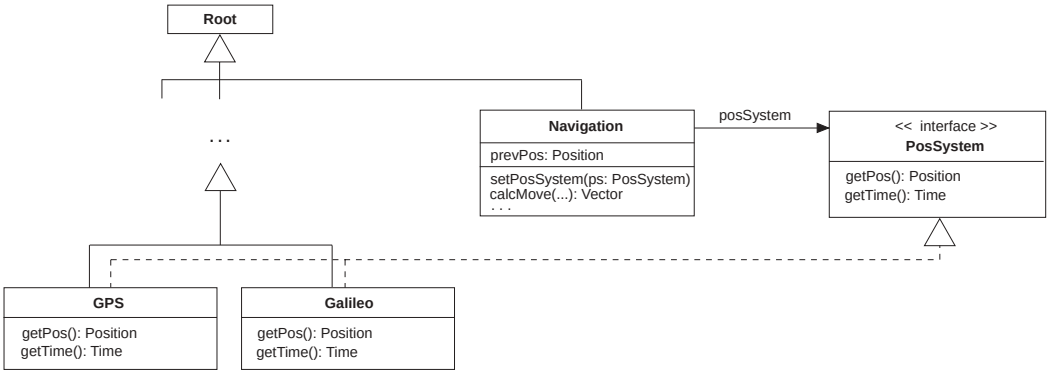

Abbildung 5.11. Die abstrakte Entität PosSystem als Schnittstelle

Die Definition von abstrakten Entitäten erfordert profundes Domänenwissen

In der Praxis gestaltet es sich als schwierig, die abstrakten Entitäten zu definieren. Es ist darauf zu achten, dass genügend gemeinsame Instanzvariablen und Methoden, die die Funktionalität festlegen, gefunden werden. Ansonsten können die Klassen, die mit den abstrakten Entitäten zu koppeln sind, nicht implementiert werden. Wenn zu spezifische Instanzvariablen und Methoden in eine abstrakte Entität gepackt werden, besteht die Gefahr, die Erweiterbarkeit der Produktfamilie einzuschränken.

Der Entwurf von abstrakten Entitäten erfordert hauptsächlich ein fundiertes Wissen über den jeweiligen Anwendungsbereich, das über jenes hinausgeht, was für spezifische Programmentwicklung nötig ist. Es muss verstanden werden, was charakteristisch für die spezifischen Entitäten ist. Meist müssen abstrakte Entitäten während diverser Anpassungen einer Produktfamilie reifen. Das ist ein Prozess, der

sich durchaus über Jahre hin erstrecken kann. Auch wenn eine abstrakte Entität nicht von Anfang an gut passt, ist es wichtig, sie vorzusehen, wenn an der entsprechenden Stelle des Softwaresystems die Erweiterbarkeit gegeben sein soll.

5.2.2 Ableitung von Konstruktionsprinzipien für objektorientierte Produktfamilien

Im vorigen Abschnitt wurde exemplarisch gezeigt, wie durch Ausnutzen der objektorientierten Konzepte Polymorphismus und dynamische Bindung Produktfamilien realisiert werden. Vom softwaretechnischen Standpunkt betrachtet ist es interessant, welche Konstruktionsprinzipien für objektorientierte Produktfamilien existieren. Wir erläutern zunächst, wie die Konstruktionsprinzipien systematisch ableitbar sind. Zu jedem der Konstruktionsprinzipien beschreiben wir den Grad an Flexibilität, der damit verbunden ist. Die Auswahl eines Konstruktionsprinzips hängt somit von der erforderlichen Flexibilität ab. Darauf wird in Abschnitt XXX eingegangen.

Template- und Hook-Methoden

Um die Konstruktionsprinzipien für objektorientierte Produktfamilien systematisch abzuleiten, ist es hilfreich, eine Kategorisierung von Methoden in sogenannte Template- und Hook-Methoden vorzunehmen: Wenn eine Methode eine andere Methode aufruft, nennen wir die aufrufende Methode die Template-Methode und die aufgerufene Methode die Hook-Methode. Es ist zu beachten, dass die hier angesprochene Template-Methode vom C++ Sprachkonstrukt Template verschieden ist. Abbildung 5.12 zeigt ein Beispiel, in dem die Methode m1() die Methode m2() aufruft, wobei beide Methoden in der gleichen Klasse sind. Die Methode m1() ist aufgrund der Aufrufbeziehung laut obiger Definition die Template-Methode, die Methode m2() die Hook-Methode. Die Buchstaben t und h links neben der Klasse markieren die beiden Methoden entsprechend. Es hängt lediglich von der Aufrufbeziehung ab, in welche Kategorie eine Methode fällt. Weder Größe noch Komplexität haben einen Einfluß auf die Kategorisierung.

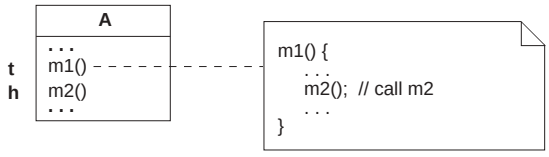

Abbildung 5.12. Template-Methode und Hook-Methode in einer Klasse

Abbildung 5.13 zeigt ein Beispiel, in dem die Template- und Hook-Methoden in verschiedenen Klassen sind. Dadurch ändert sich etwas an der Formulierung des Aufrufs, nicht an der Aufrufbeziehung.

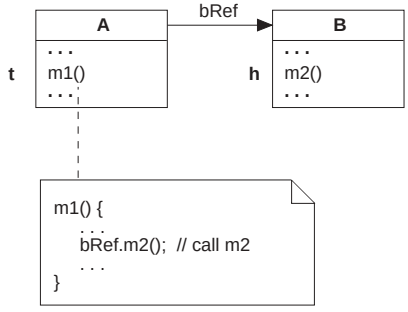

Abbildung 5.13. Template-Methode und Hook-Methode in verschiedenen Klassen

Was eine Template- und eine Hook-Methode ist, hängt vom Kontext ab. Wenn beispielsweise die Methode m2() die Methode m3() aufruft und wir diese beiden Methoden betrachten, ist m2() die Template-Methode und m3() die Hook-Methode (siehe Abbildung 5.14). Die Methode m2(), die bei Betrachtung von m1() und m2() die Hook-Methode ist, wird zur Template-Methode bei der Betrachtung von m2() und m3(). Diese Beobachtung hat eine wichtige Konsequenz, wenn wir die Konstruktionsprinzipien von objektorientierten Produktfamilien ableiten. Da die Ableitung der Konstruktionsprinzipien nur auf der Unterscheidung von Template-Methode und Hook-Methode beruht, sind die Konstruktionsprinzipien unabhängig vom Kontext anwendbar. Die betrachteten Methoden können welche mit wenigen Zeilen Code in einer simplen Klasse sein, oder komplexe Methoden in Klassen mit großem Funktionsumfang.

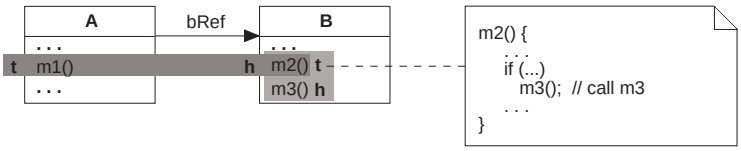

Abbildung 5.14. Eine Hook-Methode wird zu einer Template-Methode in einem anderen Kontext

Abschließend stellt sich die Frage, warum die Bezeichnungen Template und Hook gewählt wurden. Die Begriffe gehen auf die Smalltalk-Literatur zurück, in der erstmals von den beiden Methodenarten gesprochen wird. Da eine Template-Methode die entsprechende Hook-Methode aufruft, kann das Verhalten der Template-Methode durch Überschreiben der Hook-Methode verändert werden, ohne die gesamte Template-Methode ersetzen zu müssen. Die Hook-Methode hängt gleichsam wie ein Parameter in der Template-Methode.

Zur Ableitung der Konstruktionsprinzipien von objektorientierten Produktfamilien nennen wir die Template-Methode t() und die Hook-Methode h(). Die Klasse, die die Template-Methode enthält bezeichnen wir mit T. Die Klasse, die die Hook-Methode enthält bezeichnen wir mit H. Sämtliche Konstruktionsprinzipien werden abgeleitet, indem alle sinnvollen Kombinationen von Template-Methode und Hook-Methode in einer beziehungsweise in zwei Klassen betrachtet werden. Im einfachsten Fall sind beide Methoden in einer Klasse (siehe Abbildung 5.12 und Abbildung 5.15). Das ist das Unification-Konstruktionsprinzip. Wenn die beiden Methoden in verschiedenen Klassen sind, sprechen wir vom Separation-Konstruktionsprinzip. Bei den drei weiteren Konstruktionsprinzipien kommt die Vererbungsbeziehung in das Spiel. Diese Konstruktionsprinzipien werden weiter unten abgeleitet und erläutert.

5.2.3 Das Unification-Konstruktionsprinzip

Abbildung 5.15 zeigt das Unification-Konstruktionsprinzip. Da Template-Methode und Hook-Methode in einer Klasse TH sind, kann das Verhalten der Template-Methode durch Überschreiben der Hook-Methode in einer Unterklasse von TH angepasst werden.

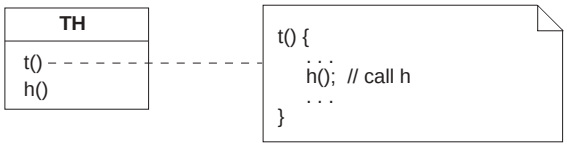

Abbildung 5.15. Das Unification-Konstruktionsprinzip

Die Anpassung erfordert, dass eine Instanz der Klasse SubTH generiert wird (siehe Abbildung 5.16). Meist ist es umständlich, zur Laufzeit eine Instanz der Klasse TH durch eine Instanz der Klasse SubTH auszutauschen, sodass eine Anpassung bei Anwendung des Unification-Konstruktionsprinzips typischerweise mit einem Neustart einer Anwendung verbunden ist.

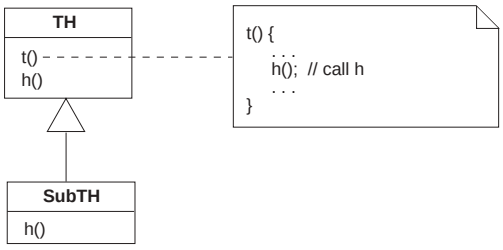

Abbildung 5.16. Anpassung der Hook-Methode

Abbildung 5.17 zeigt, wie das Navigationssubsystem bei Anwendung des Unification-Konstruktionsprinzips anpassbar ist. Die Anwendung des Unification-Konstruktionsprinzips hat einen Einfluß auf die Modularisierung. Es gibt keine explizite abstrakte Entität PosSystem. Stattdessen sind die beiden abstrakten Methoden getPos() und getTime() als Hook-Methoden in der abstrakten Klasse Navigation. Die Unterklassen GPSNavigation und GalileoNavigation überschreiben die beiden Hook-Methoden entsprechend, sodaß eine Instanz von GPSNavigation die Positionskoordinaten und die Zeit von einem GPS-Empfänger erhält. Analoges gilt für die Klasse GalileoNavigation. Da die Klasse Navigation abstrakt ist, muss eine Anpassung in einer Unterklasse erfolgen.

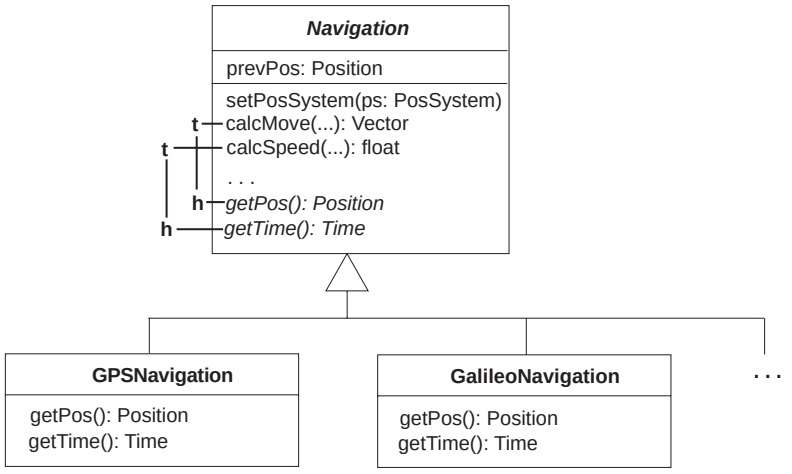

Abbildung 5.17. Anwendung des Unification-Konstruktionsprinzips am Beispiel Navigation

Abbildung 5.18 zeigt eine modifizierte Anwendung des Unification-Konstruktionsprinzips am Beispiel Navigationssubsystem. Statt die Hook-Methoden getPos() und getTime() in der Klasse Navigation als abstrakte Methoden zu definieren, sind sie konkret implementiert, zum Beispiel, um mit einem GPS-Empfänger zu arbeiten. Die Klasse ist eine konkrete Klasse und heißt dem entsprechend GPSNavigation. Wenn Galileo in Betrieb gehen wird, werden in einer Unterklasse GalileoNavigation die beiden Hook-Methoden überschrieben.

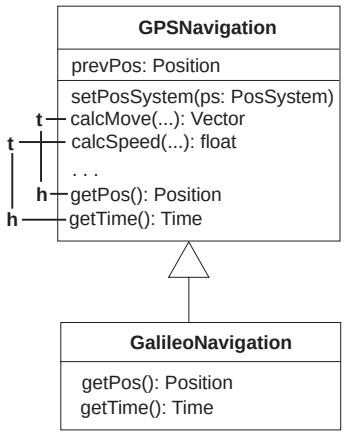

Abbildung 5.18. Konkrete statt abstrakter Hook-Methoden am Beispiel Navigation

5.2.4 Das Separation-Konstruktionsprinzip

Abbildung 5.19 zeigt das Separation-Konstruktionsprinzip, bei dem die Template-Methode und die Hook-Methode in separaten Klassen sind, die miteinander gekoppelt sind. Die Kopplung kann auf verschiedene Arten erfolgen. Meist wird in der T-Klasse eine Instanzvariable vom statischen Typ H definiert. In Abbildung 5.19 heißt die Instanzvariable hRef. Eine andere Möglichkeit der Kopplung ist, dass die Template-Methode die Referenz auf das H-Objekt als Parameter erhält. Das bedeutet, dass die Objekte nur für die Ausführung der Template-Methode verbunden sind. Schließlich können die T- und H-Instanzen über eine globale Variable, die ein H-Objekt referenziert, gekoppelt sein. Die Anzahl globaler Variablen soll klein gehalten werden, da sie die Strukturierung eines Softwaresystems verschlechtern (siehe Abschnitt XXX).

Das UML-Diagramm in Abbildung 5.19 macht keine Angabe über die Kardinalität der Beziehung zwischen T und H. Aus dem Source-Code des Kommentarblattes ist zu schließen, dass eine 1:1 Verbindung vorliegt, also ein T Objekt mit einem H Objekt gekoppelt ist. Die Eigenschaften des Separation-Konstruktionsprinzips gelten unabhängig davon, ob eine 1:1-Beziehung oder eine 1:N-Beziehung vorliegt. Bei einer 1:N-Beziehung ruft die Template-Methode durch Iterieren über alle referenzierten H-Objekte jeweils die Hook-Methode auf.

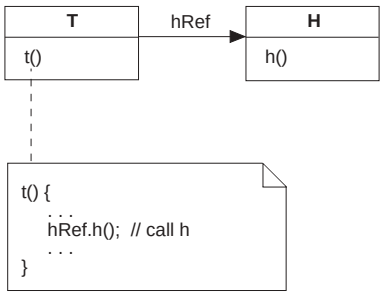

Abbildung 5.19. Das Separation-Konstruktionsprinzip

Das Verhalten der Template-Methode wird dadurch adaptiert, dass ein H-Objekt mit einer passenden Hook-Methode quasi in das Objekt, das die Template-Methode enthält, eingesteckt wird. Einstecken bedeutet, wenn die Objekte über eine Referenzvariable gekoppelt sind, dass das H-Objekt der Referenzvariable des T-Objektes zugewiesen wird. Im UML-Diagramm in Abbildung 5.19 ist für das

Einstecken eines H-Objektes die Methode defineH() vorgesehen. Da die Zuweisung einer Objektreferenz zu einer Variable zur Laufzeit möglich ist, kann das Verhalten der Template-Methode durch Komposition zur Laufzeit verändert werden. Wird das Separation-Konstruktionsprinzip im Gegensatz zum Unification-Prinzip eingesetzt, kann dieser höhere Grad an Flexibilität geboten werden.

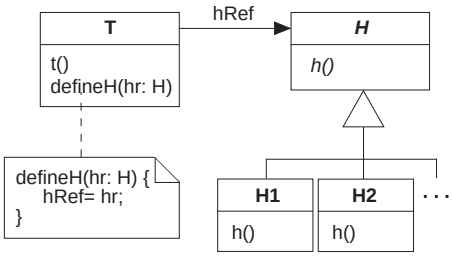

Abbildung 5.20. Anpassung durch Einstecken von H-Objekten zur Laufzeit.

Abbildung 5.20 zeigt exemplarisch die zwei Unterklassen H1 und H2 der abstrakten H-Klasse. Wir wollen zum Beispiel eine Instanz der Klasse H1 in ein T-Objekt stecken, um es zu konfigurieren. Der nachfolgende C# Code zeigt, wie die diese Konfiguration implementiert wird:

```
T sampleT= new T();
sampleT.defineH(new H1());
```

Abbildung 5.21 stellt schematisch die Anpassung von T durch Komposition des T-Objektes mit einem H1-Objekt dar. Der Aufruf von h() in der Template-Methode bleibt unverändert, unabhängig davon welches spezifische H-Objekt eingesteckt ist.

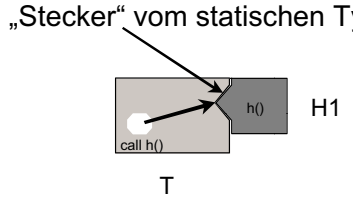

Abbildung 5.21. Komposition eines T-Objektes mit einem H1-Objekt

Im eingangs beschriebenen Entwurf des Navigationssystems ist das Separation-Konstruktionsprinzip angewendet worden. Abbildung 5.21 markiert die Template- und Hook-Methoden.

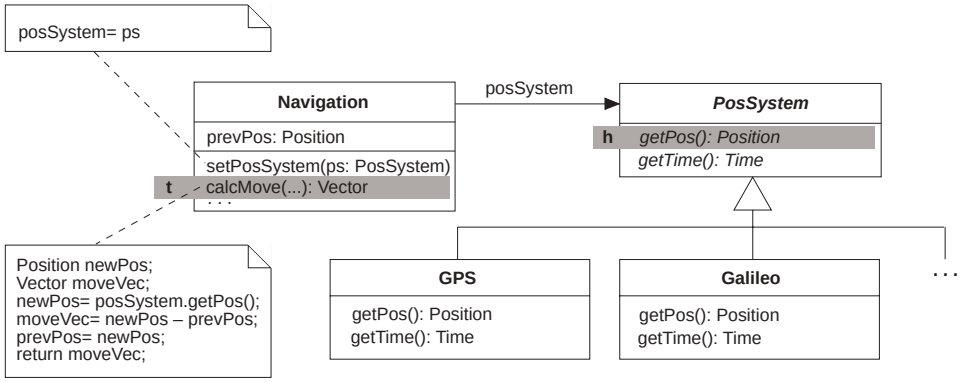

Abbildung 5.22. Anwendung des Separation-Konstruktionsprinzips beim Navigationssystem

Durch Komposition einer Instanz der Klasse Navigation mit einer Instanz der Klasse GPS entsteht ein GPS-basiertes Navigationssystem (Abbildung 5.23a). Analog entsteht durch Einstecken einer Instanz der Klasse Galileo in das Navigations-Objekt ein Galileo-basiertes Navigationssystem (Abbildung 5.23b). Zur Komposition wird die Methode setPosSystem() verwendet.

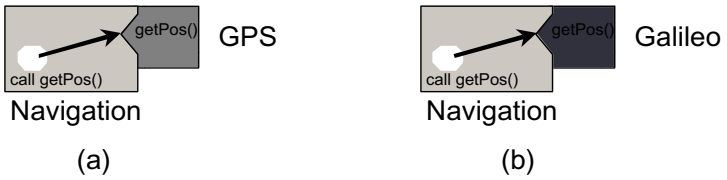

Abbildung 5.23. GPS-basiertes (a) und Galileo-basiertes (b) Navigationssystem

Erweiterung der Menge der einsteckbaren Komponenten zur Laufzeit

Ein Konfigurieren durch Einstecken von H-Objekten funktioniert gut, wenn die passenden Unterklassen von H vorhanden sind. Selbst wenn erst eine spezifische Unterklasse von H implementiert werden muss, kann durch Ausnutzen der Möglichkeiten, die sogenannte Meta-Informationen bieten, eine Konfiguration zur Laufzeit erfolgen.

Wir wollen beispielsweise ein Klasse UMTSTriangulation als Unterklasse von PosSystem implementieren, die eine Positionsbestimmung durch Triangulation im Mobilfunkstandard UMTS (Universal Mobile Telecommunications System) verwendet. Abbildung 5.24 zeigt das UML-Diagramm des Klassenbaums, der PosSystem als Wurzelklasse hat.

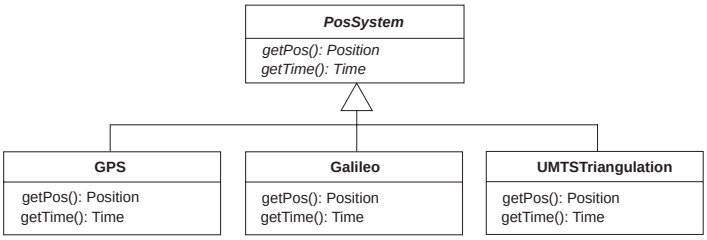

Abbildung 5.24. Unterklassen der abstrakten Klasse PosSystem

Eine Instanz von UMTSTriangulation soll, während das Navigationssystem in Betrieb ist, in das Navigations-Objekt gesteckt werden. Dazu muss das Laufzeitsystem das dynamische Laden und Linken unterstützen. Die entsprechende Funktionalität wird durch Bibliotheken und nicht durch objektorientierte Sprachkonstrukte bereit gestellt. Wir benutzen für das Beispiel die .NET-Klassenbibliothek und die Sprache C#. Im Prinzip würde folgender Code das Einstecken einer Instanz einer zusätzlich implementierten Klasse bewerkstelligen:

```
Navigation navigation= new Navigation(...);
String nameOfAddtlClass= “UMTSTriangulation“;
Object anObj= new nameOfAddtlClass;        // so nicht möglich
navigation.setPosSystem((PosSystem)anObj);
```

Die Implementierung geht davon aus, dass der Name der neuen Klasse auf geeignete Weise als Zeichenkette zur Verfügung gestellt wird. Zum Beispiel könnte der Klassenname über eine grafische Oberfläche in einer Dialogbox eingegeben werden. Mit dieser Information, nämlich wie die zusätzliche Klasse benannt ist, kann nun eine Instanz der neuen Klasse erzeugt werden. Da bei der Objektgenerierung

mit 'new' eine konkrete Klasse anzugeben ist, ist bei der Implementierung auf sogenannte Meta-Informationen zurückzugreifen. Der Begriff Meta-Informationen subsumiert Informationen über Objekte und Klassen zur Laufzeit. Grundlegende Beiträge auf diesem Gebiet haben die Arbeiten an der Smalltalk-Sprache und -Klassenbibliothek geliefert. Die Begriffe Meta-Level-Programmierung und reflexives Programmieren (von Reflection) werden synonym zum Programmieren mit Meta-Informationen verwendet. Konzeptionell besitzt jedes Objekt quasi ein Schattenobjekt, das eine Instanz der Klasse Class ist (siehe Abbildung 5.25).

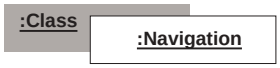

Abbildung 5.25. Jedes Schattenobjekt eines Objektes ist eine Class-Instanz

Die Klasse Class hat eine Reihe von Methoden, um Informationen über das zugehörige Objekt abzufragen. Beispielsweise kann abgefragt werden, welche Methoden und Instanzvariablen vorhanden sind. Falls es die Zugriffsrechte zulassen, kann zur Laufzeit ein Aufruf der Methoden beziehungsweise ein Auslesen des Werts einer Instanzvariablen erfolgen. Da wir diese Funktionalität für unsere Aufgabe nicht benötigen, gehen wir hier nicht näher auf die diversen Aspekte von Meta-Informationen ein. Wir verweisen auf die Fallstudie in Abschnitt XXX (ListX), in der der dynamische Zugriff auf Instanzvariablen zur Anwendung kommt.

Eine Methode der Klasse Class – newInstance() – erlaubt das Generieren eines Objektes. Dabei wird von der umgekehrten Situation ausgegangen, nämlich, dass es ursprünglich weder ein Schattenobjekt noch ein Objekt gibt. Zuerst wird das einem als Zeichenkette vorliegenden Klassennamen entsprechende Class-Objekt erzeugt (siehe Abbildung 5.26a). Die Klasse Class stellt dazu die statische Methode (= Klassenmethode; also eine Methode, die von der Klasse aufgerufen wird, nicht von einer Instanz) forName(String className) zur Verfügung. Der Rückgabeparameter von forName() ist eine Referenz auf die erzeugte Instanz der Klasse Class. Nun kann die erzeugte Instanz der Klasse Class durch den Methodenaufruf newInstance() das zugehörige Objekt erzeugen (siehe Abbildung 5.26b und Abbildung 5.26c). Der Rückgabeparameter von newInstance() ist die Referenz auf das entsprechende Objekt. Der Rückgabeparameter von newInstance() ist vom statischen Typ Object.

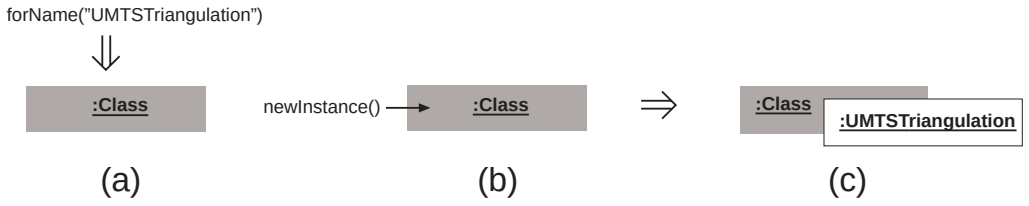

Abbildung 5.26. Erzeugen eines Objektes aus dem Schattenobjekt heraus

Der Source-Code zur Erzeugung einer Instanz von UMTSTriangulation hat zusätzlich Ausnahmebedingungen abzufangen. So könnte die Klasse, die als Zeichenkette angegeben ist, nicht existieren, wodurch ein dynamisches Laden unmöglich wird. Der nachfolgende Source-Code zeigt, wie eine Instanz der Klasse UMTSTriangulation zur Laufzeit erzeugt wird und in die Instanz der Klasse Navigation gesteckt wird.

```
Navigation navigation= new Navigation(...);
. . .
String nameOfAddtlClass= "UMTSTriangulation";
Type typeOfAddtlClass= Type.GetType(nameOfAddtlClass);
Object anObj;
```

```
PosSystem posSys;

if (typeOfAddtlClass != null) {
        anObj= Activator.CreateInstance(typeOfAddtlClass);
        if (anObj != null && anObj is PosSystem)
                posSystem= (PosSystem) anObj;
        else  ...  // error handling
}
navigation.setPosSystem(posSystem);
```

Abschließend ist anzumerken, dass die Flexibilität, ein Softwaresystem zur Laufzeit zu ändern, nur in wenigen Fällen notwendig ist. In unserem Beispiel würde es bedeuten, dass verschiedene, sogar neu implementierte Positionierungssysteme verwendet werden können, während der Helicopter fliegt. Das Separation-Konstruktionsprinzip bildet die Grundlage, um einen solchen Grad an Flexibilität zu gewähren. Nur wenn neu implementierte Komponenten zur Laufzeit eingesteckt werden sollen, ist es notwendig, die Implementierung wie oben beschrieben so zu gestalten, dass auch Klassen instantiiert werden können, deren Namen erst zur Laufzeit bekannt wird.

Allgemein ist zu beobachten, dass der Implementierungsaufwand und damit die Komplexität steigen, wenn mehr Flexibilität geboten wird. Der einfachste Entwurf auf Basis des Unification-Prinzips erfordert eine Klasse GPSNavigation. Um ein GPS-Navigationssystem auf Basis des Separation-Konstruktionsprinzips zu realisieren, sind zumindest zwei Klassen erforderlich: die Klasse Navigation und die mit ihr gekoppelte konkrete Klasse GPSPosSystem. Meist wird eine abstrakte Kopplung vorgesehen, sodass drei Klassen notwendig sind: die Klasse Navigation, die mit der abstrakten Klasse PosSystem gekoppelt ist. Die konkrete Klasse GPS als Unterklasse von PosSystem implementiert die Methoden für die GPS-Charkteristika. Weitere Komplexität kommt durch die dynamische Erweiterbarkeit der Menge der einsteckbaren Komponenten hinzu. Das betrifft sowohl die Implementierung der dynamischen Instantiierung als auch den Konfigurationsmechanismus. Zum Konfigurieren ist beispielsweise eine grafische Oberfläche zu entwickeln. Die einfache Form, über eine Dialogbox den Namen der einzusteckenden Klasse einzugeben, ist für Programmierer, nicht aber für andere Benutzer zumutbar. Stattdessen wäre eine geeignete Darstellung der Menge der einsteckbaren Komponenten samt kurzer Beschreibung der Eigenschaften notwendig, aus der der Benutzer die gewünschte Komponente auswählt und so das System konfiguriert.

Ein Entwurf wird also nicht besser, wenn mehr Flexibilität geboten wird. Das Ziel soll sein, das adäquate Maß an Flexibilität vorzusehen. Ein Entwurf, der möglichst flexibel gestaltet ist, also Flexibilität der Flexibilität wegen bietet, führt zu unnötiger Komplexität. Heuristiken zum Eruieren von Flexibilitätsanforderungen werden im Abschnitt XXX präsentiert.

5.2.5 Ableitung von Konstruktionsprinzipien mit rekursiven Charakteristika

Die Konstruktionsprinzipien von objektorientierten Produktfamilien lassen sich dadurch ableiten, dass alle sinnvollen Kombinationen von Template-Methode und Hook-Methode in einer oder zwei Klassen betrachtet werden. Was fehlt, sind die Kombinationen, in denen die Klasse mit der Template-Methode von der Klasse mit der Hook-Methode erbt. Daraus ergeben sich die drei in Abbildung 5.27 dargestellten Konstruktionsprinzipien: Composite, Decorator und Chain-Of-Responsibility. Auf den Umstand, dass die Namen der Template- und Hook-Methode gleich sind, wird weiter unten eingegangen. Die Namen der Konstruktionsprinzipien mit rekursiven

Charakteristika wurden von Gamma et al. (1995) in ihren Katalog von Entwurfsmustern vorgeschlagen.

Composite und Decorator unterscheiden sich in der Kardinalität der Beziehung zwischen der T- und der H-Klasse. Beim Composite-Konstruktionsprinzip kann ein T-Objekt eine beliebige Anzahl von H-Objekten referenzieren. Beim Decorator-Konstruktionsprinzip ist das genau ein H-Objekt. Trotz dieses scheinbar marginalen Details sind die Eigenschaften und Einsatzbereiche der beiden Konstruktionsprinzipien unterschiedlich, wie nachfolgend erläutert wird. Von den drei Konstruktionsprinzipien wird nach Einschätzung der Autoren das Composite weitaus am häufigsten verwendet. Das Decorator-Konstruktionsprinzip erfordert stringente Rahmenbedingungen, die eine Anwendung in vielen Fällen verhindern: die Methoden in der H-Klasse sollen eine Obermenge der Methoden in sämtlichen Unterklassen darstellen. Mit anderen Worten formuliert bedeutet das, dass Unterklassen die Signatur der H-Klasse nicht erweitern. Die H-Klasse muss also ausgereift sein und die Gemeinsamkeiten der Unterklassen herausfaktorisieren. Das Chain-Of-Responsibility-Konstruktionsprinzip stellt den degenerierten Fall dar, in dem die T- und H-Klasse zu einer Klasse vereint sind, aber dennoch die Referenzbeziehung erhalten geblieben ist. Das Chain-Of-Responsibility-Konstruktionsprinzip schätzen wir als unbedeutend ein und verweisen auf die Beschreibung dieses Konstruktionsprinzips in Gamma et al. (1995).

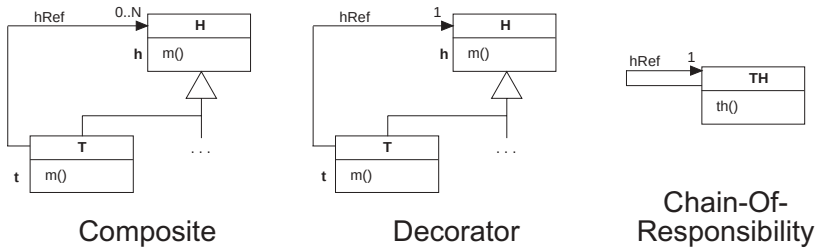

Abbildung 5.27. Konstruktionsprinzipien mit rekursiven Charakteristika

Damit sind die Konstruktionsprinzipien für objektorientierte Produktfamilien komplett. Der aufmerksame Leser könnte einwenden, dass die Kombinationen, in denen die Klasse mit der Hook-Methode von der Klasse mit der Template-Methode erbt, fehlen. Diese Kombinationen machen keinen Sinn, da sie dem Prinzip der Vererbung widersprechen, nämlich dem, dass Unterklassen deren Oberklassen verfeinern. Unterklassen sind somit konkreter als deren Oberklassen. Aufgrund der Definition, dass die Hook-Methode den Parameter der Template-Methode darstellt, ergibt es keinen Sinn, Kombinationen zu betrachten, in denen die Klasse mit der Hook-Methode (die allgemeinere Klasse) von der Klasse mit der konkreten Template-Methode (die speziellere Klasse) erbt.

Bevor die Konstruktionsprinzipien Composite und Decorator erläutert werden, gehen wir der Frage nach, was den rekursiven Charakter der drei Konstruktionsprinzipien ausmacht. Dadurch dass die T-Klasse von der H-Klasse erbt, ist der Typ T zum Typ H kompatibel. Die Beziehung „ein T-Objekt referenziert ein Objekt vom statischen Typ H“ kann daher auch aufgelöst werden in „ein T-Objekt referenziert ein T-Objekt“, da ein T-Objekt zum statischen Typ H kompatibel ist. Dadurch lassen sich rekursive Objektstrukturen komponieren. Beispielsweise erlaubt das Composite-Konstruktionsprinzip aufgrund der 0..N-Kardinalität die Komposition von T- und H-Objekten in einer Baumhierarchie.

Eine weitere Eigenschaft der drei Konstruktionsprinzipien resultiert aus der Vererbungsbeziehung: die Namen der Template-Methode und der Hook-Methode sind in den Konstruktionsprinzipien Composite und Decorator identisch. Lediglich die

Implementierung unterscheidet sich. Wie das Beispiel bei der nachfolgenden Erläuterung des Composite-Konstruktionsprinzips zeigt, erhält dadurch die Template-Methode einen rekursiven Charakter, obwohl sie keine rekursive Methode ist. Im Chain-Of-Responsibility-Konstruktionsprinzip werden die Template- und die Hook-Methoden zu einer einzigen Methode th() verschmolzen.

5.2.6 Das Composite-Konstruktionsprinzip

Abbildung 5.28 zeigt das Composite-Konstruktionsprinzip zusammen mit den Methoden addH() und removeH() zur Verwaltung der H-Objekte in einem T-Objekt. Statt der Instanzvariable hRef nennen wir die Instanzvariable zur Verwaltung von H-Objekten hList, um besser die 0..N-Kardinalität der Beziehung zwischen der T- und H-Klasse auszudrücken. Die Struktur einer Methode in der Klasse T ist dadurch charakterisiert, dass über die vom T-Objekt referenzierten H-Objekte iteriert wird und für jedes H-Objekt die jeweilige Methode aufgerufen wird. Das Kommentarblatt in Abbildung 5.28 zeigt diese Struktur am Beispiel der Template-Methode m().

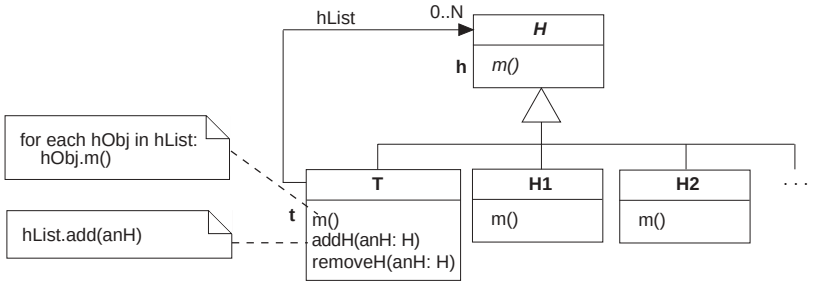

Abbildung 5.28. Struktur des Composite-Konstruktionsprinzips

Aufgrund der Eigenschaften des Composite-Konstruktionsprinzips können Objekthierarchien komponiert werden. Als Beispiel wird aus Instanzen der Klassen T, H1 und H2 die Objekthierarchie definiert, wie sie in Abbildung 5.29 dargestellt ist.

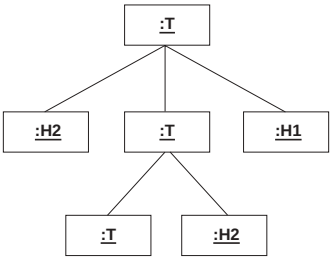

Abbildung 5.29. Ein Beispiel für eine Objekthierarchie

Der folgende C# Source-Code zeigt, wie die in Abbildung 5.29 gezeigte Objekthierarchie aufgebaut wird. Die Klasse T bietet die Infrastruktur, um die von einem T-Objekt aus referenzierten H-Objekte zu verwalten. Für den Aufbau der Objekthierarchie benötigen wir die Methode addH() zum Hinzufügen der H-Objekte:

```
T root= new T();
T subRoot= null;
root.addH(new H2());
root.addH(subRoot= new T());
root.addH(new H1());
subRoot.addH(new T());
subRoot.addH(new H2());
```

Aufgrund der Struktur der Template-Methode m(), die über alle enthaltenen H-Objekte iteriert und jeweils m() aufruft, kann die Objekthierarchie wie *ein* Objekt behandelt werden. Ein Aufruf von m() beim Wurzelobjekt der Objekthierarchie bewirkt, dass der Aufruf alle Objekte der Objekthierarchie erreicht. Auf den ersten Blick scheint die Template-Methode eine rekursive Methode zu sein:

```
void m() {
        for each hObj in hList
                hObj.m();
}
```

Es wird jedoch nicht die Methode m() selbst rekursiv aufgerufen. Stattdessen wird die Methode m() aufgerufen, wie sie im jeweiligen H-Objekt implementiert ist. Es ist möglich, dass dort die Implementierung identisch ist, weil das Objekt eine Instanz der Klasse T ist. Da jedes Objekt quasi die Methoden separat mit dabei hat, ist es dann dieselbe Methode, nicht die gleiche, die aufgerufen wird. Die Methode m() ist somit nicht rekursiv. Sie operiert aber auf einer rekursiven Datenstruktur.

Beispiel: Komposition von Flugmustern

Das Composite-Konstruktionsprinzip lässt sich beim Navigationssystem zur Komposition von Flugmustern anwenden. Ein Flugmuster besteht aus einzelnen Flugsegmenten im dreidimensionalen Raum, wie beispielsweise einer Linie, einem Kreis, oder einem Kreisbogen. Abbildung 5.30 zeigt das entsprechende UML-Diagramm.

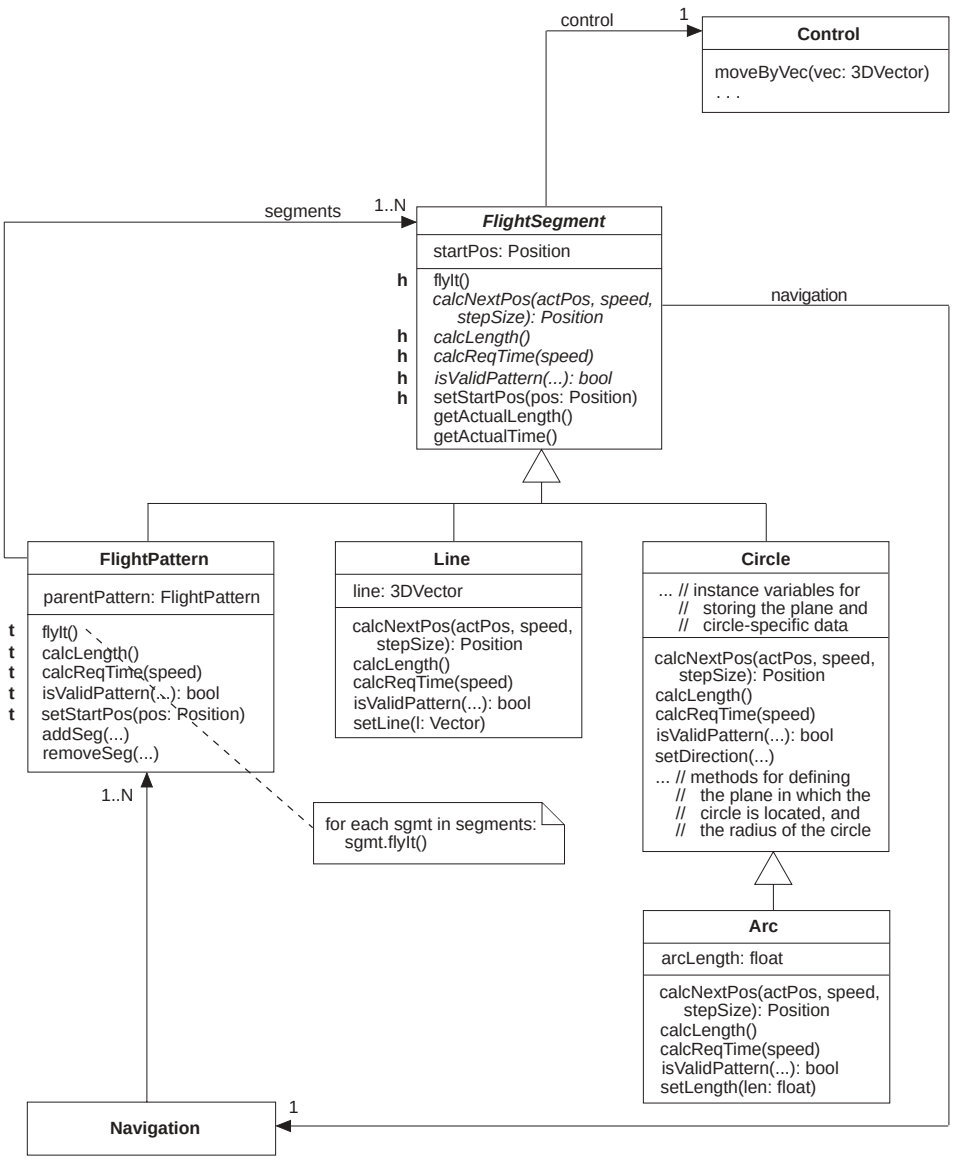

Abbildung 5.30. Anwendung des Composite-Konstruktionsprinzips bei Flugmustern

Ein Flugmuster setzt sich aus einzelnen Flugsegmenten zusammen. Bei der Definition eines Flugmusters wird dessen absolute Startposition spezifiziert. Beim Hinzufügen eines Flugsegments durch Aufruf der Methode addSeg() wird für die relative Beschreibung die entsprechende absolute Startposition berechnet. Die absoluten Startpositionen der einzelnen Segmente können somit nicht explizit in Instanzen der Klassen Line, Circle und Arc festgelegt werden. Das Flugmuster, das die Wurzel der Objekthierarchie darstellt, berechnet aufgrund dessen absoluter Startposition die jeweiligen absoluten Startpositionen der enthaltenen Flugsegmente und Flugmuster.

Um beispielsweise einen Achter in der horizontalen Ebene zu fliegen, der am Schnittpunkt der beiden Kreise begonnen wird, würde ein Flugmuster bestehend aus zwei Kreisen mit entgegengesetzter Flugrichtung definiert werden (siehe Abbildung 5.31).

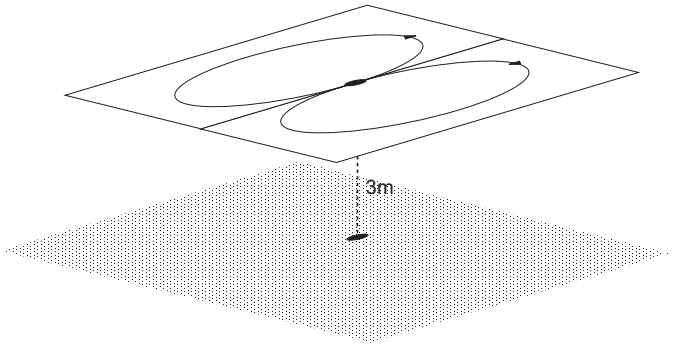

Abbildung 5.31. Ein 8er-Flugmuster in der horizontalen Ebene bestehend aus zwei Kreisen

Folgender Source-Code definiert das 8er-Flugmuster. Wir gehen davon aus, dass die Klasse Position die Angabe von Koordinaten sowohl in Längen- und Breitengraden als auch durch ein (x,y,z) Tupel in Metern zulässt. Die Position am Boden, über der der Helicopter das Flugmuster starten soll, sei mit (gL, gB) gegeben. Die Ebene, in der sich die Kreise befinden, wird als Pseudocode *horizontal plane* spezifiziert:

```
FlightPattern loop= new FlightPattern();
loop.setStartPos(new Position(gL, gB) + new Position(0, 0, 3));
loop.addSeg(new Circle (horizontal plane, 7, right));  // radius: 7 m; right dir.
loop.addSeg(new Circle (horizontal plane, 7, left));  // radius: 7 m; left dir.
```

Ein Implementierungsproblem ergibt sich daraus, dass nur für das Flugmuster, das die Wurzel der Objekthierarchie bildet, die absolute Startposition definiert werden darf, nicht für Flugmuster, die innerhalb der Hierarchie vorkommen. Beispielsweise kann das 8er-Flugmuster als Komponente in einem komplexeren Flugmuster verwendet werden (siehe Abbildung 5.32). Ein Aufruf der Methode setStartPos() muss zu einer Fehlermeldung führen, wenn ein Flugmuster eine Komponente einer Objekthierarchie ist, aber nicht deren Wurzel. Das lässt sich lösen, indem eine Referenz auf das verwaltende Flugmuster-Objekt in der Instanzvariable parentPattern gespeichert wird, wenn ein Flugmuster einem anderen Flugmuster hinzugefügt wird. Wird die Methode setStartPos() eines Flugmusters aufgerufen, dessen parentPattern ein Objekt referenziert, wird der Aufruf ignoriert und eine entsprechende Fehlermeldung ausgegeben.

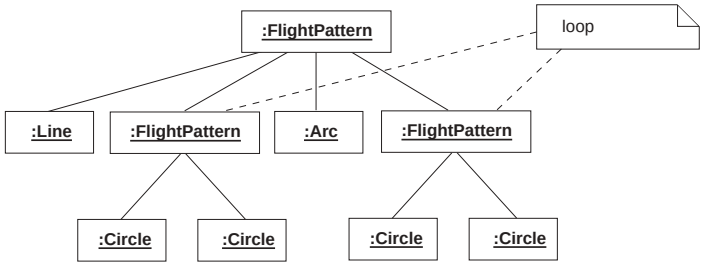

Abbildung 5.32. Komposition von einfachen Flugsegmenten und Flugmustern

Die Methode isValidPattern() prüft aufgrund der absoluten Koordinaten der Flugsegmente des Flugmusters und der Geländetopologie, ob das Flugmuster fliegbar ist, oder zu einer ungewünschten Berührung des Bodens führen würde. Beim 8er-Flugmuster würde beispielsweise eine zu starke Neigung der Ebenen dazu führen, dass ein Kreis sich mit dem Boden schneidet (siehe Abbildung 5.33). In diesem Fall liefert die Methode isValidPattern() den Wert False zurück.

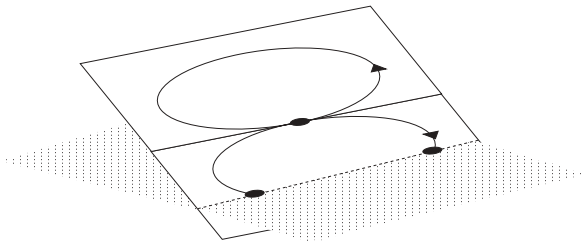

Abbildung 5.33. Ein Beispiel für ein Flugmuster, das zum Absturz führt

Die als Template-Methoden in der Klasse FlightPattern markierten Methoden folgen der Struktur der Template-Methoden im Composite-Konstruktionsprinzip. Als Beispiel ist in Abbildung 5.30 die Implementierung der Methode flyIt() auf einem Kommentarblatt notiert: Der Methodenaufruf wird an alle enthaltenen Flugsegmente weitergeleitet. Analog dazu sind die Methoden calcLength(), calcReqTime(), isValidPattern() und setStartPos() aufgebaut. Die Methoden calcLength() und calcReqTime() summieren die berechnete Länge bzw. Zeit für jedes im Flugmuster enthaltene Flugsegment auf. Die Methode isValidPattern() ruft für jedes im Flugmuster enthaltene Flugsegment diese Methode auf. Wenn alle Aufrufe den Wert True zurückliefern, ist das Flugmuster ohne Problem zu fliegen. Wenn ein oder mehrere Flugsegmente den Wert False zurückliefern, führt das Flugmuster zu einem Absturz. Die Methode setStartPos() geht von der Startposition des Flugmusters aus und berechnet für jedes Flugsegment die aus der Aneinanderreihung der Flugsegmente folgende Startposition.

Das Zusammenspiel der Methoden flyIt() und calcNextPos() in der Klasse Flugsegment erlaubt, die Methode flyIt() bereits in der Klasse FlightSegment für alle spezifischen Flugsegmente zu implementieren. Die Methode flyIt() ruft vom Steuerungssystem (Instanz der Klasse Control) die Methode moveByVec() auf und initiiert dadurch die nächste Flugbewegung indem der dreidimensionale, relative Flugvektor als Parameter dem Methodenaufruf mitgegeben wird. Der Flugvektor wird von der aktuellen Position und der nächsten erwünschten Position berechnet. Die aktuelle Position erhält ein Flugsegment durch Abfrage der Position beim Navigationssystem. Deshalb ist die Referenz zwischen den Klassen FlightSegment und Navigation im UML-Diagramm gezeichnet. Die erwünschte nächste Position wird in der Methode calcNextPos() berechnet. Die Methode flyIt() ruft dazu calcNextPos() periodisch auf. Je nach Flugsegment wird die nächste Position berechnet. Die Parameter von calcNextPos() werden durch Rückfrage beim Steuerungssystem ermittelt. Es hängt von der momentanen Fortbewegungsgeschwindigkeit und der Fortbewegungsrichtung des Helicopters ab, wie weit und in welche Richtung sich der Helicopter in einer bestimmten Zeiteinheit bewegen kann. Aus dem daraus resultierenden möglichen Flugverhalten wird der nächste Positionspunkt berechnet. Man beachte, dass im Kontext der Klasse FlightSegment die Methode flyIt() die Template-Methode und die Methode calcNextPos() die zugehörige Hook-Methode ist. Das Zusammenspiel der beiden Methoden beruht somit auf dem Unification-Konstruktionsprinzip. Die Unterklassen von FlightSegment überschreiben entsprechend der Eigenschaften der spezifischen Flugsegmente die Methode calcNextPos().

Abschließend betrachten wir die Vererbungsbeziehung zwischen den Klassen Arc und Circle: Die Klasse Arc ist eine Unterklasse von Circle, da ein Kreisbogen weitgehend wie ein Kreis spezifiziert und behandelt wird. Die Methoden, die sich vom Kreis unterscheiden, sind in der Klasse Arc angeführt.

Composite-Konstruktionsprinzip mit verschmolzener T- und H-Klasse

Eine Variante des Composite-Konstruktionsprinzips ist, dass die T- und H- Klasse verschmolzen sind (siehe Abbildung 5.34). Das ändert etwas an der Semantik der Komposition, nichts jedoch an den grundlegenden Eigenschaften. Sind T und H getrennt, hat die T-Klasse die Aufgabe, H-Objekte zu verwalten. Die Verwaltung ist von der Funktionalität der H-Klasse getrennt. Wenn die beiden Klassen verschmelzen, hat eine Instanz der gemeinsamen Klasse beide Aufgaben über. Im UML-Diagramm in Abbildung 5.34 ist die TH-Klasse eine abstrakte Klasse. Die Methoden zur Verwaltung der referenzierten TH-Objekte und die Methode m() sind hingegen implementiert. Die Entscheidung, TH als abstrakte Klasse zu modellieren, hat den Vorteil, die bei dieser Variante auch verschmolzene Template- und Hook-Methode m() besser zu strukturieren: In der abstrakten Klasse TH sorgt m() für das Iterieren über die Liste der TH-Objekte. In einer spezifischen Unterklasse wird die spezifische Funktionalität hinzugefügt. Das Iterieren wird aus der Oberklasse verwendet.

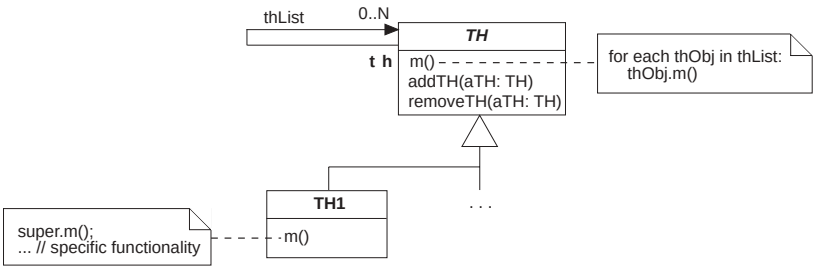

Abbildung 5.34. Eine Variante des Composite-Konstruktionsprinzips

Ein Beispiel, bei dem diese Modellierung angebracht ist, findet sich im Bereich der eingebetteten Dokumente: Ein Textdokument, das verschiedene andere Dokumente (wie zum Beispiel Zeichnungen, Sprache) enthält, verwaltet die enthaltenen Dokumente und bietet die Funktionalität zum Editieren an (siehe Abbildung 5.35).

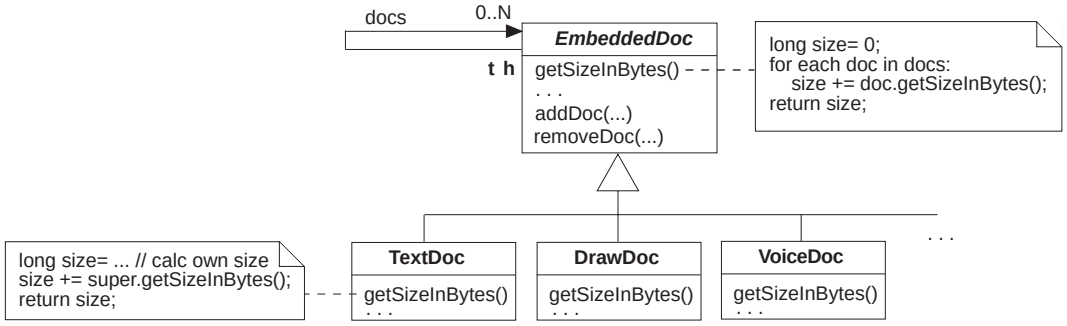

Abbildung 5.35. Verschmelzen von Verwaltung und Funktionalität bei eingebetteten Dokumenten

5.2.7 Das Decorator-Konstruktionsprinzip

Die Verwendung des Decorator-Konstruktionsprinzips ist gut zu verstehen, wenn folgendes Problem betrachtet wird: Wir nehmen an, dass in der Wurzelklasse eines umfangreichen Teilbaumes einer Klassenhierarchie die Implementierungen von Methoden zu ändern sind. Abbildung 5.36 zeigt exemplarisch einen Teilbaum einer Klassenhierarchie mit der Klasse A als Wurzel des Teilbaumes. Die Klasse A hat die Methoden m1() bis m10(). Davon soll die Implementierung der Methoden m3() und m7() geändert werden.

Wenn der Source-Code der Klassenbibliothek vorhanden ist, wird oft der vermeintlich einfachste Weg gegangen, den Source-Code der Klasse A entsprechend zu ändern. Wenn der Entwurf und die Implementierung nicht genau verstanden werden, können dadurch Seiteneffekte entstehen. Neue Versionen der Klassenbibliothek, die nicht so gut wie die ursprüngliche analysiert werden, bewirken möglicherweise Seiteneffekte in Zukunft.

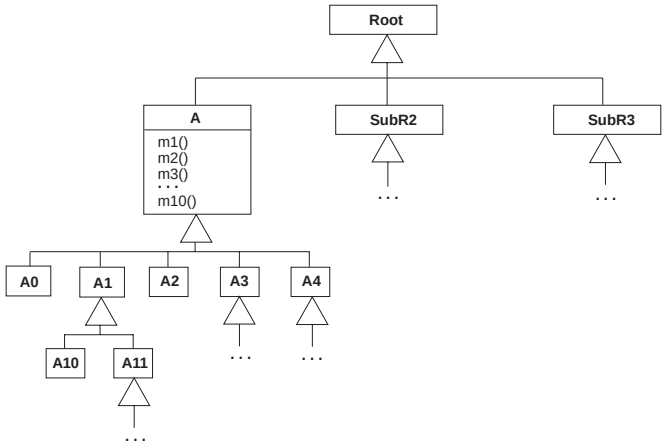

Abbildung 5.36. Beispiel eines Klassenbaums, in dem die Klasse A zu ändern ist

Wenn der Source-Code nicht vorhanden ist, kann die Klasse A nicht direkt geändert werden. Um den Effekt zu erzielen, dass Unterklassen von A das geänderte Verhalten zeigen, indem m3() und m7() modifiziert werden, kann Vererbung eingesetzt werden. Abbildung 5.37 zeigt, wie die Klasse A1 geändert wird, indem die Unterklasse A1m definiert wird, in der die genannten Methoden entsprechend implementiert sind. Um das für alle Klassen im Klassenbaum zu erreichen, sind durch aufwendige Unterklassenbildung die Anpassungen durchzuführen. Durch die Definition von A1m wird nur A1 modifiziert, nicht die Unterklassen A10, A11, etc. Die Modifikation von A10 erfordert eine Modifikation zum Beispiel in einer Unterklasse A10m. Analoges gilt für sämtliche Unterklassen von A.

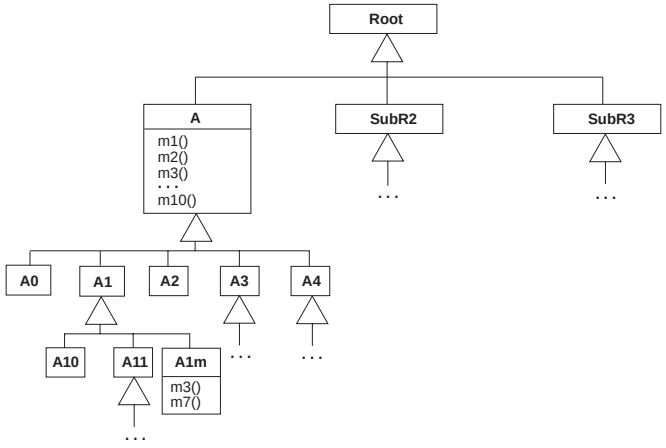

Abbildung 5.37. Änderung durch Vererbung

In Sprachen, die mehrfache Implementierungsvererbung bieten, kann die Anpassung effizienter durch sogenannte MixIn-Klassen erfolgen (siehe Abbildung 5.38). Allerdings ändert das nichts am Problem, dass für jede Klasse, deren Verhalten zu ändern ist, eine Unterklasse definiert werden muss. Die geänderte Implementierung der Methoden m3() und m7() ist lediglich besser in einer Klasse zusammengefasst.

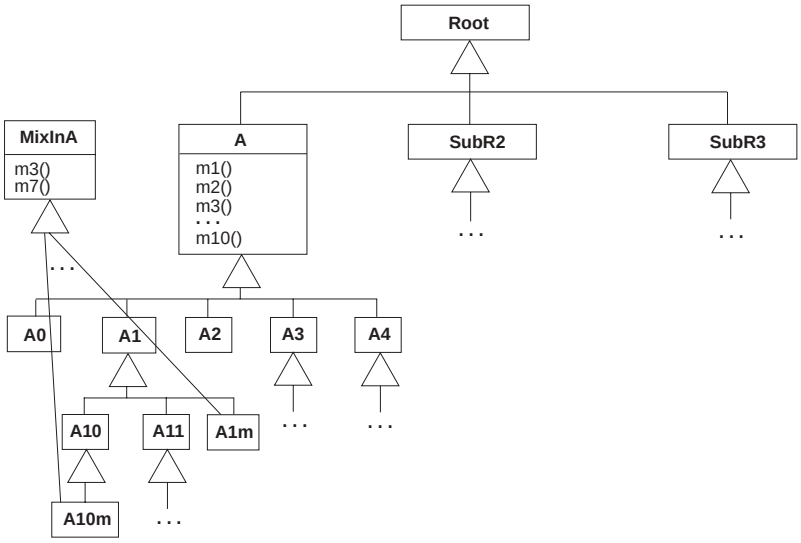

Abbildung 5.38. Änderung durch Mehrfachvererbung

Das Decorator-Konstruktionsprinzip löst das Problem durch Komposition von Objekten anstatt durch Vererbung. Abbildung 5.39a zeigt das grundlegende Konzept. Die Klasse WrapperOfA überschreibt alle Methoden von A indem sie den Methodenaufruf jeweils an das über die Instanzvariable wrappedA referenzierte Objekt weiterleitet. Die Methode setWrappedA() weist der Instanzvariable wrappedA die Referenz auf ein A-Objekt zu.

Da WrappedA eine Unterklasse von A ist, kann überall dort, wo ein statischer Typ A benötigt wird, eine Instanz der Klasse WrappedA verwendet werden. Da die Instanzvariable wrappedA vom statischen Typ A ist, kann sie auf jedes Objekt, das aus einer Unterklasse von A generiert wurde, verweisen.

Abbildung 5.39b zeigt schematisch, dass die Komposition einer Instanz von WrappedA mit einer Instanz von A40 eine Stufe der Indirektion bei Methodenaufrufen einführt: Jeder Methodenaufruf bei der WrappedA-Instanz wird an die A40-Instanz weitergeleitet. Die Komposition der beiden Objekte kann wie ein A-Objekt behandelt werden.

XXX Abbildungen (a) und (b) sind noch zu erstellen

Abbildung 5.39. Unveränderte Weiterleitung von Methodenaufrufen

Die Klasse WrappedA wird zu einem Decorator von A, wenn bestimmte Methodenaufrufe verändert werden. Es wird damit quasi ein Filter vor ein A-Objekt gestellt. Das UML-Diagramm in Abbildung 5.40 illustriert das Decorator-Konstruktionsprinzip. Wie beim Composite-Konstruktionsprinzip sind die Namen der Template-Methode und Hook-Methode jeweils identisch. Die Decorator-Klasse entspricht der H-Klasse, die Klasse A der T-Klasse.

XXX Abbildung ist noch zu erstellen

Abbildung 5.40. Eine Decorator-Klasse für A-Objekte

Eine Instanz des Filters/Decorators zusammen mit der Instanz einer Unterklasse von A kann von Klienten wie ein A-Objekt verwendet werden. Allerdings ist das Verhalten des A-Objektes modifiziert. Der Filter/Decorator kann einer Instanz einer beliebigen Unterklasse von A vorangestellt werden.

Da die Decorator-Klasse selbst eine Unterklasse von A ist, kann wiederum ein anderer Filter davor angebracht werden. Es können also beliebig viele Filter vor ein A-Objekt gestellt werden. Das ist analog zum Composite-Konstruktionsprinzip, wo ebenfalls durch die Methodenaufrufweiterleitung in den Template-Methoden viele Objekte wie ein Objekt behandelt werden.

Somit ist es gelungen, durch Komposition von Objekten das Verhalten von A-Objekten zu ändern. Dazu ist lediglich eine Decorator-Klasse zu definieren, anstatt mühsam Vererbung bei allen zu ändernden Klassen anzuwenden.

Wenn mehrere Decorator-Klassen notwendig sind, ist es sinnvoll, die Weiterleitung der Methodenaufrufe in einer gemeinsamen Oberklasse zu implementieren. Abbildung 5.41 zeigt, wie die Klasse Decorator sämtliche Methodenaufrufe an das von wrappedA referenzierte Objekt weiterleitet. Die Unterklasse Decorator1 überschreibt die m3() und m7() Methoden. Die Unterklasse Decorator2 überschreibt die Methode m9().

XXX Abbildung ist noch zu erstellen

Abbildung 5.41. Mehrere Decorator-Klassen für A-Objekte

In Abbildung 5.42 werden exemplarisch zwei Kompositionen herausgegriffen. Eine Instanz der Klasse A21 hat den Decorator1 davor angebracht (siehe Abbildung 5.42a). Vor der Instanz der Klasse A40 sind sowohl Decorator1 als auch Decorator2 (siehe Abbildung 5.42b).

XXX Abbildung ist noch zu erstellen

Abbildung 5.42. Ein (a) und zwei (b) Decorator-Objekte vor einem A-Objekt

Beispiel: Abrundung der Übergänge bei Flugmustern

Das Decorator-Konstruktionsprinzip ist gut anwendbar, um die Übergänge zwischen Flugsegmenten in einem Flugmuster harmonischer zu gestalten. Abbildung 5.43 zeigt, wie dadurch aus einem Flugmuster, in dem zwei Linien als Flugsegmente verwendet werden, ein harmonisches Flugmuster wird.

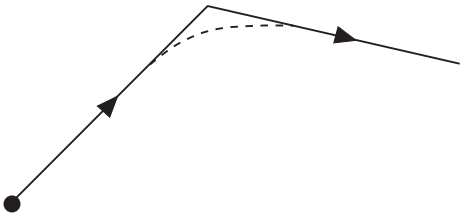

Abbildung 5.43. Abrundung eines aus zwei Linien bestehenden Flugmusters

Das Ziel ist, eine Klasse Smoother zu definieren, mit denen Flugsegmente dekoriert werden können. Wenn ein Flugsegment damit dekoriert ist, wird der Übergang zum nächsten Flugsegment, falls vorhanden, abgerundet.

Der Decorator Smoother modifiziert dazu die Methode calcNextPos(). Damit Smoother die Positionskoordinaten berechnen kann, sodass eine Abrundung erfolgt, muss das darauffolgende Flugsegment bekannt sein. Eine Smoother-Instanz benötigt dazu den Zugriff auf das Flugmuster-Objekt, in dem das abzurundende Flugsegment enthalten ist. Wir ändern daher den Entwurf dahingehend, dass die Instanzvariable parentPattern in der Klasse FlightPattern in die Klasse FlightSegment faktorisiert wird. Wenn ein Flugsegment zu einem Flugmuster hinzugefügt wird, wird die Instanzvariable entsprechend gesetzt. Weiters ist es notwendig, eine Methode getNextSeg() in der Klasse FlightPattern einzuführen, die das dem als Parameter

mitgegebenen Flugsegment nachfolgende Flugsement zurückgibt. Abbildung 5.44 zeigt die relevanten Ausschnitte der Klassenhierachie.

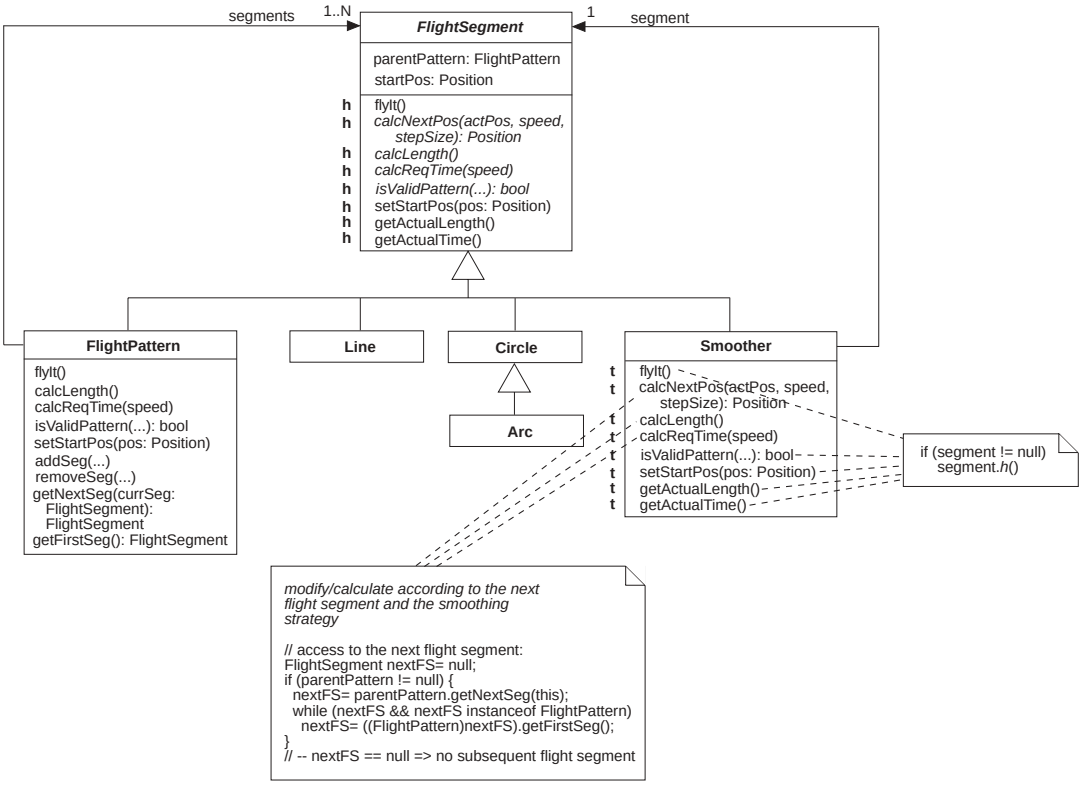

Abbildung 5.44. Die Klasse Smoother als Decorator von FlightSegment

Der folgende Source-Code stellt ein Flugmuster aus drei Geraden zusammen, wobei der Übergang zwischen den Geraden beim Fliegen des Musters abgerundet werden soll. Dazu werden die erste und die zweite Line-Instanz mit dem Smoother dekoriert. Wir nehmen an, dass die Klasse Smoother einen Konstruktor hat, dessen Parameter eine Variable vom statischen Typ FlightSegment ist. Die mitgegebene Objektreferenz wird der Instanzvariable segment zugewiesen.

```
FlightPattern triangle= new FlightPattern();
triangle.setStartPos(...);
triangle.addSeg(new Smoother(new Line(...)));
triangle.addSeg(new Smoother(new Line(...)));
triangle.addSeg(new Line(...));
```

Würde mit der dritten Geraden auch ein Smoother verbunden sein, hätte er keine Wirkung, da es kein nachfolgendes Flugsegment gibt.

Rahmenbedingungen für die Anwendung des Decorator-Konstruktionsprinzips

Das eingangs erwähnte Problem, das Verhalten an der Wurzel von Teilbäumen zu verändern, ist durch Anwendung des Decorator-Konstruktionsprinzips elegant lösbar. Die Anpassung durch Objektkomposition erfordert lediglich die Definition einer Klasse. Allerdings erfordert das Decorator-Konstruktionsprinzip eine Rahmenbedingung, die oft nicht gegeben ist: Die Signatur von H, also von der Wurzelklasse des Teilbaums, soll von den Unterklassen von H nicht erweitert werden. Der Grund dafür ist, dass zusätzliche Methoden in den Decorator-Klassen nicht berücksichtigt werden können. Die Schaffung dieser Rahmenbedingung erfordert ein profundes Herausfaktorisieren der Gemeinsamkeiten der Unterklassen

von H in die Wurzelklasse. Bei vielen Klassenbibliotheken ist die Rahmenbedingung nicht gegeben, wodurch die Anwendung des Decorator-Konstruktionsprinzips nicht zielführend ist.

Das Problem ist auch in unserem Beispiel gegeben, in dem die Klasse Smoother auf Basis des Decorator-Konstruktionsprinzips definiert ist: In den Klassen Line, Circle und Arc werden Methoden zur Signatur von FlightSegment hinzugefügt, die dazu dienen, die spezifischen Merkmale der jeweiligen Flugsegmente zu definieren, wie beispielsweise die Flugrichtung beim Kreis und beim Kreisbogen. Da es von der Modellierung nicht adäquat ist, diese Methoden in die Klasse FlightSegment zu verschieben, belassen wir den Entwurf wie er ist. Wir müssen aber die spezifischen Methoden für die erwähnten Objekte bei den Objekten selbst aufrufen, da sie von einer Smoother-Instanz nicht weitergeleitet werden können:

```
Circle circle= new Circle(...);
circle.setDirection(cRight);
Smoother smoother= new Smoother(circle);
```

Würde Smoother die erwähnte Rahmenbedingung erfüllen, könnte eine Smoother-Instanz wie jedes spezifische FlightSegment-Objekt behandelt werden:

```
Smoother smoother= new Smoother(new Circle(...));
smoother.setDirection(cRight);
```

Eine Möglichkeit, die Flugsegment-spezifischen Methoden zu eliminieren, ist, alle Merkmale nur über den Konstruktor der jeweiligen Klasse angeben zu lassen:

```
Smoother smoother= new Smoother(new Circle(..., cRight));
```

Die eingangs skizzierte Motivation zur Anwendung des Decorator-Konstruktionsprinzips geht davon aus, dass eine Klassenhierarchie im nachhinein modifiziert werden soll, ohne den Source-Code der Wurzelklasse eines Teilbaumes zu ändern. Das Decorator-Konstruktionsprinzip ist gut geeignet, um die Klassen nahe der Wurzel des Klassenbaumes leichtgewichtiger zu machen. Funktionalität, die nicht in allen Klassen benötigt wird, wird in Decorator-Klassen implementiert. Nur die Objekte, die die Funktionalität benötigen, erhalten sie durch Komposition mit der entsprechenden Decorator-Instanz. Das heißt, dass das Decorator-Konstruktionsprinzip von vorneherein beim Entwurf berücksichtigt wird.

Ein Beispiel für diese Anwendung des Decorator-Konstruktionsprinzips ist der Clipping-Mechanismus bei GUI-Bibliotheken. Nicht alle GUI-Elemente benötigen ein Clipping. Daher kann eine Decorator-Klasse Clipper eingeführt werden, statt den Clipping-Mechanismus in der Wurzel des Teilbaumes vorzusehen.

5.2.8 Zusammenfassung der Eigenschaften der Konstruktionsprinzipien

Die folgende Tabelle subsumiert die Eigenschaften der Konstruktionsprinzipien. Dabei wird das Chain-Of-Responsibility-Konstruktionsprinzip mit COR abgekürzt. Beim Unification-Konstruktionsprinzip sind die Template-Methode und Hook-Methode in einer Klasse, während die beiden Methoden bei den Separation-, Composite- und Decorator-Konstruktionsprinzipien getrennt sind. Wir nennen die Klasse, die die Template-Methode enthält, T und die Klasse, die die Hook-Methode enthält, H. Bei den Konstruktionsprinzipien Composite und Decorator erbt H von T. Im COR-Konstruktionsprinzip löst sich die Vererbungsbeziehung dadurch auf, dass T und H zu einer Klasse verschmelzen. Ebenso verschmelzen die Template-Methode und Hook-Methode.

Was die Anzahl der beteiligten Objekte betrifft, unterscheiden sich die Konstruktionsprinzipien wie folgt: Beim Unification-Konstruktionsprinzip wird eine Instanz jener Klasse generiert, die die passende Hook-Methode enthält. Beim Separation-Konstruktionsprinzip wird das T-Objekt durch ein H-Objekt konfiguriert.

Wenn die Beziehung zwischen T- und H-Klasse die Kardinalität 1..N oder 0..N hat, konfigurieren N(+1) H-Objekte das Verhalten des T-Objektes. Bei den Konstruktionsprinzipien mit rekursiven Charakteristika können beliebig viele Objekte komponiert werden, die wie ein Objekt behandelt werden. Beim Composite-Konstruktionsprinzip werden die Objekte in einer Baumhierarchie angeordnet. Beim Decorator-Konstruktionsprinzip sind die einem Objekt vorangestellten Decorator-Objekte in einer Liste angeordnet. Beim COR-Konstruktionsprinzip können beliebige Graphen definiert werden. Bei den drei zuletzt erwähnten Konstruktionsprinzipien ist es wichtig, dass keine zirkulären Objektbeziehungen innerhalb der Graphen definiert werden. Das würde zu einer Endlosschleife in der Template-Methode führen.

Schließlich unterscheiden sich die Konstruktionsprinzipien dadurch, wie die Anpassung erfolgt. Bei allen Konstruktionsprinzipien muss die Hook-Methode in Unterklassen der H-Klasse entsprechend überschrieben sein. Vererbung ist somit Voraussetzung für eine Anpassung. Bei den Konstruktionsprinzipien Separation, Composite, Decorator und COR erfolgt die Anpassung des Verhaltens des T-Objektes durch Komposition mit einem oder beliebig vielen H-Objekten.

	Unification	**Separation**	**Composite**	**Decorator**	**COR**
Template- und Hook-Methoden	t() und h() in einer Klasse	t() und h() in getrennten Klassen			t() = h()
			Namen von t() und h() gleich		
			H erbt von T		
					T = H
Anzahl der beteiligten Objekte	1	1 (T) + 1 (H) oder 1 (T) + N (H)	$N \rightarrow 1$		
Anpassung	durch Instanziierung der entsprechenden Klasse	durch Komposition (bei Bedarf zur Laufzeit)			

5.2.9 Die Anwendung von Konstruktionsprinzipien bei Entwurfsmustern

Mit Entwurfsmustern (im Englischen als Design Patterns bezeichnet) meinen wir die von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (Gamma et al. 1995) beschriebenen 23 Entwurfsmuster. In den 90er-Jahren sind zahlreiche Publikationen über Entwurfsmuster entstanden und diverse Konferenzen abgehalten worden. Dadurch sind keine breit akzeptierten Entwurfsmuster beschrieben worden, die über die 23 hinausgehen.

Im folgenden betrachten wir den Zusammenhang zwischen den fünf im vorigen Abschnitt beschriebenen Konstruktionsprinzipien für objektorientierte Produktfamilien und den Entwurfsmustern. Wir nehmen dabei an, dass die Entwurfsmuster dem Leser bekannt sind.

Mehr als die Hälfte der 23 Entwurfsmuster beschreibt, wie objektorientierte Produktfamilien entworfen und implementiert werden. Abbildung 5.45 zeigt im Überblick die 14 Entwurfsmuster, die auf einem der Konstruktionsprinzipien für objektorientierte Produktfamilien beruhen. Unten greifen wir zwei Entwurfsmuster heraus, um die Zusammenhänge im Detail zu erläutern. Die drei Konstruktionsprinzipien mit rekursiven Charakteristika decken sich mit drei Entwurfsmustern. Daher sind sie gleich benannt wie die Entwurfsmuster.

Die übrigen 9 der 23 Entwurfsmuster behandeln diverse Aspekte der Softwareentwicklung, die zum Teil bei Produktfamilien irrelevant sind oder allgemein interessant sind, unabhängig davon ob eine Produktfamilie entwickelt wird. Beispielsweise geht das Facade-Entwurfsmuster auf die Modularisierung von Software ein. Im Singleton-Konstruktionsprinzip wird beschrieben, wie in Sprachen wie C++ sichergestellt wird, dass nur exakt eine Instanz einer Klasse generiert werden kann. Da diese 9 Entwurfsmuster nicht direkt zur Entwicklung von Produktfamilien beitragen, verweisen wir den interessierten Leser auf die Beschreibung in Gamma et al. (1995).

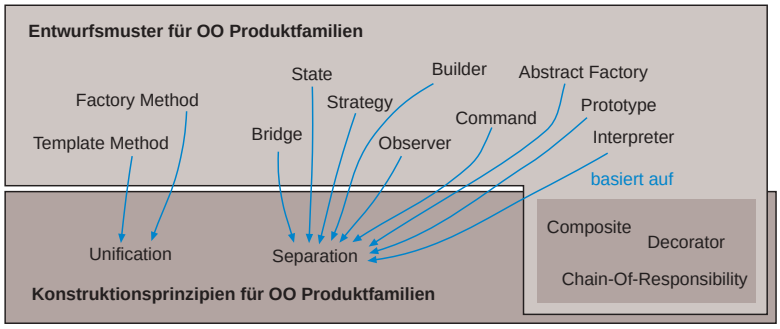

Abbildung 5.45. Zusammenhang zwischen Entwurfsmustern und Konstruktionsprinzipien

Im folgenden greifen wir die Entwurfsmuster Factory Method (Fabriksmethode in der deutschen Übersetzung der Entwurfsmuster) und Abstract Factory heraus, um die Zusammenhänge mit den Konstruktionsprinzipien herauszuarbeiten. In Abbildung 5.46 werden durch Annotation der Struktur des Entwurfsmusters Factory Method die Template- und Hook-Methoden markiert. Die Methode anOperation() der Klasse Creator entspricht der Template-Methode, die Methode factoryMethod() der Hook-Methode. Da Template-Methode und Hook-Methode in einer Klasse sind, liegt dem Entwurfsmuster das Unification-Konstruktionsprinzip zugrunde.

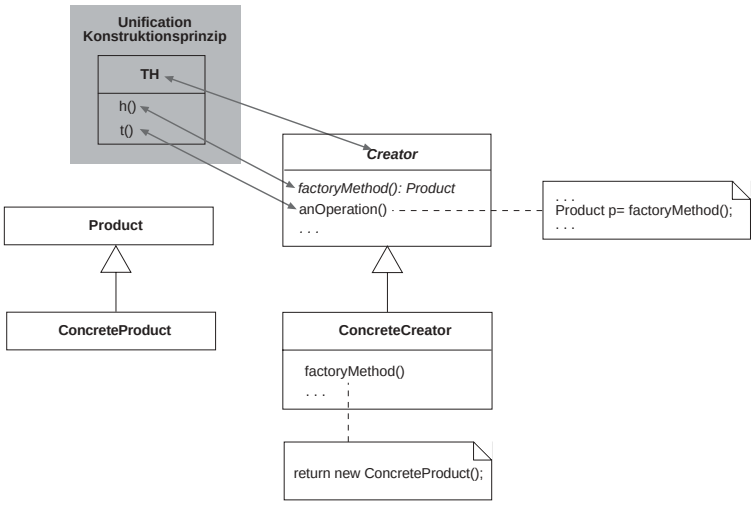

Abbildung 5.46. Template- und Hook-Methode im Entwurfsmuster Factory Method

Wenn man die Charakteristika des Unification-Konstruktionsprinzips kennt, weiß man, welche Flexibilität damit möglich ist und wie die Anpassung erfolgt, unabhängig davon, welche Funktionalität die Template- und Hook-Methoden bereitstellen. Im Fall des Entwurfsmusters Factory-Method muss die Hook-Methode createProduct() in einer Unterklasse von Creator überschrieben werden, um zu defiinieren, welche spezifische Product-Klasse instantiiert wird.

Der Name und die Funktionalität der Hook-Methode drücken aus, welcher Aspekt bei einem Entwurfsmuster flexibel gehalten werden. Im Beispiel Factory-Method geht es darum, die Objektkreation flexibel zu halten. Dazu wird das Unification-Konstruktionsprinzip angewendet. Das ist eine gute Lösung für Sprachen wie C++, die Meta-Informationen nicht ausreichend oder nicht einheitlich unterstützen. Der Nachteil des Unification-Konstruktionsprinzips ist, dass eine Unterklasse von Creator definiert werden muss, womöglich nur, um die Hook-Methode createProduct() zu überschreiben.

Dasselbe Ziel, nämlich die Objektkreation in der Klasse Creator flexibel zu halten, kann bei Sprachen wie Java und C# eleganter dadurch erreicht werden, dass der Name der Klasse, aus der ein Objekt zu erzeugen ist, als Parameter an die Methode anOperation() übergeben wird (siehe Abbildung 5.47). In der Methode anOperation() werden die entsprechende Instanz der Klasse Class und das zugehörige Objekt instantiiert. Was den Objekttyp betrifft, ist zu prüfen, ob er dem Typ Product entspricht. Damit wird derselbe Effekt erzielt, ohne das Bilden einer Unterklasse erforderlich zu machen. Der Einsatz von Meta-Informationen anstatt des Unification-Konstruktionsprinzips ist lediglich für diesen Fall möglich, da die Funktionalität – Objekterzeugung zur Laufzeit – über Meta-Informationen bereitgestellt wird.

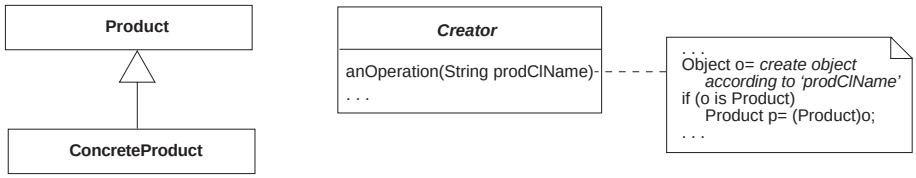

Abbildung 5.47. Dynamische Objekterzeugung als Alternative zum Entwurfsmuster Factory Method

Eine weitere Alternative, bei der keine Unterklassenbildung nötig ist, greift auf das Unification-Konstruktionsprinzip zurück. Anstatt die Hook-Methode als abstrakte Methode in der Klasse Creator zu definieren, wird factoryMethod() implementiert. Somit ist kein Überschreiben der Hook-Methode in einer Unterklasse nötig, wenn das damit erzeugte Objekt passt.

Das Entwurfsmuster Abstract Factory dient ebenfalls dazu, die Objektkreation flexibel zu halten. Es bietet im Vergleich zum Entwurfsmuster Factory Method mehr Flexibilität, da es auf dem Separation-Konstruktionsprinzip basiert. Interessant ist, dass die beiden Entwurfsmuster Factory Method und Abstract Factory exemplarisch zeigen, wie ein Entwurf, der auf dem Unification-Konstruktionsprinzip beruht, in einen Entwurf transformiert werden kann, dem das Separation-Konstruktionsprinzip zugrunde liegt. Dazu ist lediglich die Hook-Methode in eine separate Klasse oder Schnittstelle zu verschieben. Im erwähnten Beispiel ist die Hook-Methode factoryMethod() in eine separate Klasse zu verschieben, die mit der Creator-Klasse abstrakt gekoppelt ist (siehe Abbildung 5.48). Im Buch von Gamma et al. (1995) ist die Transformation durch Umbenennungen nicht offensichtlich. Es wurde im Entwurfsmuster Abstract Factory die Hook-Methode von factoryMethod() in createProduct() umbenannt. Die Klasse Creator ist in Client umbenannt worden.

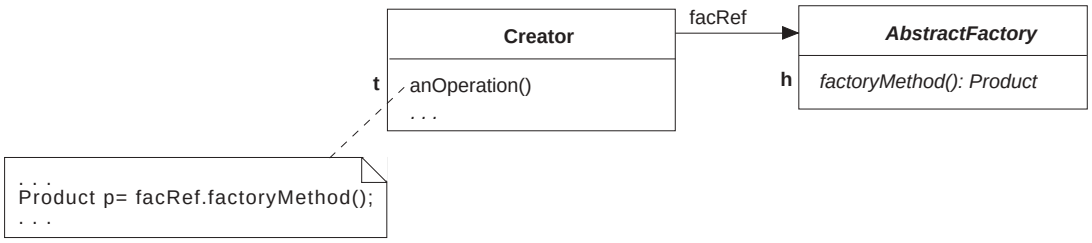

Abbildung 5.48. Transformation des Entwurfsmusters Factory Method zum Entwurfsmuster Abstract Factory

Namensgebung der Entwurfsmuster

Die Namen der Entwurfsmuster für Produktfamilien bei Gamma et al. (1995) leiten sich aus der Semantik der Hook-Methode ab. Mit anderen Worten formuliert, bestimmt jener Aspekt eines Entwurfsmusters, der flexibel gehalten wird, welchen Namen das Entwurfsmuster bekommt.

Analysiert man die Entwurfsmuster dahingehend, fällt auf, dass jeweils entweder der Name der Hook-Methode oder der Name der Klasse, die die Hook-Methode enthält, dem Namen des Entwurfsmusters entspricht. Beispielsweise heißt im Entwurfsmuster Factory Method die Hook-Methode factoryMethod(). Im Entwurfsmuster Abstract Factory heißt die Klasse, die die Hook-Methode enthält, Abstract Factory. Analoges gilt für die Entwurfsmuster State, Strategy, Builder, Observer, Command, Prototype und Interpreter.

Die Namensgebung von selbst definierten Entwurfsmustern könnte sich nach dieser Regel richten: die Hook-Semantik bestimmt den Namen des Entwurfsmusters. Somit lassen sich beliebig viele Entwurfsmuster für objektorientierte Produktfamilien systematisch benennen. Das erlaubt eine weitere Perspektive auf den Katalog von Entwurfsmustern, wie ihn Gamma et al. (1995) definiert haben: Im Katalog sind weitgehend jene Entwurfsmuster aufgenommen, die in verschiedenen Anwendungsbereichen einsetzbar sind. Die beiden Entwurfsmuster Factory Method und Abstract Factory sind gute Beispiele dafür, da die Flexibilisierung der Objekterzeugung in vielen Anwendungsbereichen einsetzbar ist. Das gilt aber nicht für alle Produktfamilien-Entwurfsmuster im Katalog. Beispielsweise ist das Entwurfsmuster Interpreter eng mit den Anwendungsbereichen verbunden, die auf formalen Sprachen beruhen.