5 Konstruktion anpassbarer 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 verständlich, 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.

Obwohl es ausgereifte Techniken gibt, diverse Aspekte eines Softwaresystems anpassbar zu machen, bleibt folgendes Problem bestehen: Die Anpassbarkeit wird nur dort gut erreicht, wo man Anpassungen explizit vorsieht. Je mehr Anpassbarkeit a priori vorgesehen wird, umso komplexer und unverständlicher wird jedoch das Softwaresystem. Diese Komplexität wiederum beeinträchtigt generell die Änderbarbarkeit und Erweiterbarkeit, insbesondere, wenn nicht vorhergesehene Anforderungen zu berücksichtigen sind. Wie die Erfahrung zeigt, sind viele Anforderungen zum Zeitpunkt der Entwicklung eines Softwaresystems nicht vorhersehbar.

Dieses Kapitel soll ein Verständnis dafür schaffen, welche Komplexität durch Flexibilisierung entsteht. Die adäquate Balance zwischen Änderbarkeit und Verständlichkeit zu schaffen, ist eine der Herausforderungen beim Systementwurf.

5.1 Konfigurationsparameter als Basis für anpassbare Software

Sogenannte Konfigurationsparameter sind in *Parametrierungsdateien* (= *Konfigurationsdateien*) abgelegt. Die *Konfigurationsparameter* entsprechen persistenten, globalen (= statischen) Variablen. Im Folgenden betrachten wir, wie Konfigurationsparameter genutzt werden, um Softwaresysteme anzupassen, ohne deren Quelltext ändern zu müssen.

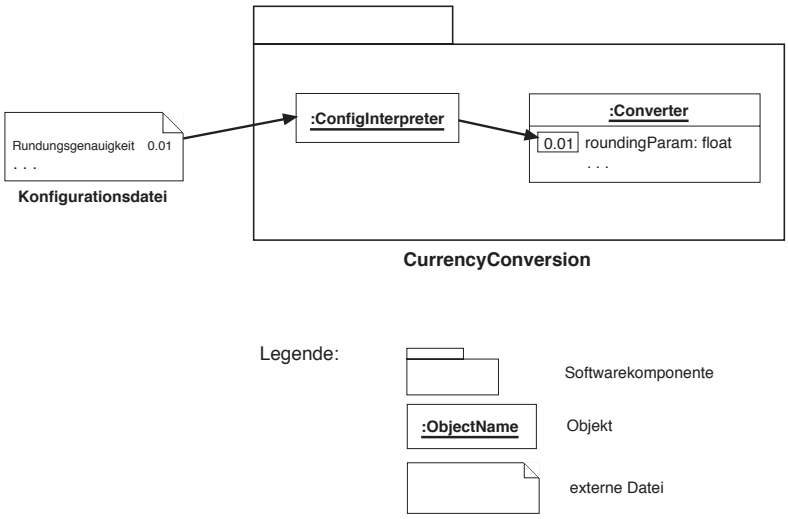

Abbildung 5.1. Anpassung über eine Konfigurationsdatei[1]

5.1.1 Paramtereinstellungen über globale, statische Variable

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 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 eines Converter-Objektes ändern zu können, ohne den Quelltext 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. Die Werte in der Konfigurationsdatei entsprechen globalen, statischen Variablen.

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 Quelltextes zu vermeiden, wenn die Benutzungsschnittstelle eines Softwareproduktes

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

beispielsweise in verschiedenen Sprachen verfügbar gemacht werden soll, eignen sich Konfigurationsdateien, um die zur Festlegung der gewünschten Ausprägung eines GUI erforderlichen Parametrierungswerte bereitzustellen. Als spezielle Bezeichnung für GUI-Konfigurationsdateien wird häufig der Begriff *Ressource-Datei* verwendet. 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.

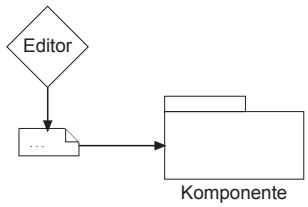

Abbildung 5.2. Anpassung unter Zuhilfenahme eines Editors

5.1.2 Callback-Style of Programming – Funktionen und Prozeduren als Parameter

In manchen Programmiersprachen, wie z.B C, Pascal, Oberon, und C#, können nicht nur Datenobjekte sondern auch Funktionen oder Prozeduren als Parameter verwendet werden. 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. Diese Technik wird *„Callback-Style of Programming“* genannt. Wenn das Laufzeitsystem das dynamische Laden und Linken von Programmteilen unterstützt, können Prozeduren und Funktionen sogar 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 der Programmiersprache C ein Funktionsparameter wie folgt deklariert:

```
void DoSomething( int (*Compare)(void*, void*), void*elem1, void*elem2 )
```

Das bedeutet, dass die Funktion DoSomething() als ersten formalen Parameter einen 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 Problem, dass dazu die Typprüfung durch einen sogenannten *Type-Cast* umgangen werden muss. In C wird das durch Angabe des Typs in runden Klammern vor dem Variablennamen ausgedrückt, wie zum Beispiel (char*)string1. Die Funktion StringCompare soll zwei Zeichenketten vom Typ char* vergleichen. Da Compare als formalen 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(    // C-Bibliotheksfunktion strcmp
                    (char*)string1,
                    (char*)string2
          );
```

```
} // StringCompare
```

Ein Aufruf von DoSomething() könnte beispielsweise die folgenden aktuellen Parameter mitgeben:

```
DoSomething(StringCompare, “first“, “second“);
```

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

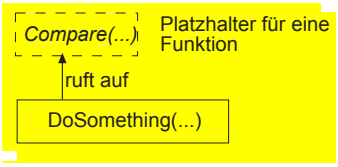

Abbildung 5.3. Beispiel für Callback-Style of Programming

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.

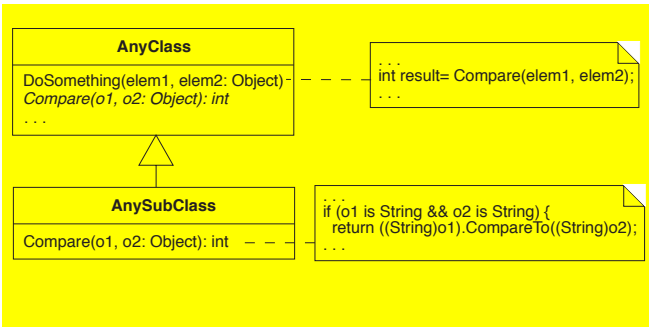

Abbildung 5.4. Callback-Style of Programming durch Vererbung und dynamische Bindung

Den interessierten Leser verweisen wir auf die objektorientierte Programmiersprache Oberon (Reiser und Wirth, 1992). Oberon ist eine Weiterentwicklung von Modula-2, die durch einen erweiterbaren Verbund (Record) und Funktionsparameter zu einer objektorientierten Sprache wird. Das Studium von Oberon macht ersichtlich, wie eine einzige konzeptionelle Ergänzung einer konventionellen Programmiersprache – der erweiterbare Verbund – ausreicht, um objektorientiert programmieren zu können.

5.2 Konzepte und Konstruktionsprinzipien für anpassbare, objektorientierte Produktfamilien

In der industriellen Fertigung sind Produktfamilien weit verbreitet. Beispielsweise sind die Modelle eines Automobilherstellers im Kern weitgehend identisch, sie unterscheiden sich zum Beispiel 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 Produktfamilien ebenfalls skizzieren (siehe Abbildung 5.5): Durch Aufstecken einer bestimmten Komponente auf die Komponente „elektischer Antrieb für Küchenmaschine“ wird daraus beispielsweise ein Fruchtmixer oder ein Fleischwolf. Die Komponente „elektischer Antrieb für Küchenmaschine“ besteht unter anderem aus einem Elektromotor, einem Schalter, einem Gehäuse und einer Antriebswelle, die die Schnittstelle zu den aufsteckbaren Komponenten bildet. Die Komponente „elektischer Antrieb für Küchenmaschine“ an sich ist für einen bestimmten Benutzer, z.B. einen Koch, nutzlos. Durch Aufstecken einer passenden Komponente wird sie 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 Produktfamilien zu flexibilisieren und zu rationalisieren. Die von objektorientierten Sprachen angebotenen Sprachkonstrukte eignen sich gut, um Produktfamilien für Softwaresysteme zu realisieren.

In der Softwaretechnik sind objektorientierte Konstruktionsprinzipien zur Herstellung von Produktfamilien erstmals bei der Entwicklung von Software zur Simulation diskreter Ereignisse eingesetzt worden; dafür wurde von Dahl und Nygard (1966) eine eigene Programmiersprache, Simula genannt, entwickelt.

Zu einer weit verbreiteten Technik wurde der Einsatz von Produktfamilien, als man begann, Softwareprodukte mit grafischen Benutzungsschnittstellen zu entwickeln. Die Sprache Smalltalk zusammen mit einer umfangreichen Klassenbibliothek trugen ebenfalls zur Verbreitung dieser Technik bei.

Die Produkte der Firma SAP stellen eine Produktfamilie 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 Produktfamilien auf.

In den vergangenen Jahren sind diverse Synonyme für den Begriff Produktfamilie eingeführt worden, zum Beispiel Rahmenwerk als Übersetzung des englischen Begriffes Framework oder Produktlinie. Da der Begriff Produktfamilie gut ausdrückt, dass aus einem Kern verschiedene Ausprägungen eines Softwareproduktes gebildet werden können, verwenden wir diese Begriffsbezeichnung.

Wir definieren eine *Produktfamilie* als

> *ein Stück Software, aus dem durch den Callback-Style-of-Programming verschiedene Ausprägungen gebildet werden können, d.h. dessen Verhalten dadurch veränderbar und/oder 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 aber aus. Wir behandeln hier ausschließlich Konstruktionsprinzipien für objektorientierte Produktfamilien.

In den folgenden beiden Abschnitten 5.2.1 und 5.2.2 wird skizziert, wie objektorientierte Produktfamilien (in objektorientierten Sprachen) auf Basis der sogenannten *abstrakten Kopplung* und der *Template-* und *Hook-Methoden* konzipiert werden. Danach werden essentielle Konstruktionsprinzipien für die Entwicklung objektorientierter Produktfamilien erläutert (Abschnitte 5.2.3 bis 5.2.6).

Die unterschiedlichen Konstruktionsprinzipien erlauben verschiedene Freiheitsgrade bei der Anpassung einer Produktfamilie für einen bestimmten Zweck. Die Anwendung der beschriebenen Konstruktionsprinzipien fand auch ihren Niederschlag bei der Entwicklung von Entwurfsmustern durch Gamma et al. (1995). Eine Darstellung der Zusammenhänge zwischen den Konstruktionsprinzipien und den Entwurfsmustern von Gamma et al. in Abschnitt 5.3 soll das Verständnis von Entwurfsmustern erleichtern.

5.2.1 Das Konzept der abstrakten Kopplung

Das Potenzial objektorientierter Sprachen in Bezug auf Wiederverwendbarkeit wird oft nicht ausgenutzt, insbesondere wenn sie nur dazu benutzt werden, eine modulorientierte Architektur eines Softwaresystems zu implementieren, bei der spezifisch auf eine Anwendung zugeschnittene Komponenten miteinander gekoppelt werden. Abbildung 5.6 zeigt schematisch (nicht in UML-Notation), wie die Instanzen von Klassen (= Komponenten) in solcherart entworfenen Systemen miteinander gekoppelt sind.

Als Beispiel betrachten wir ein Softwaresystem zur Steuerung eines autonom, also ohne Piloten fliegenden Helikopters. Wir nehmen an, dass sich das Softwaresystem in ein Steuerungssubsystem, das das Flugverhalten kontrolliert, und ein Navigationssubsystem gliedert. Abbildung 5.6 zeigt eine grobgranulare Modularisierung des Navigationssubsystems, die in der höchsten Abstraktionsstufe ein Navigationskomponente und eine Global Positioning System (GPS-)Komponente definiert. Navigationskomponente und GPS-Komponente sind miteinander gekoppelt, indem Quelltextteile der Navigationskomponente Funktionalität der GPS-Komponente benutzen.

Die GPS-Komponente liefert aufgrund des empfangenen GPS-Signals die aktuelle Position des Helikopters und die dazugehörenden Zeitangaben. Die Navigationskomponente 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 in der Navigationskomponente in Abbildung 5.6 stellen schematisch den Quelltextteil dar, der die Navigationskomponente mit der GPS-Komponente koppelt.

GPS-Komponente

Navigationskomponente

Abbildung 5.6. Kopplung der Navigationskomponente mit der GPS-Komponente.

Angenommen es wäre zusätzlich zum amerikanischen GPS bereits das europäische Galileo-System verfügbar, und man will die GPS-Komponente durch eine Galileo-

Komponente ersetzen, dann erfordert dies eine entsprechende Änderung des Quelltextes in der Navigationskomponente (siehe Abbildung 5.7).

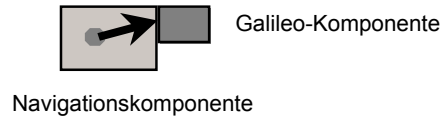

Abbildung 5.7. Kopplung der Navigationskomponente mit der Galileo-Komponente.

Abstrakte Entitäten als Basis für Produktfamilien

Die Kopplung mit einer bestimmten Komponente manifestiert sich hier in der Implementierung der Navigationskomponente. Sie hängt also davon ab, welche Komponente (GPS oder Galileo) verwendet wird. Das ist unerwünscht und es erhebt sich die Frage, wie eine derartige Abhängigkeit verhindert und wie das Navigationssubsystem zu einer Produktfamilie werden kann, in der das verwendete Positionierungssystem austauschbar ist, ohne dass die Navigationskomponente 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. Den Begriff abstrakte Entität verwenden wir somit als Oberbegriff für abstrakte Klasse und Schnittstelle. Abbildung 5.8 zeigt schematisch die Konzeption des Navigationssubsystems als Produktfamilie, so dass die Navigationskomponente mit jedem Positionierungssystem gekoppelt werden kann, das mit der abstrakten Klasse oder Schnittstelle PosSystem kompatibel ist. Wir sprechen dann von *abstrakter Kopplung*, wenn eine Klasse mit einer abstrakten Entität assoziiert ist.

Der Polymorphismus der Objektorientierung wird ausgenutzt, um „steckerkompatible“ Klassen zu definieren. Die dynamische Bindung ermöglicht es, Quelltext der Navigationskomponente 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 die Navigationskomponente „eingesteckten“ Objektes ab. Das Einstecken von Objekten, die zur abstrakten Entität kompatibel sind, erfordert keine Änderung dieses Quelltextes in der Navigationskomponente.

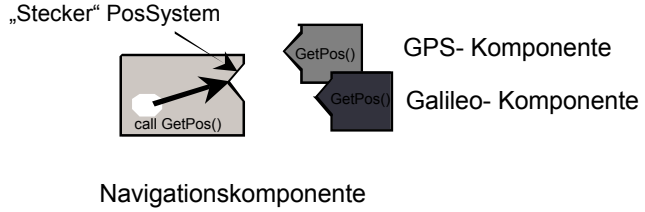

Abbildung 5.8. Abstrakte Kopplung der Navigationskomponente mit der Positionierungskomponente.

Definition einer abstrakten Entität als abstrakte Klasse

Abbildung 5.9 zeigt das der Abbildung 5.8 entsprechende UML-Diagramm. Wir nehmen an, dass die abstrakte Entität PosSystem als abstrakte Klasse definiert wird, die die abstrakten Methoden GetPos() und GetTime() zur Verfügung stellt. Jede

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 zu tun ist, nämlich die Positionskoordinaten abfragen, aber nicht wie die Positionskoordinaten geliefert werden. 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 besorgt auf andere Art die Positionskoordinaten wie eine Instanz der Klasse GPS.

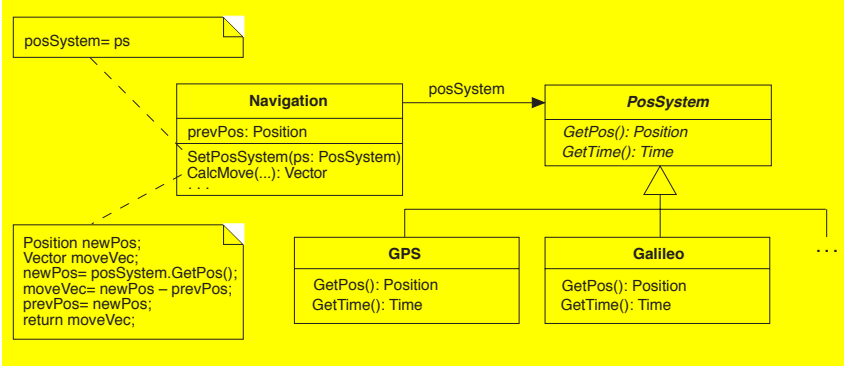

Abbildung 5.9. UML-Diagramm der objektorientierten Produktfamilie Navigationssystem

Durch die Entwurfsentscheidung, eine abstrakte Entität PosSystem einzuführen, ist eine Produktfamilie entstanden, die es gestattet, das Navigationssubsystem im Hinblick auf das zu verwendende Positionierungssystem an spezielle Gegebenheiten anzupassen, indem eine (kompatible) Positionierungskomponente eingesteckt werden kann. Die abstrakte Entität stellt quasi eine Einsteckvorrichtung dar, die mit der Einsteckvorrichtung der eingangs erwähnten Küchenmaschine vergleichbar ist.

Komponenten, die zur Produktfamilie gehören und zur Konfiguration verwendet werden, müssen mit der abstrakten Entität (das heißt mit der Einsteckvorrichtung) kompatibel sein. Die Einsteckvorrichtung definiert einen Standard und dieser muss so gestaltet sein, dass er für die spezifischen Komponenten passend ist. Wäre beispielsweise bei der Küchenmaschine die Kupplung der Antriebswelle zu schwach dimensioniert, würde eine Konfiguration als Fleischwolf vielleicht nicht funktionieren. Hätte man in der abstrakten Klasse PosSystem die Methode GetTime() nicht vorgesehen, wären Geschwindigkeitsberechnungen in der Navigationskomponente nicht möglich.

Zusammenfassend gilt für die Verwendung von abstrakten Klassen zur Bildung von Produktfamilien folgendes:

- Instanzvariablen und Methoden von ähnlichen spezifischen Klassen werden in einer abstrakten Klasse zusammengefasst, von der die spezifischen Klassen abgeleitet werden.
- Abstrakte Klassen enthalten sowohl Methoden, für die – im Sinne der Abstraktion – nur ihr Name und die Parameter festlegt werden, als auch Methoden, die

[2] Der statische Typ einer Variablen ist der Typ, der für die Variable in der Deklaration im Programmtext festgelegt wurde. Der dynamische Typ einer Variable ist der Typ des Objektes, auf das die Variable zur Laufzeit zeigt (siehe auch Kapitel 4). Eine Variable vom statischen Typ PosSystem kann zum Beispiel auf Instanzen jeder Unterklasse von PosSystem verweisen, da diese Klassen den Typ PosSystem erweitern.

implementiert werden können. Aufgrund der Vererbung stellen die von der abstrakten Klasse abgeleiteten Klassen alle Methoden, die in der abstrakten Klasse festgelegt wurden, zur Verfügung. Dadurch wird ein Standard festgelegt, der auch als Kontrakt, Signatur oder Protokoll bezeichnet wird. Die Instanzen von Unterklassen der abstrakten Klasse können daher alle Methodenaufrufe ausführen, die in der abstrakten Klasse festgelegt sind.

- Die Methoden, für die in der abstrakten Klassen die Schnittstellen (Namen und Parameter) definiert worden sind, sind in den spezifischen Unterklassen zu implementieren. Eventuell werden bereits implementierte Methoden überschrieben oder Methoden hinzugefügt. Idealerweise wird bei der Konzeption von Produktfamilien in Unterklassen die Signatur nicht verändert, es werden also keine Methoden hinzugefügt.
- 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 der Kontrakte, die durch die abstrakten Klassen gebildet werden, implementiert werden können und somit objektorientierte Produktfamilien entstehen.

Analoges gilt, wenn man abstrakte Entitäten mittels des Schnittstellen-Konstrukts definiert. Schnittstellen können jedoch keine Methodenimplementierungen enthalten.

Definition einer abstrakten Entität als Schnittstelle (Interface)

In statisch typisierten Sprachen wie C++ ist die Typhierarchie mit der Klassenhierarchie identisch: Nur Instanzen von Unterklassen einer Klasse A sind, den Polymorhismusregeln entsprechend, kompatibel mit dem Typ A. Schnittstellen ermöglichen das Aufbrechen dieser rigiden Beziehung, wie nachfolgend erläutert wird.

Syntaktisch ist eine *Schnittstelle*, wie sie von Java und C# angeboten wird, einer abstrakten Klasse mit ausschließlich abstrakten Methoden ähnlich. Statt der Schlüsselwörter ‘abstract class‘ wird das Schlüsselwort ‘interface‘ verwendet. Es dürfen keine Methodenimplementierungen und keine Instanzvariablen vorgesehen werden.

Eine Schnittstelle ist vergleichbar mit einem „Typ-Stempel“, das heißt jede Klasse, unabhängig davon wo sie sich in der Klassenhierarchie befindet, kann mit dem Typ der Schnittstelle „gestempelt“ werden, also auch den Typ der Schnittstelle annehmen, indem die Klasse die Schnittstelle implementiert. Das Implementieren einer Schnittstelle durch eine Klasse bedeutet, dass sämtliche in der Schnittstelle vorgesehenen Methoden in der Klasse zu implementieren sind. Eine Klasse kann beliebig viele Schnittstellen implementieren, also mit beliebig vielen Typen gestempelt werden. Außerdem kann eine Schnittstelle von beliebig vielen anderen Schnittstellen erben. Damit wird eine eingeschränkte Form von mehrfacher Vererbung möglich, mit dem Vorteil, dass viele Probleme, die bei uneingeschränkter mehrfacher Vererbung auftreten können, nicht auftreten, weil bei Schnittstellen keine Methodenimplementierungen erlaubt sind.

Abbildung 5.10 zeigt, wie die Klassen A und SubR1 die Schnittstelle I implementieren. Dadurch haben Objekte dieser Klassen auch den Typ I. 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. Im Kommentarblatt wird in C#-Syntax gezeigt, wie die Klasse A die Schnittstelle I implementiert. Das Kommentarblatt ist am „Eselsohr“ in der oberen, rechten Ecke erkennbar. In der UML-Darstellung würde eine zusätzliche Implementierungsbeziehung 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 implementierten und somit mit dem Typ der Schnittstelle kompatibel sind. Das folgende Code-Beispiel zeigt diesen Sachverhalt anhand der Klassen A und SubR1, die die Schnittstelle I implementieren:

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

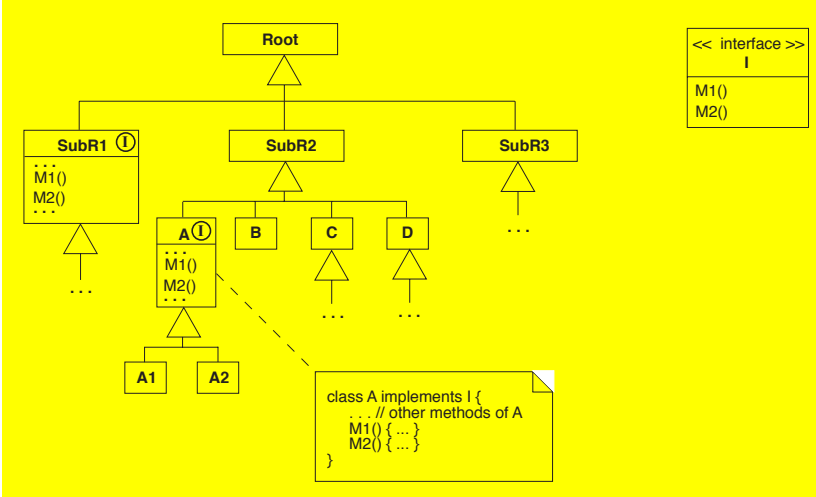

Abbildung 5.10. Das Schnittstellen-Konstrukt als Typ-Stempel

Abbildung 5.11 zeigt das UML-Diagramm des Navigationssubsystems, wenn eine Schnittstelle PosSystem anstatt einer abstrakten Klasse verwendet wird. Ein Vorteil der Verwendung des Schnittstellenkonstrukts als Alternative zur abstrakten Klasse kann sein, dass die Klassen GPS und Galileo nicht Unterklassen von einer abstrakten Klasse PosSystem sein müssen, sondern auch von einer anderen Klasse abgeleitet sein können.

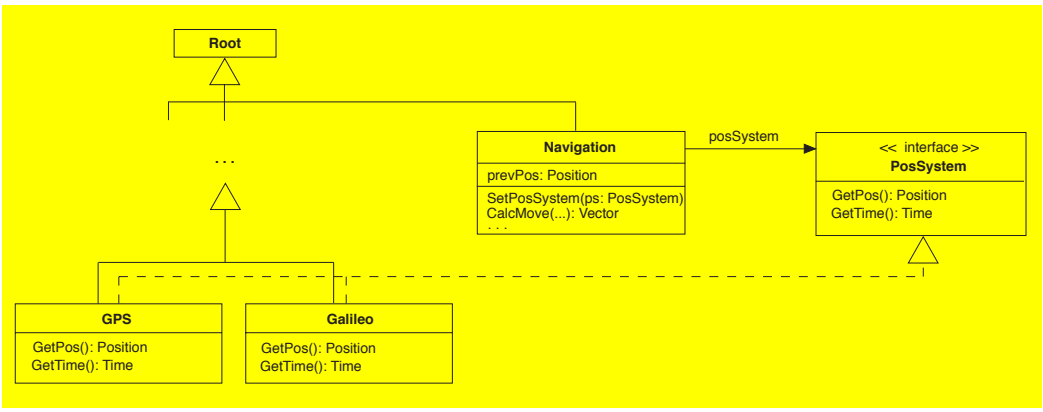

Abbildung 5.11. Die abstrakte Entität PosSystem als Schnittstelle

In der Praxis gestaltet es sich oft als schwierig, feststellen zu können, welche Eigenschaften (Methoden, Instanzvariablen) einer abstrakten Entität zugeordnet werden sollen. Es ist darauf zu achten, dass genügend gemeinsame

Instanzvariablen und Methoden, die die Funktionalität festlegen, gefunden werden. Die Modellierung von abstrakten Entitäten erfordert daher ein umfassendes Domänenwissen. Wenn zu spezifische Instanzvariablen und Methoden in eine abstrakte Entität gepackt werden, besteht die Gefahr, die Erweiterbarkeit der Produktfamilie einzuschränken.

Es muss verstanden werden, was charakteristisch für die spezifischen Entitäten ist. Meist gelingt ein guter Entwurf nicht auf Anhieb. Deshalb müssen abstrakte Entitäten im Zuge der Entwicklung von Produktfamilien immer wieder verbessert, das heißt einem Redesign unterzogen werden, bis sie einen ausreichend hohen Reifegrad aufweisen. Das ist ein Prozess, der sich durchaus über Jahre erstrecken kann. Auch wenn eine abstrakte Entität nicht von Anfang an allen Anforderungen entspricht, ist es wichtig, sie vorzusehen, damit die Qualitätsmerkmale Erweiterbarkeit und Flexibilität für die Produktfamilie entsprechend ausgeprägt sind.

5.2.2 Das Konzept der Template- und Hook-Methoden

Im Abschnitt 5.2.1 wurde exemplarisch gezeigt, wie durch abstrakte Kopplung objektorientierte Produktfamilien realisiert werden können. Vom softwaretechnischen Standpunkt aus betrachtet ist es interessant, welche Konstruktionsprinzipien für objektorientierte Produktfamilien existieren.

Um Konstruktionsprinzipien für objektorientierte Produktfamilien, wie sie in den nachfolgenden Abschnitten beschrieben sind, aufzustellen, ist es hilfreich, das erstmals in der Smalltalk-Literatur erwähnte Konzept der sogenannten Template- und Hook-Methoden zu erläutern:

Wenn in einer Methodenimplementierung eine andere Methode aufgerufen wird, nennen wir die aufrufende Methode die Template-Methode und die aufgerufene Methode die Hook-Methode. Man beachte, dass die hier angesprochene Template-Methode nichts mit dem C++ Sprachkonstrukt template zu tun hat. Abbildung 5.12 zeigt ein Beispiel, in dem eine Methode M1() eine Methode M2() aufruft, wobei beide Methoden zur gleichen Klasse gehören. Diese Aufrufbeziehung, die sich in der Implementierung der Methode M1() manifestiert, ist in einem Kommentarblatt im UML-Diagramm dargestellt. Die Methode M1() ist aufgrund der Aufrufbeziehung die Template-Methode, die Methode M2() die Hook-Methode. Die Buchstaben t und h links neben der Klasse in Abbildung 5.12 markieren die beiden Methoden[3] entsprechend. Es hängt lediglich von der Aufrufbeziehung ab, in der die beiden Methoden stehen, zu welcher Kategorie eine Methode gehört. Weder Größe noch Komplexität einer Methode haben einen Einfluss auf deren Kategorisierung.

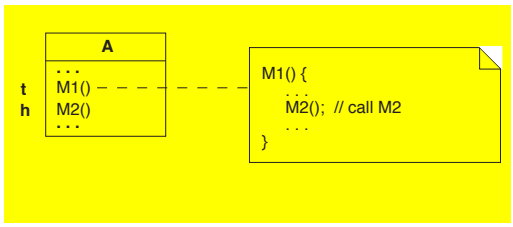

Abbildung 5.12. Template-Methode und Hook-Methode in einer Klasse

Abbildung 5.13 zeigt ein Beispiel, in dem die Template- und Hook-Methoden in verschiedenen Klassen sind. Dadurch ändert sich nur die Formulierung des Aufrufs, jedoch nichts an der Aufrufbeziehung. Wir nehmen an, dass die im UML-Diagramm

[3] In UML ist eine Markierung von Template- und Hook-Methoden nicht vorgesehen. Diese Erweiterung wurde im UML-F-Profil definiert (siehe Fontoura, Pree, Rumpe, 2002).

mit bRef bezeichnete Assoziation zwischen den Klassen A und B durch eine entsprechende Instanzvariable in der Klasse A implementiert wird.

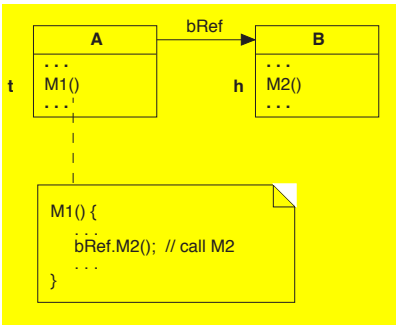

Abbildung 5.13. Template-Methode und Hook-Methode in verschiedenen Klassen

Welche Methode eine Template- und welche eine Hook-Methode ist, hängt vom Kontext ab. Wenn beispielsweise die Methode M2() die Methode M3() aufruft und wir nur die 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 Betrachtung von M2() und M3(). Dies ist von Bedeutung, wenn wir die Konstruktionsprinzipien von objektorientierten Produktfamilien betrachten. Da die Konstruktionsprinzipien nur auf der Unterscheidung von Template-Methode und Hook-Methode beruhen, sind die Konstruktionsprinzipien unabhängig von der Komplexität der Methoden-implementierung anwendbar. Die betrachteten Methoden können wenige Zeilen Code haben und zu einer sehr einfachen Klasse gehören, oder komplexe Methoden von Klassen mit großem Funktionsumfang sein. Die Konstruktionsprinzipien sind außerdem unabhängig vom spezifischen Anwendungsbereich.

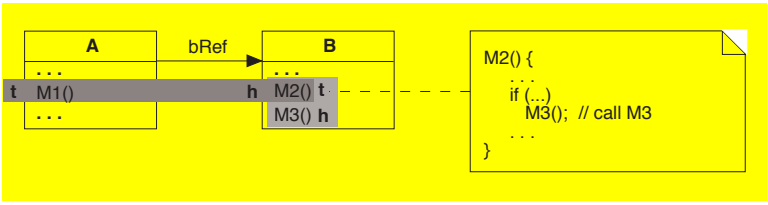

Abbildung 5.14. Eine Hook-Methode wird zu einer Template-Methode in einem anderen Kontext

Die Bezeichnungen Template und Hook wurden gewählt, weil sie in der Smalltalk-Literatur, die zur Verbreitung des objektorientierten Denkmodells einen wesentlichen Beitrag geleistet hat, geprägt wurden. Da eine Template-Methode die entsprechende Hook-Methode aufruft, kann das Verhalten der Template-Methode durch Überschreiben der Hook-Methode verändert werden, ohne dass man die Template-Methode ersetzen muss. Die Hook-Methode „hängt“ gleichsam wie ein Parameter in der Template-Methode.

In der Beschreibung der Konstruktionsprinzipien für objektorientierte Produktfamilien bezeichnen wir die Template-Methode mit T() und die Hook-Methode mit H(). Die Klasse, die die Template-Methode enthält bezeichnen wir mit T. Die Klasse, die die Hook-Methode enthält bezeichnen wir mit H. Wir betrachten alle sinnvollen Kombinationen von Template- und Hook-Methoden in einer beziehungsweise in zwei Klassen.

Im einfachsten Fall sind beide Methoden Teil ein und derselben Klasse (siehe Abbildung 5.12 und Abbildung 5.16). Diesen Fall bezeichnen wir als *Hook-Method-*

Konstruktionsprinzip. Wenn die beiden Methoden in verschiedenen Klassen sind, sprechen wir vom *Hook-Object-Konstruktionsprinzip*. Das Hook-Method-Konstruktionsprinzip wird in Abschnitt 5.2.3, das Hook-Object-Konstruktionsprinzip in Abschnitt 5.2.4 näher erläutert.

Zieht man auch die Vererbung in Betracht, ergeben sich die drei in Abbildung 5.15 dargestellten Konstruktionsprinzipien: Composite, Decorator und Chain-Of-Responsibility. Auf den Umstand, dass (wie aus Abbildung 5.15 ersichtlich) die Namen von Template- und Hook-Methoden gleich sind, wird weiter unten eingegangen. Die Bezeichnung der Konstruktionsprinzipien – Composite, Decorator, Chain-Of-Responsibility – wurde aus dem Katalog von Entwurfsmustern von Gamma et al. (1995) übernommen. Das Composite-Konstruktionsprinzip entspricht dem Composite-Entwurfsmuster. Analoges gilt für Decorator und Chain-Of-Responsibility.

Mit dem *Composite-Konstruktionsprinzip* kann ein Baum von Objekten definiert werden, der sich für den Benutzer wie ein einziges Objekt verhält. Mit Benutzer ist der Programmierer gemeint, der ein Objekt verwendet.

Mit dem *Decorator-Konstruktionsprinzip* kann das Verhalten eines Objektes durch Komposition mit einem Decorator-Objekt angepasst werden. Auch hier verwendet der Benutzer das „dekorierte“ Objekt wie ein einziges Objekt, das, wenn nötig, durch Komposition mit weiteren Decorator-Objekten weiter angepasst werden kann.

Analog zu den beiden Konstruktionsprinzipien Composite und Decorator kann mit dem Chain-Of-Responsibility-Konstruktionsprinzip die Erledigung einer Aufgabe auf mehrere Objekte aufgeteilt werden, ohne dass der Benutzer der „Kette“ (= Liste) von Objekten das zu berücksichtigen hat.

Composite- und Decorator-Konstruktionsprinzipien unterscheiden sich in der Kardinalität der Beziehung zwischen der T- und der H-Klasse. Beim Composite-Konstruktionsprinzip darf ein T-Objekt eine beliebige Anzahl von H-Objekten referenzieren. Beim Decorator-Konstruktionsprinzip darf ein T-Objekt nur ein H-Objekt referenzieren. Trotz dieses geringen Unterschiedes sind, wie in den Abschnitten 5.2.5 und 5.2.6 erläutert wird, die Eigenschaften und Einsatzbereiche der beiden Konstruktionsprinzipien unterschiedlich.

Von den drei Konstruktionsprinzipien ist nach Einschätzung der Autoren das Composite das am häufigsten anwendbare Konstruktionsprinzip. Ein Grund dafür ist, dass es oft zweckmäßig ist, Bäume zur Verwaltung von Objekten zu verwenden. Weiters gibt es, im Gegensatz zum Decorator-Konstruktionsprinzip, keine Anforderung an die H-Klasse: Beim Decorator-Konstruktionsprinzip sollen die Methoden in der H-Klasse eine Obermenge der Methoden in sämtlichen Unterklassen sein. Mit anderen Worten ausgedrückt 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* bezieht sich auf den (degenerierten) Fall, in dem die T- und H-Klasse zu einer Klasse vereint sind, aber dennoch die Referenzbeziehung erhalten ist. Das Chain-Of-Responsibility-Konstruktionsprinzip wird als unbedeutend eingeschätzt und wir verweisen auf die Beschreibung dieses Konstruktionsprinzips in Gamma et al. (1995).

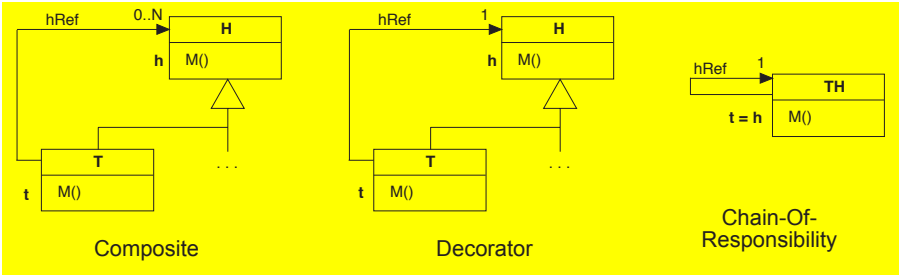

Abbildung 5.15. Rekursive Konstruktionsprinzipien

Damit haben wir die wichtigsten Konstruktionsprinzipien für objektorientierte Produktfamilien vorgestellt. Nicht betrachtet wurden Kombinationen, in denen die Klasse mit der Hook-Methode von der Klasse mit der Template-Methode erbt. Diese Kombinationen machen keinen Sinn, da sie dem Prinzip der Vererbung, dass Unterklassen Oberklassen spezialisieren beziehungsweise verfeinern, widersprechen. Da Hook-Methoden Parameter der Template-Methoden repräsentieren, ergibt es keinen Sinn, Kombinationen zu betrachten, in denen die Klasse mit der Hook-Methode (die allgemeinere Klasse) von der Klasse mit der Template-Methode (die speziellere Klasse) erbt.

Was macht den rekursiven Charakter der drei Konstruktionsprinzipien aus? Weil die T-Klasse von der H-Klasse erbt, ist der Typ T mit dem Typ H kompatibel. Die Beziehung „ein T-Objekt referenziert ein Objekt vom statischen Typ H“ schließt daher auch die Beziehung „ein T-Objekt referenziert ein T-Objekt“ mit ein, da ein T-Objekt mit dem 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 im Composite- und Decorator-Konstruktionsprinzip identisch. Lediglich ihre Implementierungen unterscheiden sich. Wie das Beispiel zur Erläuterung des Composite-Konstruktionsprinzips in Abschnitt 5.2.5 zeigt, erhält dadurch die Template-Methode einen rekursiven Charakter, obwohl sie keine rekursive Methode ist.

Das Composite-Konstruktionsprinzip wird in Abschnitt 5.2.5, das Decorator-Konstruktionsprinzip in Abschnitt 5.2.6 näher erläutert.

5.2.3 Das Hook-Method-Konstruktionsprinzip

Das Hook-Method-Konstruktionsprinzip ermöglicht die Anpassung von Software durch Vererbung und Überschreibung von Methoden. Wie nachfolgend erläutert wird, ist damit der im Vergleich zu den anderen Konstruktionsprinzipien geringste Grad an Flexibilität für eine Anpassung gegeben.

Abbildung 5.16 zeigt die wesentlichen Aspekte des Hook-Method-Konstruktionsprinzips. Da Template-Methode T() und Hook-Methode H() in einer Klasse TH sind, kann das Verhalten der Template-Methode durch Überschreiben der Hook-Methode in einer Unterklasse von TH angepasst werden. Wenn kein typisches Verhalten einer Hook-Methode angegeben werden kann, sollte sie als eine abstrakte Methode definiert werden, die unbedingt überschrieben werden muss.

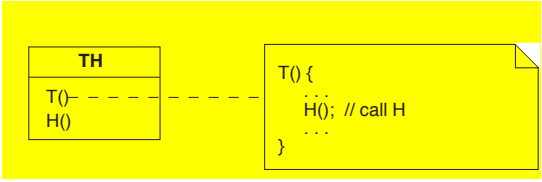

Abbildung 5.16. Die verallgemeinerte Anwendung des Hook-Method-Konstruktionsprinzips

Wenn eine Anpassung des Verhaltens erfolgen soll, erfordert dies, dass eine Instanz der Klasse SubTH statt einer Instanz der Klasse TH erzeugt wird (siehe Abbildung 5.17). Typischerweise ist bei Anwendung des Hook-Method-Konstruktionsprinzips die Anpassung mit einem Neustart der Anwendung verbunden: Es wird die Unterklasse SubTH implementiert und dann werden die Stellen im Programmcode, die eine Instanz von TH erzeugen, dahingehend geändert, dass eine Instanz von SubTH erzeugt wird.

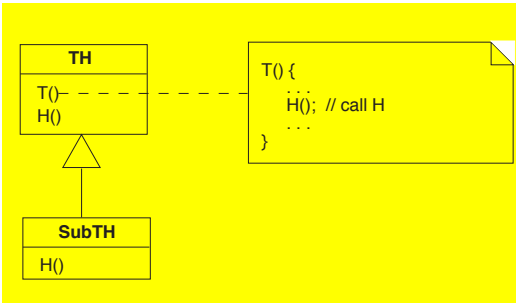

Abbildung 5.17. Anpassung der Hook-Methode

Abbildung 5.18 zeigt die Anwendung des Hook-Method-Konstruktionsprinzips bei der Modellierung des Navigationssubsystems. Daraus wird die Anpassbarkeit des Navigationssubsystems ersichtlich. Die Anwendung des Hook-Method-Konstruktionsprinzips beeinflußt die Modularisierung. Es gibt keine explizite abstrakte Entität PosSystem. Stattdessen sind die beiden abstrakten Methoden GetPos() und GetTime() als Hook-Methoden Teil der abstrakten Klasse Navigation. Die Unterklassen GPSNavigation und GalileoNavigation überschreiben die beiden Hook-Methoden so, dass eine Instanz von GPSNavigation die Positionskoordinaten und die Zeit von einem GPS-Empfänger und eine Instanz von GalileoNavigation die Positionskoordinaten und die Zeit von einem Galileo-Empfänger besorgen können. Da die Klasse Navigation abstrakt ist, muss eine Anpassung in einer Unterklasse erfolgen.

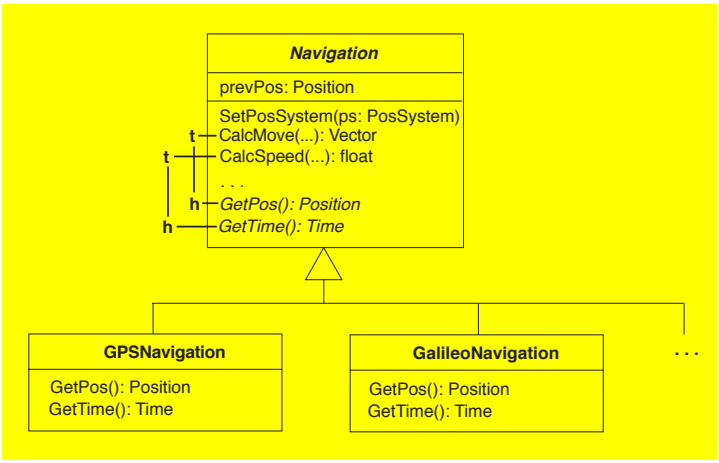

Abbildung 5.18. Anwendung des Hook-Method-Konstruktionsprinzips am Beispiel Navigation

Abbildung 5.19 zeigt eine andere Art der Anwendung des Hook-Method-Konstruktionsprinzips am Beispiel Navigationssubsystem. Statt einer abstrakten Klasse Navigation wird hier eine konkrete Klasse GPSNavigation verwendet, in der die Hook-Methoden GetPos() und GetTime() implementiert werden, um mit einem GPS-Empfänger zu arbeiten. Wenn Galileo in Betrieb gehen wird, werden in einer Unterklasse GalileoNavigation die beiden Hook-Methoden überschrieben, so dass auch mit einem Galileo-Empfänger gearbeitet werden kann. Der unschöne Nebeneffekt dabei ist, dass die Vererbungsbeziehung ausdrückt, dass Galileo eine Spezialisierung von GPS sei. In der Praxis treffen wir häufig auf solche Lösungen, die man aber nach Möglichkeit vermeiden sollte.

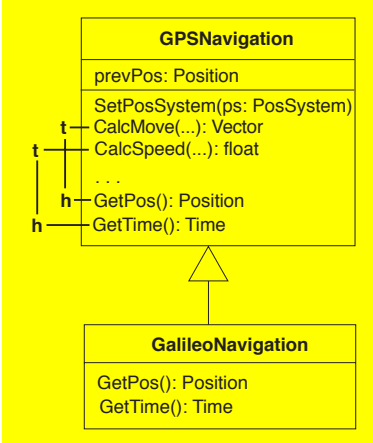

Abbildung 5.19. Konkrete statt abstrakter Hook-Methoden am Beispiel Navigation

Der Vorteil des Hook-Method-Konstruktionsprinzips ist seine Einfachheit: Es muss lediglich eine Hook-Methode für das Verhalten, das anpassbar sein soll, vorgesehen werden. Der Nachteil ist, dass die Anpassung eine Unterklassenbildung und das Überschreiben der Hook-Methode erfordert. In vielen Fällen ist aber die durch das Hook-Method-Konstruktionsprinzip gebotene Flexibilität für Anpassungen ausreichend.

5.2.4 Das Hook-Object-Konstruktionsprinzip

Das Hook-Object-Konstruktionsprinzip ermöglicht die Anpassung von Software durch Komposition von Objekten anstatt durch Vererbung. Wie nachfolgend erläutert wird, ist damit die Möglichkeit der Anpassung einer Anwendung zur Laufzeit gegeben. Die Anpassungsflexibilität steigt durch Anwendung des Hook-Object-Konstruktionsprinzips im Vergleich zur Anwendung des Hook-Method-Konstruktionsprinzips.

Abbildung 5.20 zeigt die wesentlichen Aspekte des Hook-Object-Konstruktionsprinzips. Die Template-Methode Tm() und die Hook-Methode Hm() sind dabei in zwei verschiedenen (= separaten) Klassen, die miteinander gekoppelt sind. Wir nennen die Template-Methode Tm(), da T() bedeuten würde, dass es sich um einen Konstruktor handelt. Analoges gilt für die Hook-Methode.

Die Kopplung kann auf verschiedene Arten erfolgen. Meist wird in der T-Klasse eine Instanzvariable vom statischen Typ H definiert. In Abbildung 5.20 heißt die Instanzvariable hRef. Eine andere Möglichkeit der Kopplung ist, dass die Template-Methode eine 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 Verwendung globaler Variablen ist jedoch abzulehnen und nur in wenigen, begründeten Ausnahmen überhaupt in Betracht zu ziehen, weil mit der Verwendung von globalen Variablen viele Probleme einhergehen und die Güte der Modularisierung dadurch abnimmt.

Das UML-Diagramm in Abbildung 5.20 macht keine Angabe über die Kardinalität der Beziehung zwischen T und H. Aus dem Quelltext des Kommentarblattes ist zu schließen, dass eine 1:1 Verbindung vorliegt, also ein T Objekt mit einem H Objekt gekoppelt ist. Die Flexibilitäts-Eigenschaften des Hook-Object-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.

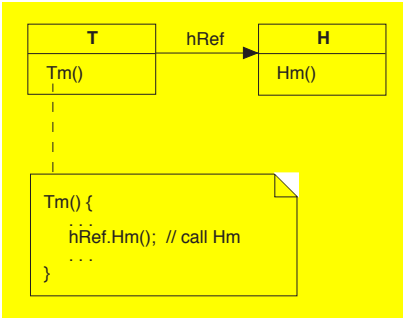

Abbildung 5.20. Die verallgemeinerte Anwendung des Hook-Object-Konstruktionsprinzips

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 Referenzvariablen des T-Objektes zugewiesen wird. Im UML-Diagramm in Abbildung 5.21 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 quasi durch „Einstecken“ eines H-Objektes in ein T-Objekt zur Laufzeit verändert werden. In der Literatur (zum Beispiel in Gamma et al., 1995) bezeichnet man diese Art der Anpassung als *(Objekt-)Komposition*. Der Grad der Flexibilität wird durch Verwendung des Hook-Object-Konstruktionsprinzips im Vergleich zur Verwendung des Hook-Method-Prinzips erhöht.

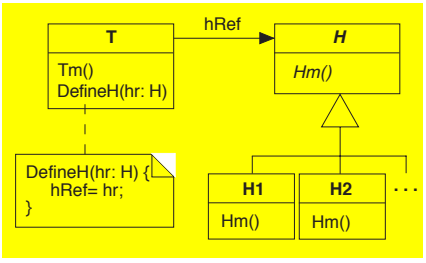

Abbildung 5.21. Die Methode DefineH() zur Anpassung durch Einstecken von H-Objekten zur Laufzeit.

In Abbildung 5.21 sind exemplarisch zwei Unterklassen H1 und H2 der abstrakten H-Klasse angeführt. Wir wollen zum Beispiel eine Instanz der Klasse H1 in ein T-Objekt stecken, um das gewünschte Verhalten zu erreichen (wir sagen dazu auch: ein T-Objekt konfigurieren). Der nachfolgende C# Code zeigt, wie diese Konfiguration implementiert wird:

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

Die durch das Hook-Object-Konstruktionsprinzip ermöglichte Konfiguration eines Objektes durch Komposition mit einem anderen Objekt lässt sich auch wie folgt darstellen: Abbildung 5.22 zeigt schematisch die Anpassung von T durch Komposition des T-Objektes mit einem H1-Objekt. Der Aufruf von Hm() in der Template-Methode bleibt unverändert, unabhängig davon welches spezifische H-Objekt eingesteckt ist.

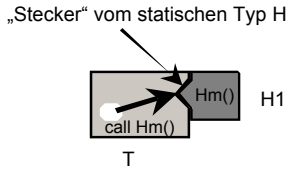

Abbildung 5.22. Komposition eines T-Objektes mit einem H1-Objekt

Als Beispiel für die Anwendung des Hook-Object-Konstruktionsprinzips betrachten wir jenen Teil eines Navigationssubsystems, bei dem die Berechnung der Position und die Ermittlung der Zeit vom zur Verfügung stehenden Satelliten-Navigationssystem abhängt und zur Laufzeit anpassbar sein soll. Abbildung 5.23 zeigt das UML-Diagramm[4], in dem die Template- und Hook-Methoden markiert sind.

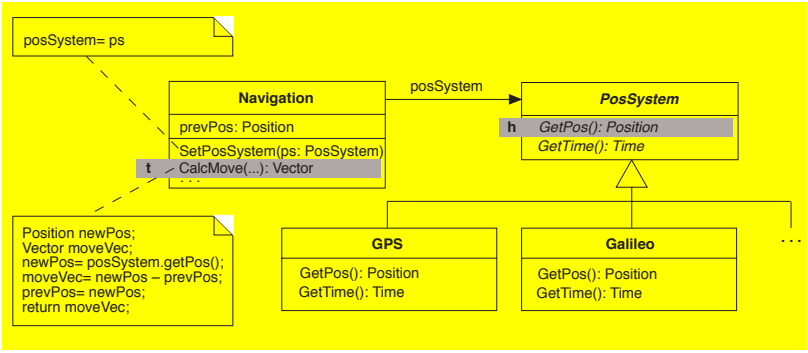

Abbildung 5.23. Anwendung des Hook-Object-Konstruktionsprinzips beim Navigationssubsystem

Durch Komposition einer Instanz der Klasse Navigation mit einer Instanz der Klasse GPS entsteht ein GPS-basiertes Navigationssystem (Abbildung 5.24a). Analog entsteht durch Einstecken einer Instanz der Klasse Galileo in das Navigations-Objekt ein Galileo-basiertes Navigationssystem (Abbildung 5.24b). Zur Komposition wird die Methode SetPosSystem() verwendet.

Abbildung 5.24. GPS-basiertes (a) und Galileo-basiertes (b) Navigationssystem

Erweiterung der Menge der einsteckbaren Komponenten zur Laufzeit

Das Konfigurieren durch Einstecken von H-Objekten funktioniert gut, wenn passende Unterklassen von H vorhanden sind. Aber auch wenn erst eine spezifische Unterklasse von H implementiert werden muss, kann durch Ausnutzen der

[4] In UML ist eine Markierung von Template- und Hook-Methoden nicht vorgesehen. Diese Erweiterung wurde im UML-F-Profil definiert (siehe Fontoura, Pree, Rumpe, 2002).

Möglichkeiten, die sogenannte Meta-Informationen bieten, eine Konfiguration zur Laufzeit erfolgen.

Es soll beispielsweise ein Klasse UMTSTriangulation als Unterklasse von PosSystem implementiert werden, die eine Positionsbestimmung durch Triangulation mittels Mobilfunkstandard UMTS (Universal Mobile Telecommunications System) verwendet. Abbildung 5.25 zeigt ein UML-Diagramm des Klassenbaums, der PosSystem als Wurzelklasse hat und neben der GPS- und Galileo-Klasse auch die Klasse UMTSTriangulation zur Verfügung stellt.

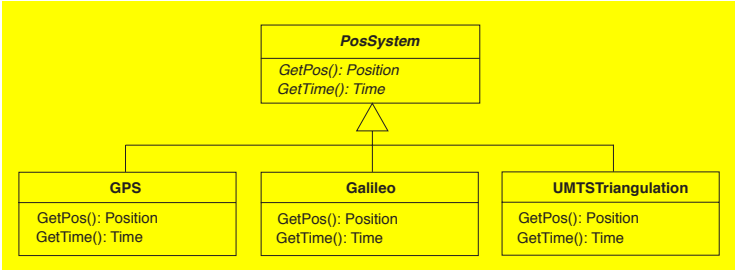

Abbildung 5.25. Unterklassen der abstrakten Klasse PosSystem

Wenn nun eine Instanz von UMTSTriangulation, während das Navigationssystem in Betrieb ist, in das Navigations-Objekt gesteckt werden soll, muss das Laufzeitsystem das dynamische Laden und Linken unterstützen. Die dazu erforderlichen Mechanismen werden durch Bibliotheken und nicht durch objektorientierte Sprachkonstrukte bereitgestellt.

Benutzen wir beispielsweise die .NET-Klassenbibliothek und die Sprache C#, dann würde folgender Code das Einstecken einer Instanz einer zusätzlich implementierten Klasse bewerkstelligen, der allerdings so syntaktisch nicht korrekt ist, da bei der Instanzierung einer Klasse der Name der Klasse und nicht eine Variable, die den Namen als Zeichenkette enthält, angegeben werden darf:

```
Navigation navigation= new Navigation(...);
String nameOfAddtlClass= “UMTSTriangulation“;
Object anObj= ~~new nameOfAddtlClass;~~ // so nicht möglich;
                                        // korrekte Lösung siehe weiter unten
navigation.SetPosSystem((PosSystem)anObj);
```

Die Implementierung geht davon aus, dass der Name der neuen Klasse, von der ein Objekt zur Laufzeit eingesteckt werden soll, auf geeignete Weise als Zeichenkette zur Verfügung gestellt wird. Zum Beispiel könnte der Klassenname über eine Dialogbox in der grafischen Benutzungsschnittstelle eingegeben werden. Mit dieser Information, nämlich dem Namen der Klasse, kann nun eine Instanz dieser (neu hinzugekommenen) Klasse erzeugt werden. Da bei der Objekterzeugung mit ’new’ der Name der Klasse fix im Quelltext anzugeben ist, und die Verwendung einer Variablen nicht möglich ist, muss bei der Implementierung auf sogenannte Meta-Informationen zurückgegriffen werden. Der Begriff *Meta-Information* subsumiert Informationen zur Laufzeit über Objekte und die Klassen von denen sie gebildet wurden. Die Meta-Information ermöglicht es, zur Laufzeit zum Beispiel den Namen der Klasse eines Objektes zu erfahren oder festzustellen, ob ein Objekt direkt oder indirekt zu einer bestimmten Klasse gehört. Der Umfang der durch verschiedene Programmiersprachen und den dazugehörenden Bibliotheken verfügbar gemachten Meta-Information ist unterschiedlich groß. Konzeptionell existiert zu jedem Objekt ein Schattenobjekt, das eine Instanz der speziell dafür vorgesehenen Klasse Type ist (siehe Abbildung 5.26). Tatsächlich existiert aber nur ein solches Schattenobjekt pro Klasse, das allen Instanzen dieser Klasse zugeordnet ist.

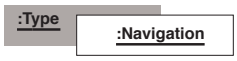

Abbildung 5.26. Objekt und Schattenobjekt(Type-Instanz) für Meta-Information

Die Klasse Type hat 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 Lesen des Wertes einer Instanzvariablen erfolgen. Da wir diese Funktionalität hier nicht benötigen, gehen wir auch nicht näher auf die diversen Aspekte der Meta-Information ein.

Eine Methode der Klasse Activator – CreateInstance() – erlaubt die Erzeugung eines Objektes. Dabei wird von der Situation ausgegangen, dass es zunächst weder ein Schattenobjekt noch ein Objekt gibt. Zuerst wird das einem als Zeichenkette vorliegenden Klassennamen entsprechende Type-Objekt erzeugt (siehe Abbildung 5.27a). Die Klasse Type stellt dazu die statische Methode (= Klassenmethode; also eine Methode, die von der Klasse aufgerufen wird, nicht von einer Instanz) GetType(String className) zur Verfügung. Der Rückgabeparameter von GetType() ist eine Referenz auf die erzeugte Instanz der Klasse Type. Nun kann die erzeugte Instanz der Klasse Type als Parameter für den Methodenaufruf CreateInstance() verwendet werden, wodurch das zugehörige Objekt erzeugt wird (siehe Abbildung 5.27b und Abbildung 5.27c). Der Rückgabeparameter von CreateInstance() ist die Referenz auf das entsprechende Objekt. Der Rückgabeparameter von CreateInstance() ist vom statischen Typ Object (Object ist der Name der Wurzelklasse in der Klassenhierarchie).

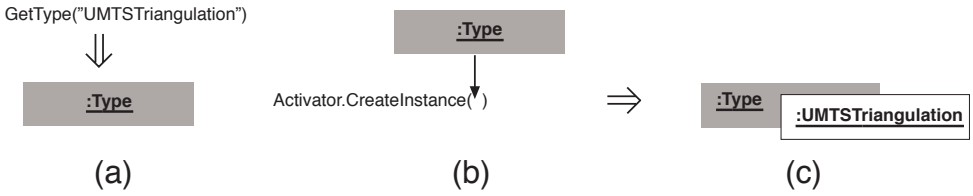

Abbildung 5.27. Erzeugen eines Objektes aus dem als Zeichenkette vorliegenden Klassennamen

Im Quelltext zur Erzeugung einer Instanz von UMTSTriangulation ist zusätzlich das Abfangen von Ausnahmebedingungen vorzusehen. Zum Beispiel könnte es der Fall sein, dass die Klasse, deren Namen als Zeichenkette angegeben ist, nicht existiert. Ein dynamisches Laden wäre dann unmöglich. Der nachfolgende Quelltext 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);
```

Die Flexibilität, das Verhalten beziehungsweise die Konfiguration eines Softwaresystems zur Laufzeit ändern zu können, ist in der Praxis nur in wenigen Fällen erforderlich. In unserem Beispiel würde es bedeuten, dass man, während der Helikopter fliegt, verschiedene Positionierungssysteme verwenden will und man sogar neu hinzugekommene Implementierungen, unmittelbar nachdem sie verfügbar sind, nutzen will.

Das Hook-Object-Konstruktionsprinzip bildet die Grundlage, um einen so hohen Grad an Flexibilität zu realisieren. Nur wenn neu implementierte Komponenten zur Laufzeit eingesteckt werden sollen, ist es notwendig, die Systemarchitektur, wie oben beschrieben, zu gestalten, das heißt, dass auch Klassen instanziiert werden können, deren Namen erst zur Laufzeit bekannt werden.

Der Implementierungsaufwand und die Komplexität steigen, wenn der Grad der Flexibilität gesteigert wird. Wie das oben angeführte Beispiel zeigt, finden wir im Entwurf auf Basis des Hook-Method-Prinzips nur eine Klasse, nämlich GPSNavigation. Um das Navigationssubsystem auf Basis des Hook-Object-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 und eine Klasse GPS die als Unterklasse von PosSystem realisiert wird.

Durch Vorsehen der Möglichkeit, dass die Menge der einsteckbaren Komponenten dynamisch erweiterbar ist, steigt die Komplexität zusätzlich. Das betrifft sowohl die Implementierung der dynamischen Instantiierung als auch den Konfigurationsmechanismus. Zur Unterstüzung der Konfiguration benötigt man beispielsweise eine grafische Benutzungsschnittstelle, die ebenfalls zu entwickeln ist. Die einfache Form, über eine Dialogbox den Namen der einzusteckenden Klasse einzugeben, ist einem Programmierer, aber nicht anderen Benutzern zumutbar. Stattdessen wäre beispielsweise eine geeignete Darstellung der Menge der einsteckbaren Komponenten mit einer Beschreibung ihrer Eigenschaften zweckmäßig, sodass der Benutzer die gewünschte Komponente auf komfortable Weise auswählen kann und damit bestmöglich bei der Systemkonfiguration unterstützt wird.

Ein Softwaresystem wird nicht notwendigerweise besser, wenn mehr Flexibilität geboten wird. Das Ziel soll sein, ein adäquates Maß an Flexibilität vorzusehen sowie Komplexität und Flexibilität im Gleichgewicht zu halten. Das ist leichter gesagt als getan und erfordert Erfahrung im jeweiligen Anwendungsbereich.

Zusammenfassend ist anzumerken, dass der Vorteil des Hook-Object-Konstruktionsprinzips in der Einfachheit der Anpassung (Konfiguration) liegt: durch die Komposition von Objekten wird das Verhalten angepasst. Der Nachteil ist, dass dadurch die Komplexität des Entwurfs und der Implementierung höher sind als beim Hook-Method-Konstruktionsprinzip.

5.2.5 Das Composite-Konstruktionsprinzip

Abbildung 5.28 zeigt die relevanten Aspekte des Composite-Konstruktionsprinzips mit den Methoden AddH() und RemoveH() zur Verwaltung der H-Objekte in einem T-Objekt. Statt 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 Implementierung einer Methode M() in der Klasse T ist dadurch charakterisiert, dass über die vom T-Objekt referenzierten H-Objekte iteriert wird und für jedes H-Objekt die Methode M() aufgerufen wird. Das Kommentarblatt in Abbildung 5.28 zeigt diese Struktur am Beispiel der Template-Methode M().

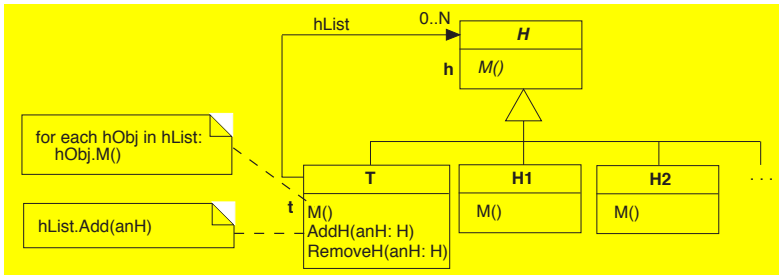

Abbildung 5.28. Composite-Konstruktionsprinzip

Aufgrund der Eigenschaften, die sich aus der Anwendung des Composite-Konstruktionsprinzips ergeben, können Objekthierarchien komponiert werden. Zum Beispiel kann die Objekthierarchie, wie sie in Abbildung 5.29 dargestellt ist, gebildet werden. Dies benötigt man dazu, um hierarchische Beziehungen zwischen Objekten auszudrücken. Beispielsweise kann es zweckmäßig sein, eine Stückliste in Form einer Objekthierarchie zu bilden.

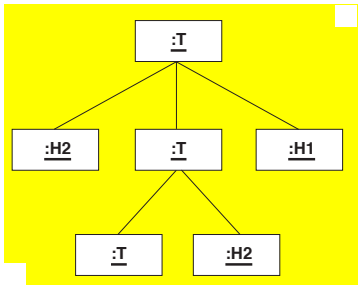

Abbildung 5.29. Objekthierarchie

Der folgende C#-Quelltext 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());
subRoot= new T();
root.AddH(subRoot);
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();
    } // M
```

Es wird jedoch nicht die Methode M() selbst rekursiv aufgerufen, sondern es wird jene Methode M() aufgerufen, die zum jeweiligen H-Objekt gehört. Es ist möglich, dass die Implementierung von M() für ein bestimmtes H-Objekt identisch mit der Implementierung in T ist, weil das H-Objekt eine Instanz der Klasse T ist. Da jedes Objekt quasi die Methoden separat bei sich hat, ist es die gleiche, nicht dieselbe Methode, die aufgerufen wird. Die Methode M() ist somit nicht rekursiv. Sie operiert aber auf einer rekursiven Datenstruktur.

Anhand eines ausführlichen Beispiels wollen wir die Anwendung des Composite-Konstruktionsprinzips illustrieren.

Beispiel: Komposition von Flugmustern

Das Composite-Konstruktionsprinzip lässt sich bei der Konstruktion des Navigationssubsystems zur Komposition von Flugmustern sehr gut anwenden. Zur Illustration der Anwendung des Composite-Konstruktionsprinzips wird ein praxisrelevantes und daher umfangreiches Beispiel herangezogen.

Ein Flugmuster (FlightPattern) besteht aus einzelnen Flugsegmenten im dreidimensionalen Raum, wie beispielsweise einer Linie, einem Kreis, oder einem Kreisbogen. Abbildung 5.30 zeigt das entsprechende UML-Diagramm.

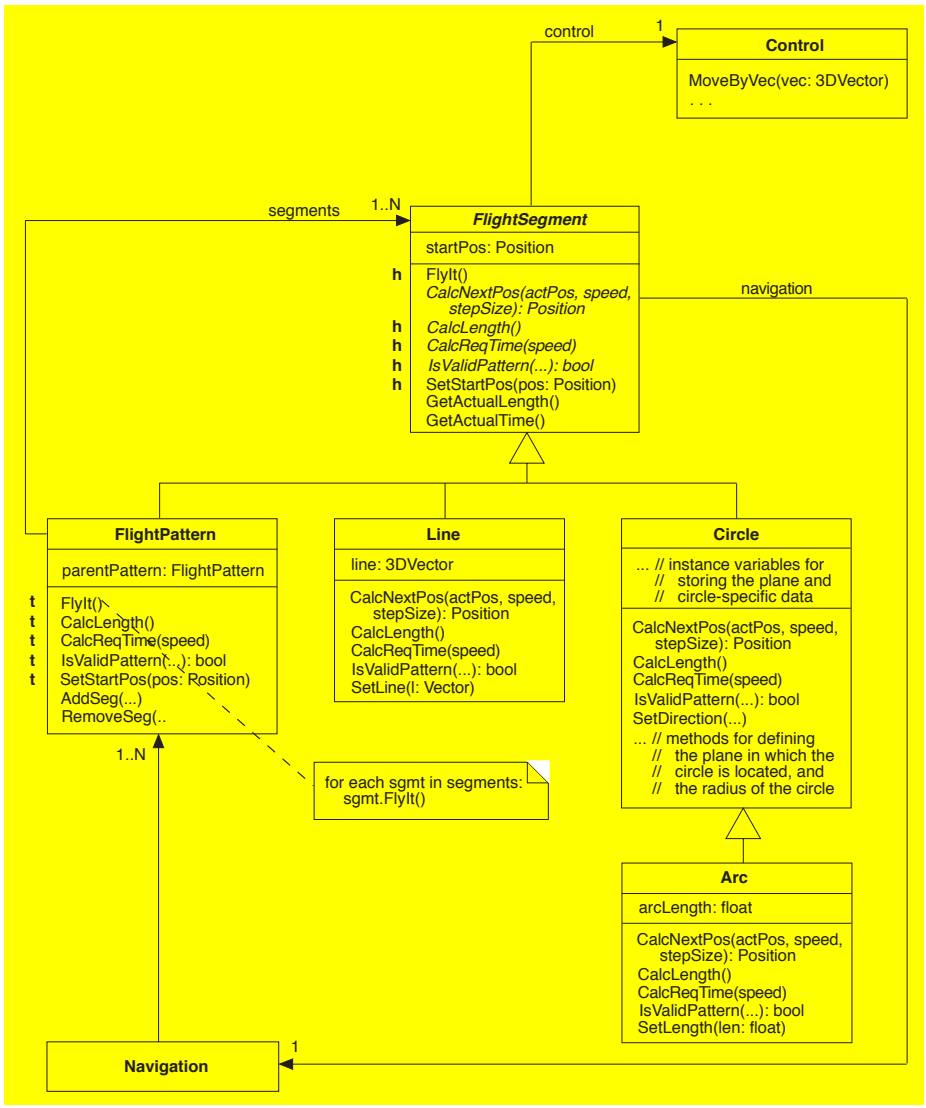

Abbildung 5.30. Anwendung des Composite-Konstruktionsprinzips bei Flugmustern

Ein Flugmuster wird aus einzelnen Flugsegmenten zusammengesetzt. Bei der Definition eines Flugmusters wird zunächst dessen absolute Startposition spezifiziert. Beim Hinzufügen eines Flugsegments durch Aufruf der Methode AddSeg() wird für die Beschreibung des hinzugefügten Flugsegments, die durch relative Positionierung erfolgt, die entsprechende absolute Startposition berechnet. Die absoluten Startpositionen der einzelnen Segmente können nicht explizit in Instanzen der Klassen Line, Circle und Arc festgelegt werden. Sonst könnte es sein, dass bei manueller Spezifikation der Startposition jedes Flugsegments das Flugmuster als Ganzes nicht zusammenhängend ist. Daher berechnet das Flugmuster, das die Wurzel der Objekthierarchie darstellt, aufgrund der absoluten Startposition die jeweiligen Startpositionen der enthaltenen weiteren Flugsegmente und Flugmuster.

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

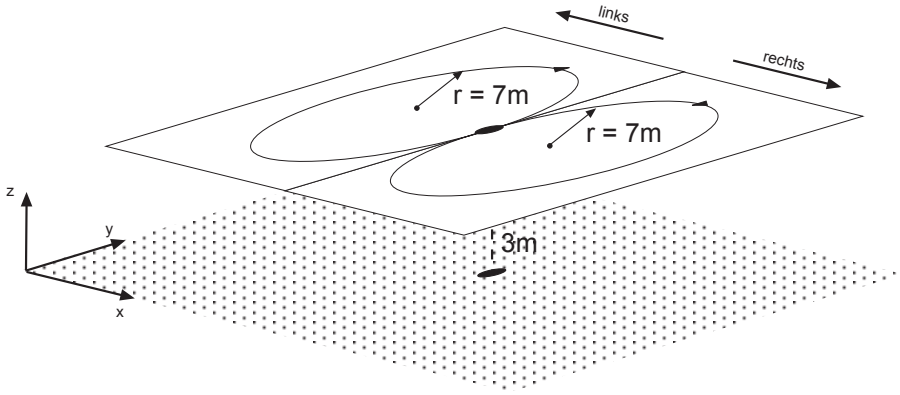

Abbildung 5.31. Ein 8er-Flugmuster in der horizontalen Ebene bestehend aus zwei Kreisen

Der folgende Quelltext definiert dieses Flugmuster. Wir gehen davon aus, dass die Klasse Position die Angabe von geografischen Koordinaten in Längen- und Breitengraden oder (durch überladene Konstruktoren) die Angabe von räumlichen Koordinaten durch ein (x,y,z) Tupel in Längeneinheiten (zum Beispiel in Meter oder Zentimeter) zulässt. Die geografische Position am Boden, über der der Helikopter das Flugmuster beginnen soll, sei mit (gL, gB) gegeben. Die Ebene, in der sich die Kreise befinden, ist als *horizontalPlane* spezifiziert:

```
FlightPattern loop= new FlightPattern();
loop.SetStartPos(new Position(gL, gB) + new Position(0, 0, 3));
loop.AddSeg(new Circle (horizontalPlane, 7, right));  // radius: 7 m; right dir.
loop.AddSeg(new Circle (horizontalPlane, 7, left));  // radius: 7 m; left dir.
```

Ein Implementierungsproblem ergibt sich daraus, dass beim vorliegenden Design 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 könnte aber das 8er-Flugmuster als Teil eines komplexeren Flugmusters vorkommen (siehe Abbildung 5.32). Ein Aufruf der Methode SetStartPos() führt aber zu einer Fehlermeldung, wenn ein Flugmuster eine Komponente einer Objekthierarchie ist, aber nicht deren Wurzel.

Das lässt sich lösen, indem man eine Referenz auf das verwaltende Flugmuster-Objekt in der Instanzvariable parentPattern speichert, 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.

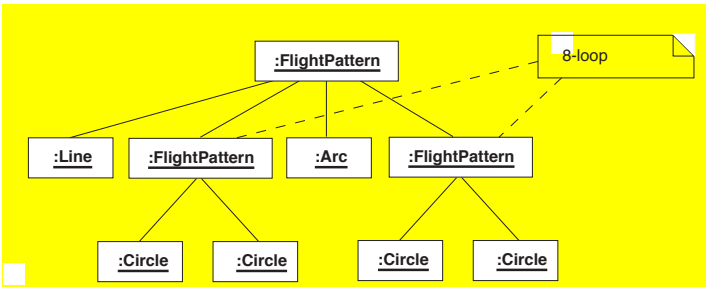

Abbildung 5.32. Komposition von einfachen Flugsegmenten und Flugmustern

Als nächstes wollen wir Einzelheiten zur Implementierung der Template- und Hook-Methoden betrachten. Zum Beispiel soll eine Methode IsValidPattern() prüfen, ob aufgrund der absoluten Koordinaten der Flugsegmente des Flugmusters und der Geländetopologie, das Flugmuster überhaupt fliegbar ist, oder zu einer Bodenberührung führen würde. Bei dem vorher definierten 8er-Flugmuster würde beispielsweise bei der angegebenen Höhe des Startpunktes (3 m) und einem entsprechend großen Kreisradius eine zu starke Neigung der Ebene dazu führen, dass ein Kreis sich mit der Erdoberfläche schneidet (siehe Abbildung 5.33). In diesem Fall soll die Methode IsValidPattern() den Wert false zurück liefern.

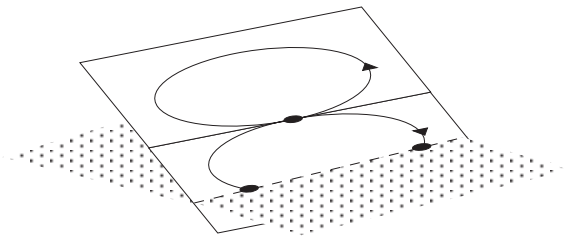

Abbildung 5.33. Ein Beispiel für ein Flugmuster, das zum Absturz führt

Die als Template-Methoden in der Klasse FlightPattern markierten Methoden (siehe Abbildung 5.31) folgen der Implementierungsstruktur der Template-Methoden im Composite-Konstruktionsprinzip; dies ist beispielsweise in Abbildung 5.30 für 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 und die benötigte Zeit für jedes im Flugmuster enthaltene Flugsegment auf. Die Methode IsValidPattern() ruft für jedes im Flugmuster enthaltene Flugsegment die Methode IsValidPattern() auf. Wenn alle Aufrufe den Wert true liefern, kann das Flugmuster ohne Probleme (ungewollte Bodenberührungen) geflogen werden, sonst muss die Ausführung verhindert werden. Wenn ein oder mehrere Flugsegmente den Wert false liefern, bedeutet das, dass das Flugmuster zum Absturz führen würde. Die Methode SetStartPos() geht von der Startposition des Flugmusters aus und berechnet für jedes Flugsegment die aus der Aneinanderreihung der Flugsegmente resultierende Startposition.

Das Zusammenspiel der Methoden FlyIt() und CalcNextPos() in der Klasse Flight-Segment erlaubt es, dass die Methode FlyIt() bereits in der Klasse FlightSegment für alle spezifischen Flugsegmente implementiert werden kann. Die Methode FlyIt() ruft vom Steuerungssystem (Instanz der Klasse Control) die Methode MoveByVec() auf und initiiert dadurch die nächste Flugbewegung indem der dreidimensionale Flugvektor als Parameter dem Methodenaufruf mitgegeben wird.

Der Flugvektor wird aus der aktuellen Position und der nächsten erwünschten Position gebildet. Ein Flugsegement ermittelt die aktuelle Position durch Abfrage der Position beim Navigationssystem. Deshalb ist die Referenz zwischen den Klassen FlightSegment und Navigation im UML-Diagramm dargestellt (siehe Abbildung 5.30). Somit kann ein FlightSegment-Objekt auf das Navigation-Objekt zugreifen. 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 Geschwindigkeit und der Richtung des Helikopters ab, wie weit und in welche Richtung sich der Helikopter in einer bestimmten Zeiteinheit bewegen kann. Aus dem daraus resultierenden möglichen Flugverhalten (wie schnell kann sich der Helikopter aus der aktuellen Stellung wohin im Raum bewegen?) 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 Hook-Method-Konstruktionsprinzip. Die von FlightSegment abgeleiteten Klassen überschreiben entsprechend der Eigenschaften der spezifischen Flugsegmente die Methode CalcNextPos().

Bezüglich der Vererbungsbeziehung zwischen den Klassen Arc und Circle gilt im vorliegenden Entwurf: Die Klasse Arc ist eine Unterklasse von Circle. Das ist eine sinnvolle Entwurfsentscheidung, da ein Kreisbogen weitgehend wie ein Kreis spezifiziert und behandelt wird. Die überschriebenen Methoden, deren Implementierungen sich also vom Kreis unterscheiden, sind in der Klasse Arc angeführt. Die Methode SetLength() ist neu hinzugekommen.

Composite-Konstruktionsprinzip mit verschmolzener T- und H-Klasse

Eine Variationsmöglichkeit des Composite-Konstruktionsprinzips besteht darin, dass die T- und H- Klasse verschmolzen werden (siehe Abbildung 5.34). Dadurch ändert sich die Semantik der Komposition, die grundlegende Eigenschaft, eine Objekthierarchie definieren zu können, bleibt jedoch erhalten.

Sind T und H getrennt, wird die T-Klasse dazu herangezogen, H-Objekte zu verwalten. Wenn die beiden Klassen verschmelzen, ist eine Instanz der gemeinsamen Klasse oder einer Unterklasse davon für beide Aufgaben zuständig. 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 aber implementiert. Die Entscheidung, TH als abstrakte Klasse zu modellieren, hat den Vorteil, dass die verschmolzene Template- und Hook-Methode M() besser gegliedert werden kann: die Implementierung von M() in der abstrakten Klasse TH sorgt für das Iterieren über die Liste der TH-Objekte. In den jeweiligen Unterklassen wird die spezifische Funktionalität hinzugefügt. Zur Iteration wird die Methode aus der Oberklasse verwendet.

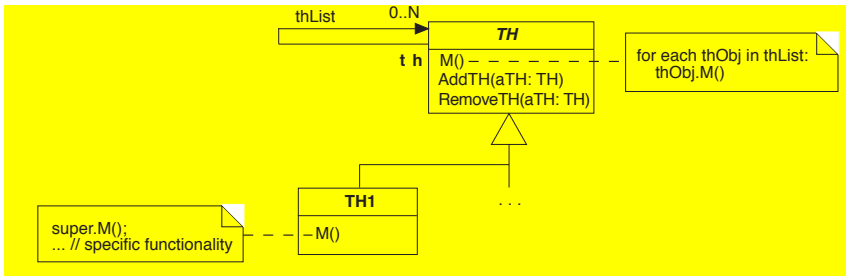

Abbildung 5.34. Variante des Composite-Konstruktionsprinzips

Ein Beispiel, bei dem diese Modellierung angebracht ist, ist die Verwaltung von sogenannten zusammengesetzten Dokumenten (Klasse CompoundDoc in Abbildung 5.35). Ein Textdokument, das verschiedene andere Dokumente wie zum Beispiel Zeichnungen, Ton- oder Videosequenzen enthält, sei für die Verwaltung der enthaltenen Dokumente zuständig und bietet zum Beispiel zusätzlich Funktionalität zum Editieren dieser eingebetteten Dokumente an.

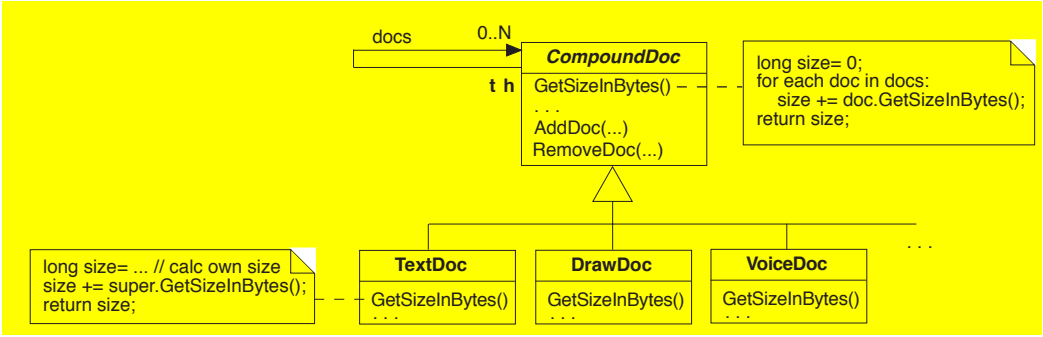

Abbildung 5.35. Verschmelzen von Verwaltung und Funktionalität bei eingebetteten Dokumenten

Der Vorteil des Composite-Konstruktionsprinzips liegt darin, dass auf einfache Weise anpassbare und damit flexibel verwendbare Objekthierarchien gebildet werden können. Es können problemlos Unterklassen der Hook-Klasse definiert und für die Bearbeitung von Objekthierarchien herangezogen werden, ohne die Template-Klasse ändern zu müssen. Objekthierarchien kommen sehr häufig und in vielen Anwendungsbereichen vor. Beispiele dazu sind: in Fenstern gruppierte GUI-Elemente, Stücklisten, Workflows.

Der Nachteil des Composite-Konstruktionsprinzips liegt in der Komplexität der Interaktionen zwischen den in der Hierarchie angeordneten Objekten, um die automatische Iteration über die Baumhierarchie durchzuführen. Die Iteration wird von den Template-Methoden realisiert. Die automatische Iteration über die Objekthierarchie hat allerdings den Vorteil, dass der Benutzer die gesamte Objekthierarchie wie ein einziges Objekt verwenden kann.

5.2.6 Das Decorator-Konstruktionsprinzip

Mit dem Decorator-Konstruktionsprinzip kann das Verhalten eines Objektes durch Komposition mit einem oder mehreren Decorator-Objekten angepasst werden. Die Nützlichkeit des Decorator-Konstruktionsprinzips wird deutlich, wenn wir die Situation betrachten, dass in der Wurzelklasse eines umfangreichen Teilbaumes einer Klassenhierarchie die Funktionalität von Methoden geändert werden sollen.

Abbildung 5.36 zeigt exemplarisch eine Klassenhierarchie mit einer Klasse A als Wurzel eines Teilbaumes. Die Klasse A stellt die Methoden M1() bis M10() zur Verfügung. Die Implementierung der Methoden M3() und M7() soll geändert werden, weil eine modifizierte Funktionalität bereitgestellt werden soll.

Wenn der Quelltext der Klassenbibliothek verfügbar ist, wird oft der vermeintlich einfachste Weg beschritten, nämlich den Quelltext der Klasse A entsprechend zu ändern. Wenn Entwurf und Implementierung nicht genau verstanden werden, kann es dazu kommen, dass durch die vorgenommenen Änderungen ungewollte Nebeneffekte entstehen. Diese Art der Anpassung ist daher nicht ideal.

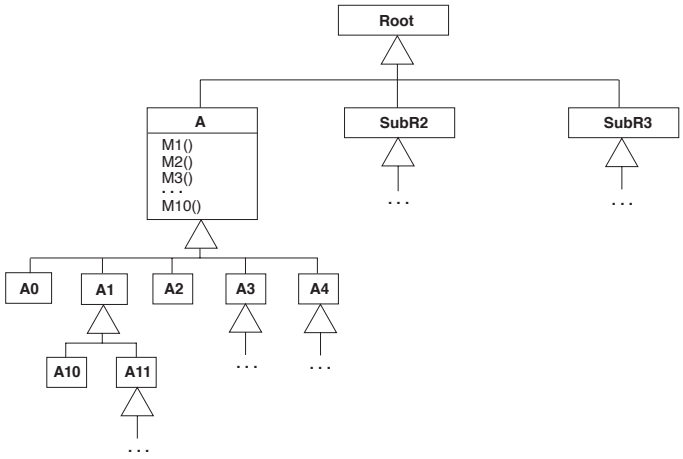

Abbildung 5.36. Beispiel eines Klassenbaums, in dem die Klasse A geändert werden soll

Wenn der Quelltext nicht verfügbar ist, kann der Quelltext der Klasse A auch nicht geändert werden. Um trotzdem den Effekt zu erzielen, dass Objekte der Unterklassen von A das angestrebte geänderte Verhalten bezüglich M3() und M7() aufweisen, kann das Vererbungskonzept dafür herangezogen werden.

Abbildung 5.37 zeigt, dass dazu die Klasse A1 so geändert wird, dass zum Beispiel eine Unterklasse A1m definiert wird, in der die Methoden M3() und M7() entsprechend implementiert (überschrieben) sind. Will man das für mehrere oder gar alle Klassen im Klassenbaum erreichen, erfordert diese Vorgehensweise eine aufwändige Unterklassenbildung. Durch die Hinzunahme von A1m wird nur das Verhalten von A1 in der Klasse A1m modifiziert, nicht aber A1 selbst und die Unterklassen A10, A11, etc. Eine Modifikation von A10 erfordert zum Beispiel die Hinzunahme einer Unterklasse A10m. Analoges gilt für alle Unterklassen von A. Auch diese Art der Anpassung ist daher nicht ideal.

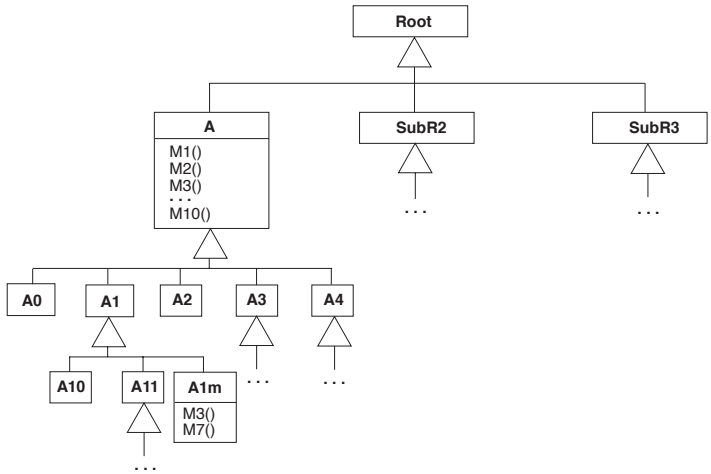

Abbildung 5.37. Anpassung durch Vererbung

In Sprachen, die mehrfache Implementierungsvererbung unterstützen, kann die Anpassung durch sogenannte MixIn-Klassen erfolgen (siehe Abbildung 5.38). Das ändert allerdings nichts an der Tatsache, dass für jede Klasse, deren Verhalten angepasst werden soll, eine Unterklasse gebildet werden muss. Die anzupassenden

Implementierungen der Methoden M3() und M7() sind allerdings in einer Klasse zusammengefasst.

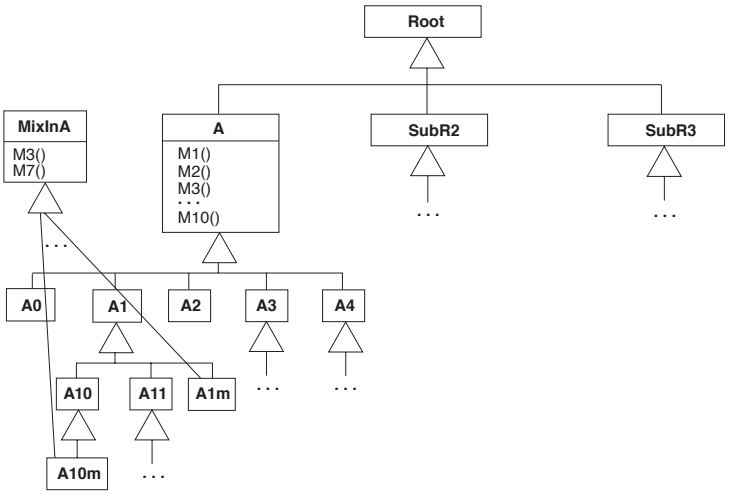

Abbildung 5.38. Anpassung durch Mehrfachvererbung

Mit Hilfe des Decorator-Konstruktionsprinzips wird das Problem auf elegante Weise durch Komposition von Objekten anstatt durch Vererbung gelöst. Abbildung 5.39a zeigt das grundlegende Konzept. In der Klasse WrapperOfA werden alle Methoden von A überschrieben, indem der Methodenaufruf jeweils an ein über die Instanzvariable wrappedA referenziertes Objekt delegiert wird. Die Methode SetWrappedA() weist der Instanzvariable wrappedA die Referenz auf ein A-Objekt zu.

Da WrapperOfA eine Unterklasse von A ist, kann überall dort, wo ein Objekt vom statischen Typ A gefordert wird, eine Instanz der Klasse WrapperOfA verwendet werden. Da die Instanzvariable wrappedA den statischen Typ A hat, kann sie auf jedes Objekt einer Unterklasse von A verweisen.

Abbildung 5.39b zeigt, dass durch die Komposition einer Instanz von WrapperOfA mit einer Instanz von A4 eine Indirektion bei Methodenaufrufen einführt wird: Jeder Methodenaufruf in einer WrapperOfA-Instanz wird an die A4-Instanz weitergeleitet. Die Komposition der beiden Objekte kann wie ein A-Objekt behandelt werden.

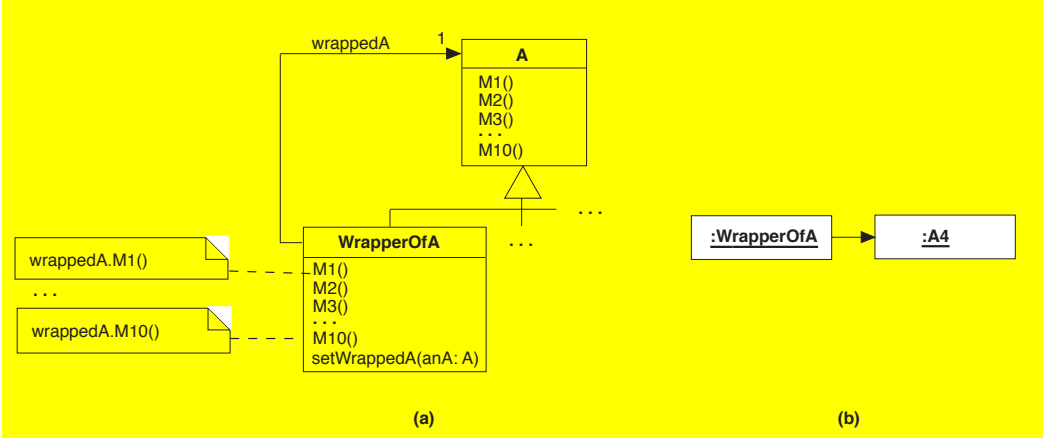

Abbildung 5.39. Weiterleitung von Methodenaufrufen (a) und Decorator-Komposition (b)

Die Klasse WrapperOfA wird zu einem Decorator von A, wenn in den weitergeleiteten Methodenaufrufen auch zusätzlich das Verhalten verändert wird. Man kann sich das so vorstellen, dass ein „Filter“ vor ein A-Objekt gestellt wird.

Das UML-Diagramm in Abbildung 5.40 illustriert die verallgemeinerte Anwendung des Decorator-Konstruktionsprinzips. Wie beim Composite-Konstruktionsprinzip sind die Namen der Template-Methode und der Hook-Methode identisch. Im Beispiel aus Abbildung 5.39 entspricht die Decorator-Klasse WrapperOfA der T-Klasse, die Klasse A der H-Klasse.

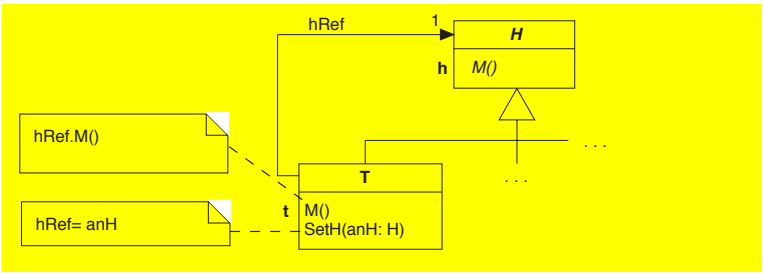

Abbildung 5.40. Eine Decorator-Klasse für H-Objekte

Eine Instanz des Decorators (Filters) T zusammen mit der Instanz einer Unterklasse von H kann von Klienten wie ein H-Objekt verwendet werden. Allerdings wird das Verhalten des H-Objektes dadurch modifiziert. Der Decorator (Filter) kann einer Instanz einer beliebigen Unterklasse von H vorangestellt werden.

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

Dadurch ist es möglich, durch Komposition von Objekten das Verhalten von H-Objekten zu verändern beziehungsweise anzupassen. Dazu muss lediglich *eine* Decorator-Klasse definiert werden. Das ist wesentlich eleganter als der aufwändige Weg, dieses Problem mittels Vererbung zu lösen.

Wenn mehrere Decorator-Klassen erforderlich sind, ist es sinnvoll, die Weiterleitung der Methodenaufrufe in einer gemeinsamen Basisklasse zu implementieren. Abbildung 5.41 zeigt eine Klasse Decorator, die sämtliche Methodenaufrufe an ein von wrappedA referenziertes Objekt weiterleitet. Die Unterklasse Decorator1

überschreibt die Methoden M3() und M7(). Die Unterklasse Decorator2 überschreibt die Methode M9().

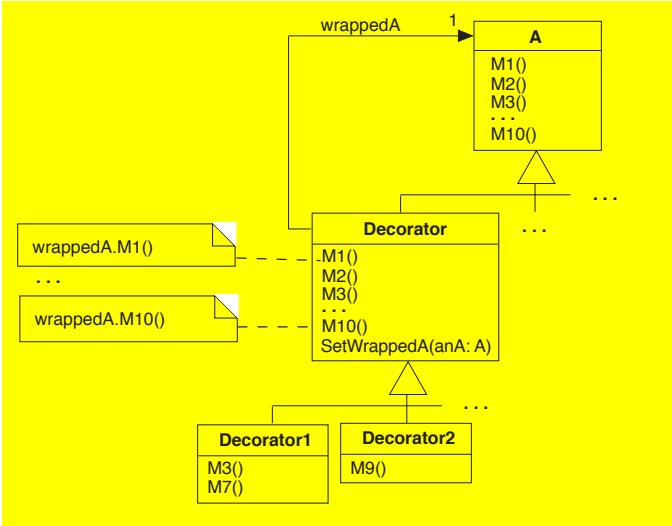

Abbildung 5.41. Mehrere Decorator-Klassen für A-Objekte

In Abbildung 5.42 werden exemplarisch zwei Kompositionen herausgegriffen. Eine Instanz der Klasse A2 hat den Decorator1 vorangestellt (siehe Abbildung 5.42a). Der Instanz der Klasse A4 sind sowohl Decorator1 als auch Decorator2 vorangestellt (siehe Abbildung 5.42b). Die Beispiele illustrieren, dass das Verhalten durch Objektkomposition für einzelne Objekte individuell angepasst werden kann.

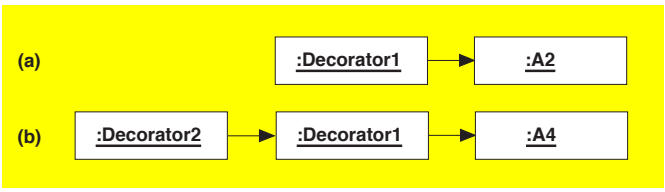

Abbildung 5.42. Ein (a) und zwei (b) Decorator-Objekte vor A-Objekten

Anhand eines ausführlichen Beispiels wollen wir die Anwendung des Decorator-Konstruktionsprinzips illustrieren.

Beispiel: Abrundung der Übergänge bei Flugmustern

Damit die Übergänge zwischen den Flugsegmenten in einem Flugmuster, der Praxis entsprechend, harmonisch gestaltet werden können, bedienen wir uns zur Modellierung der dazu erforderlichen Systemteile des Decorator-Konstruktionsprinzips. Abbildung 5.43 zeigt, was mit einem „harmonischen Übergang“ gemeint ist, das heisst, wie aus einem Flugmuster, das zwei in einem rechten Winkel angeordnete Geraden als Flugsegmente enthält, ein harmonisches Flugmuster wird.

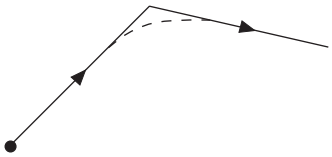

Abbildung 5.43. Abrundung eines aus zwei Geraden bestehenden Flugmusters

Wir entscheiden uns, eine Klasse Smoother einzuführen, die dazu dient, Flugsegmente zu dekorieren. Wenn ein Flugsegment mit einem Smoother-Objekt dekoriert ist, soll der Übergang zum nächsten Flugsegment, falls vorhanden, abgerundet werden. Der Decorator Smoother modifiziert dazu die Methode CalcNextPos(), damit mittels Smoother die Positionskoordinaten berechnet werden können, und zwar so, dass eine Abrundung erfolgen kann. Das nachfolgende Flugsegment muss dazu bekannt sein. Eine Smoother-Instanz benötigt daher den Zugriff auf das Flugmuster-Objekt, in dem die abzurundenden Flugsegmente enthalten sind.

Wir ändern daher den Entwurf so, dass die Instanzvariable parentPattern der Klasse FlightPattern in die Klasse FlightSegment transferiert (das heisst faktorisiert) wird. Wenn ein Flugsegment zu einem Flugmuster hinzugefügt wird, wird die Instanzvariable entsprechend gesetzt. Darüber hinaus ist es notwendig, eine Methode GetNextSeg() in der Klasse FlightPattern einzuführen, die das nachfolgende Flugsegment zu dem als Parameter mitgegebenen Flugsegment liefert. Abbildung 5.44 zeigt die relevanten Ausschnitte der Systemarchitektur (Klassenhierachie). Die Tatsache, dass Änderungen der Klassen FlightPattern und FlightSegment nötig sind, zeigt, dass es in der Praxis häufig nötig ist, die Verwendung des Decorator-Konstruktionsprinzips von vorne herein vorzusehen. Eine elegante Änderung von Methoden einer Klasse an der Wurzel der Klassenhierarchie ohne Änderungen ist nur in seltenen Fällen möglich. Dieser Aspekt wird unten näher diskutiert.

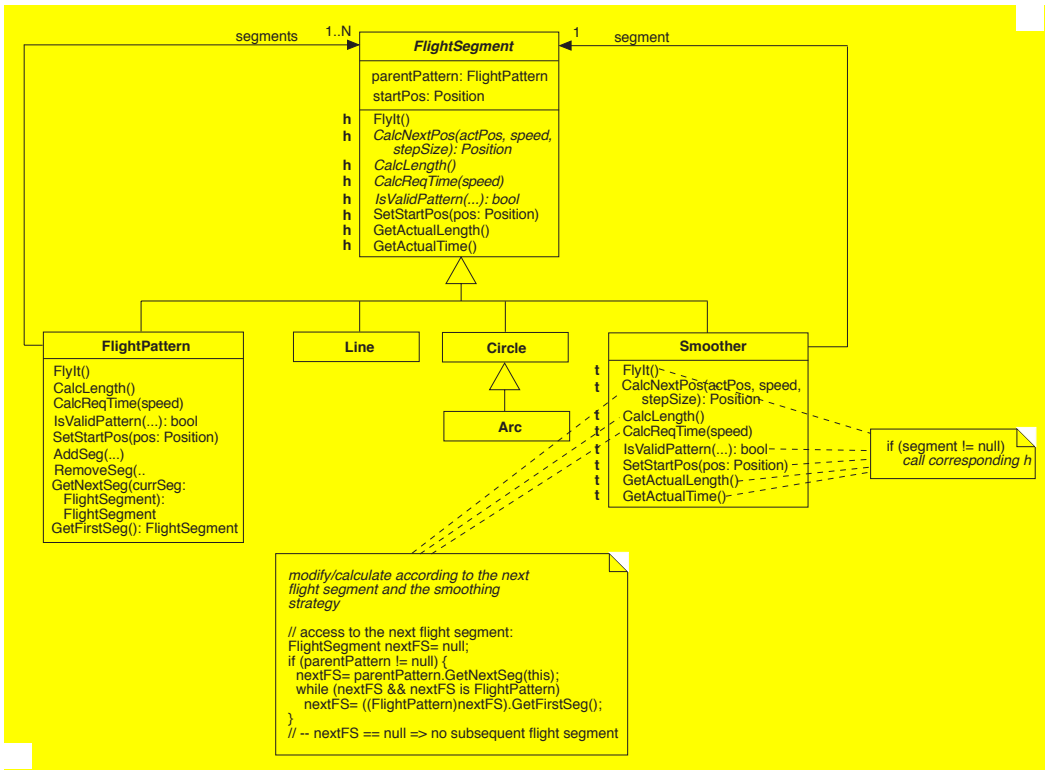

Abbildung 5.44. Die Klasse Smoother als Decorator von FlightSegment

Der folgende Quelltext illustriert, wie ein dreieckiges Flugmuster, das aus drei Geraden besteht, und in dem der Übergang zwischen den Geraden abgerundet werden soll, gebildet wird. Dazu werden die beiden Instanzen der Klasse Line mit

Smoother dekoriert (würde mit der weiteren Instanz der Klasse Line auch ein Smoother verbunden, hätte er keine Wirkung, da es kein nachfolgendes Flugsegment gibt). Wir nehmen an, dass die Klasse Smoother einen Konstruktor hat, dessen Parameter vom statischen Typ FlightSegment ist. Die mitgegebene Objektreferenz wird im Konstruktor der Instanzvariable segment zugewiesen. Das referenzierte Objekt ist das mit dem Smoother dekorierte Flugsegment.

```
FlightPattern triangle= new FlightPattern();
triangle.SetStartPos(...);
triangle.AddSeg(new Smoother(new Line(...)));
triangle.AddSeg(new Smoother(new Line(...)));
triangle.AddSeg(new Line(...));
```

Rahmenbedingungen für die Anwendung des Decorator-Konstruktionsprinzips

Die zu Beginn dieses Abschnittes zur Illustration der Nützlichkeit des Decorator-Konstruktionsprinzips angeführte Problemstellung, dass in der Wurzelklasse eines (umfangreichen) Teilbaumes einer Klassenhierarchie die Funktionalität von Methoden geändert werden müssen, ist durch Anwendung des Decorator-Konstruktionsprinzips in manchen Fällen elegant lösbar, also ohne Änderung der bestehenden Klassen. Die Anpassung durch Objektkomposition erfordert dann lediglich die Definition einer entsprechenden Klasse.

Allerdings erfordert die Anwendung des Decorator-Konstruktionsprinzips die Erfüllung einer Forderung an H, 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 Unterklassen, die die Signatur der Wurzelklasse erweitern, in den Decorator-Klassen nicht berücksichtigt werden können.

Um die Erfüllung dieser Forderung sicherzustellen, ist es notwendig, dass die Gemeinsamkeiten aller Unterklassen von H in die Wurzelklasse transferiert, das heißt faktorisiert werden. Bei vielen Klassenbibliotheken ist diese Forderung nicht erfüllt. Die Anwendung des Decorator-Konstruktionsprinzips ist daher in solchen Fällen nicht immer in vollem Umfange möglich.

Auch in unserem Systementwurf, in dem die Klasse Smoother auf Basis des Decorator-Konstruktionsprinzips definiert ist, ist diese Forderung nicht erfüllt. 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 jedoch nicht adäquat ist, diese Methoden in die Klasse FlightSegment zu verschieben, belassen wir den Entwurf wie er ist.

Als Folge dieser Entscheidung müssen aber die spezifischen Methoden für die erwähnten Objekte – bei Line SetLine(); bei Circle SetDirection(), sowie die Methoden zur Festlegung der Ebene, in der der Kreis liegt: bei FlightPattern AddSeg() und RemoveSeg() – explizit aufrufen werden, da sie von einer Smoother-Instanz nicht weitergeleitet werden können:

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

Wäre die erwähnte Forderung erfüllt, 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 Eigenschaften nur über den Konstruktor der jeweiligen Klasse angeben zu lassen:

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

Das Decorator-Konstruktionsprinzip kann dazu herangezogen werden, Klassen nahe der Wurzel des Klassenbaumes leichtgewichtiger zu machen. Funktionalität, die nicht in allen Klassen benötigt wird, wird in Decorator-Klassen implementiert. Nur jene Objekte, welche die spezielle Funktionalität benötigen, erhalten diese durch Komposition mit der entsprechenden Decorator-Instanz. Das Decorator-Konstruktionsprinzip kann sowohl beim (Erst-)Entwurf einer Klassenhierarchie als auch bei der Erweiterung von Klassenhierarchien nutzbringend eingesetzt werden.

Ein weiteres typisches Beispiel für die Anwendung des Decorator-Konstruktionsprinzips ist die Modellierung des für den sogenannten Clipping-Mechanismus bei GUI-Bibliotheken zuständigen Teils der Systemarchitektur. Da der Clipping-Mechanismus – das Zuschneiden eines GUI-Elements auf dessen festgelegte Größe – nicht für alle GUI-Elemente benötigt wird, ist es sinnvoll, den Clipping-Mechanismus nicht in der Wurzel des Teilbaumes vorzusehen, sondern eine Decorator-Klasse Clipper einzuführen.

5.2.7 Zusammenfassung der Merkmale der Konstruktionsprinzipien

In der Tabelle 5.1 sind wichtige Merkmale der oben beschriebenen Konstruktionsprinzipien zusammengefasst. Das Chain-Of-Responsibility-Konstruktionsprinzip wird mit COR abgekürzt. Beim Hook-Method-Konstruktionsprinzip sind die Template- und die Hook-Methoden in einer Klasse, während sie bei den Hook-Object-, Composite- und Decorator-Konstruktionsprinzipien getrennt sind. Wir nennen die Klasse, welche die Template-Methode(n) enthält, T und die Klasse, die die Hook-Methode(n) enthält, H. Bei den Konstruktionsprinzipien Composite und Decorator erbt H von T. Im COR-Konstruktionsprinzip wird die Vererbungsbeziehung dadurch aufgelöst, dass T und H zu einer Klasse verschmelzen. Ebenso verschmelzen Template- und Hook-Methode.

Was die Anzahl der beteiligten Objekte betrifft, unterscheiden sich die Konstruktionsprinzipien wie folgt: Beim Hook-Method-Konstruktionsprinzip wird eine Instanz jener Klasse angelegt, die die passende Hook-Methode enthält. Beim Hook-Object-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 Verwendung rekursiver Konstruktionsprinzipien 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, darauf zu achten, dass keine zirkulären Objektbeziehungen innerhalb der Graphen definiert werden. Das würde eine Endlosschleife in der Template-Methode zur Folge haben.

Schließlich wirkt sich die Wahl des Konstruktionsprinzips auch auf die Anpassungsart aus, die erforderlich ist, um das gewünschte Verhalten eines T-Objektes in einem bestimmten Kontext sicherzustellen. Bei allen Konstruktionsprinzipien muss die Hook-Methode in Unterklassen der H-Klasse entsprechend überschrieben werden. Vererbung ist somit Voraussetzung für eine Anpassung. Bei Verwendung der Konstruktionsprinzipien Hook-Object, Composite, Decorator und COR erfolgt die Anpassung des Verhaltens des T-Objektes durch Komposition mit einem oder beliebig vielen H-Objekten.

		Konstruktionsprinzipien				
		Hook-Method	Hook-Object	Composite	Decorator	COR
Charakteristika von Template- und Hook-Methoden	Positionierung	T() und H() in einer Klasse	T() und H() in getrennten Klassen			T() = H()
	Namensgebung	verschiedene Namen		Namen von T() und H() sind gleich		
	Vererbungsbeziehung	n.a.		H erbt von T		T = H
Anzahl der beteiligten Objekte		1	1(T) + 1(H) oder 1(T) + N(H)	N Objekte, die wie ein Ojekt verwendet werden		
Anpassung		durch Vererbung und Instanziierung der entsprechenden Klasse	durch Komposition (bei Bedarf zur Laufzeit)			

Tabelle 5.1. Merkmale der Konstruktionsprinzipien von Produktfamilien

5.3 Konstruktionsprinzipien und Entwurfsmuster

In den 1990er-Jahren sind zahlreiche Vorschläge für die Verwendung spezieller *Entwurfsmuster* (im Englischen als *Design Patterns* bezeichnet) zur Gestaltung objektorientierter Systemarchitekturen publiziert und auf Konferenzen diskutiert worden. Zu einer einheitlichen Bewertung oder gar Normung kam es nicht. Eine hohe Publizität und breite Akzeptanz erlangten die von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (Gamma et al. 1995) beschriebenen 23 Entwurfsmuster.

Im folgenden betrachten wir den Zusammenhang zwischen den fünf in den vorhergehenden Abschnitten beschriebenen Konstruktionsprinzipien für objektorientierte Produktfamilien und den Entwurfsmustern von Gamma et al. (1995) Es würde den Rahmen des Buches sprengen, alle 23 Entwurfsmuster von Gamma et al. im Detail zu diskutieren. Im folgenden wird daher auf diesen Zusammenhang und die daraus resultierenden softwaretechnischen Aspekte überblicksartig eingegangen.

Mehr als die Hälfte der 23 Entwurfsmuster beschreiben, wie objektorientierte Produktfamilien entworfen und implementiert werden sollen. Abbildung 5.45 zeigt im Überblick die 14 der vorgeschlagenen Entwurfsmuster, die auf einem der vorgestellten Konstruktionsprinzipien für objektorientierte Produktfamilien beruhen. Weiter unten greifen wir beispielhaft zwei Entwurfsmuster heraus, um die Zusammenhänge im Detail zu erläutern. Die drei rekursiven Konstruktionsprinzipien finden sich auch als Entwurfsmuster im Vorschlag von Gamma et al. Dies wird in Abbildung 5.45 entsprechend zum Ausdruck gebracht.

Die restlichen 9 der 23 Entwurfsmuster beziehen sich auf spezielle Aspekte der Softwareentwicklung, die entweder für die Gestaltung von Produktfamilien irrelevant sind oder allgemein gültige Aspekte betreffen, die unabhängig davon sind, ob eine Produktfamilie oder eine spezielle Applikation entwickelt werden soll. Beispielsweise behandelt das Facade-Entwurfsmuster die Modularisierung eines Softwaresystems, das Singleton- Entwurfsmuster beschreibt, wie in klassenbasierten Sprachen wie zum Beispiel C++ sichergestellt werden kann, dass nur exakt eine Instanz einer Klasse erzeugt wird. Da diese 9 Entwurfsmuster in keinem direkten Bezug zur Entwicklung von Produktfamilien stehen, werden diese hier nicht weiter betrachtet. Wir verweisen den interessierten Leser auf ihre Beschreibung in Gamma et al. (1995).

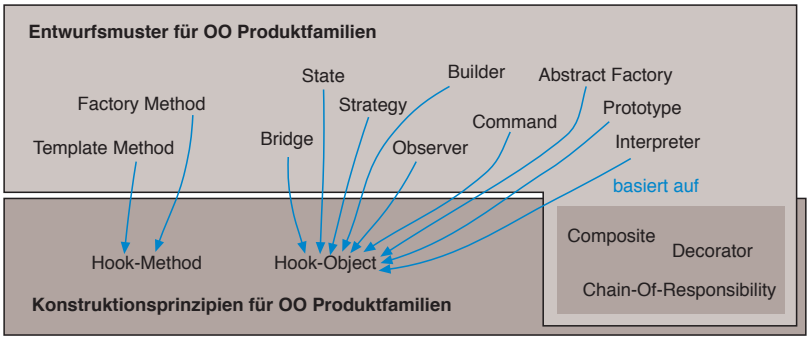

Abbildung 5.45. Zusammenhang zwischen Entwurfsmustern und Konstruktionsprinzipien

Im folgenden greifen wir beispielhaft die Entwurfsmuster Factory Method 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 Hook-Method-Konstruktionsprinzip zugrunde.

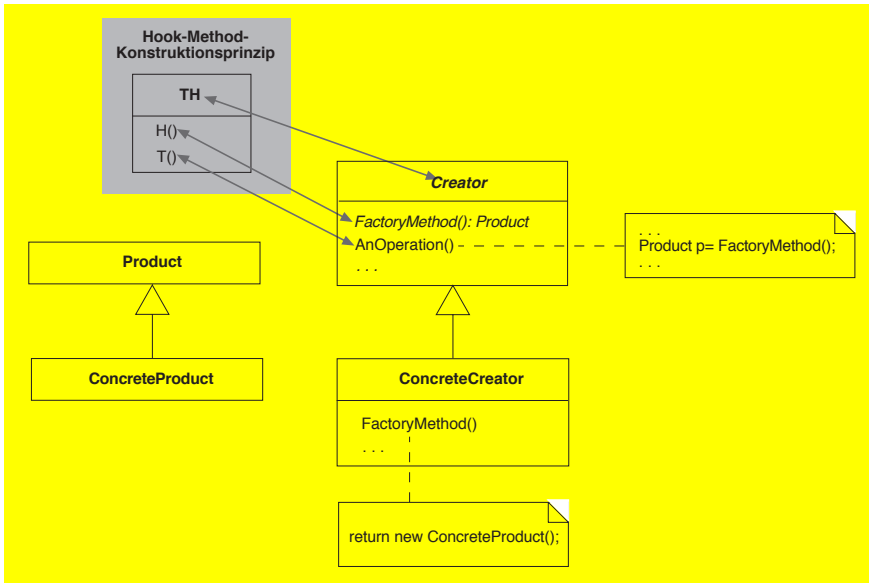

Abbildung 5.46. Template- und Hook-Methode im Entwurfsmuster Factory Method

Wenn man das Hook-Method-Konstruktionsprinzip kennt, weiß man, welche Flexibilität damit erreichbar ist und wie die Anpassung (das heißt die Festlegung des konkreten Verhaltens von T-Objekten) erfolgt, unabhängig davon, welche Funktionalität die Template- und Hook-Methoden bereitstellen. Im Fall des Entwurfsmusters Factory-Method muss die Hook-Methode FactoryMethod() in einer Unterklasse von Creator überschrieben werden, um zu definieren, welche spezifische Product-Klasse instantiiert werden soll.

Der Name und die Funktionalität der Hook-Methode drücken aus, welcher Aspekt bei einem Entwurfsmuster flexibel gehalten wird. Im Beispiel Factory-Method geht es darum, die Objekterzeugung flexibel zu halten. Dazu wird das Hook-Method-Konstruktionsprinzip angewendet. Dieses Konstruktionsprinzip ist von Vorteil, wenn bei Verwendung einer bestimmten Programmiersprache (wie C++), die Bereitstellung von Meta-Informationen nicht ausreichend oder nicht einheitlich unterstützt wird. Der Nachteil des Hook-Method-Konstruktionsprinzips ist, dass eine Unterklasse von Creator definiert werden muss, womöglich nur, um die Hook-Methode FactoryMethod() zu überschreiben.

Dasselbe Ziel, nämlich die Objekt-Erzeugung in der Klasse Creator flexibel zu halten, kann bei der Verwendung von Sprachen wie Java und C# eleganter dadurch erreicht werden, dass der Name der Klasse, von der ein Objekt erzeugt werden soll, mittels Parameter der Methode AnOperation() übergeben wird (siehe Abbildung 5.47). Durch die Methode AnOperation() wird die Klasse geladen und ein Objekt davon erzeugt. Was den Objekttyp des erzeugten Objektes betrifft, ist zu prüfen, ob er dem Typ Product entspricht. Damit wird derselbe Effekt erzielt wie bei Anwendung des Hook-Method-Konstruktionsprinzips, jedoch mit dem Vorteil, dass keine Unterklasse mehr erforderlich ist. Der Nachteil ist, dass die statische Typprüfung dabei umgangen wird. Der Rückgriff auf Meta-Informationen anstatt der Verwendung des Hook-Method-Konstruktionsprinzips ist lediglich für diesen Fall möglich, da die Möglichkeit der Objekterzeugung zur Laufzeit auf Basis der verfügbaren Meta-Informationen besteht.

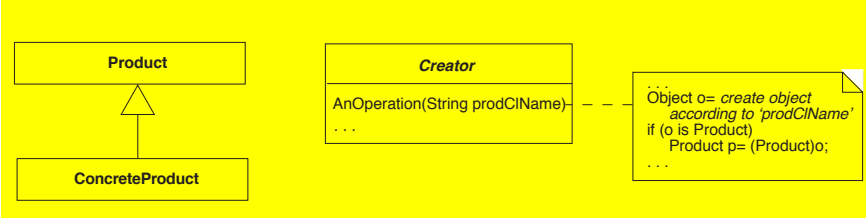

Abbildung 5.47. Dynamische Objekterzeugung als Alternative zum Entwurfsmuster Factory Method

Eine weitere Alternative, die keine Unterklassenbildung erfordert, basiert ebenfalls auf dem Hook-Method-Konstruktionsprinzip. 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 Objekt-Erzeugung flexibel zu halten. Es bietet im Vergleich zum Entwurfsmuster Factory Method mehr Flexibilität, da es auf dem Hook-Object-Konstruktionsprinzip basiert. Das Entwurfsmuster Abstract Factory ist ein Beispiel dafür, wie ein Entwurfsmuster, das dem Hook-Method-Konstruktionsprinzip entspricht (nämlich Factory Method) in ein Entwurfsmuster, das dem Hook-Object-Konstruktionsprinzip entspricht, transformiert werden kann. Es genügt, die Hook-Methode in eine separate Klasse oder Schnittstelle zu verschieben. In diesem Beispiel wird die Hook-Methode FactoryMethod() in eine separate Klasse (AbstractFactory) verschoben, die mit der Creator-Klasse abstrakt gekoppelt ist (siehe Abbildung 5.48).

Anmerkung: Im Buch von Gamma et al. (1995) ist die Transformation durch Umbenennungen nicht offensichtlich. Im Entwurfsmuster Abstract Factory wurde die Hook-Methode von FactoryMethod() in CreateProduct() umbenannt und die Klasse Creator ist in Client umbenannt worden.

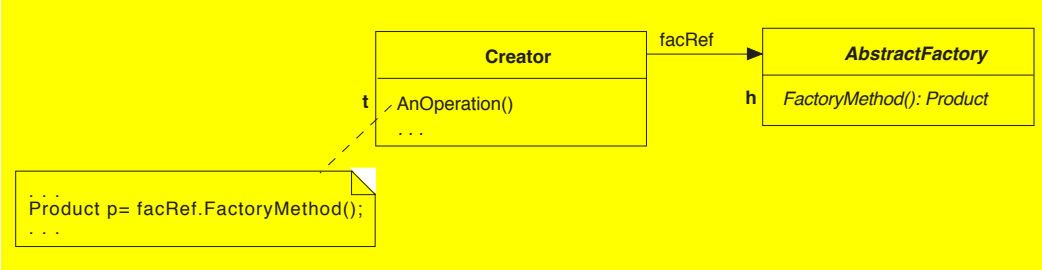

Abbildung 5.48. Transformation des Entwurfsmusters Factory Method in das Entwurfsmuster Abstract Factory

Namensgebung für Entwurfsmuster

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

Analysiert man die Entwurfsmuster von Gamma et al., 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, AbstractFactory. Analoges gilt für die Entwurfsmuster State, Strategy, Builder, Observer, Command, Prototype und Interpreter.

Diese Art der Namensgebung ist sinnvoll und wird daher auch für die Entwicklung neuer, spezieller Entwurfsmuster empfohlen. Wir postulieren also folgende Regel: die Hook-Semantik bestimmt den Namen des Entwurfsmusters. Damit kann eine Systematik zur Benennung von objektorientierten Entwurfsmustern eingeführt werden.

Dahl, O. und Nygard, K: SIMULA—An ALGOL-based Simulation Language: CACM 9(9) 671-678 (1966).

E. Gamma, R. Helm, R. Johnson und J. Vlissides: Design Patterns—Elements for Reusable Object-Oriented Software, Addison-Wesley, 1995

M. Fontoura, W. Pree, B. Rumpe: The UML Profile for Framework Architectures, Pearson Education, 2002

M. Fayad, D. Schmidt und R. Johnson (Hrsg.): Domain-Specific Application Frameworks: Manufacturing, Networking, Distributed Systems, and Software Development, Wiley, 1999

M. Reiser und N. Wirth: Programming in Oberon—Steps Beyond Pascal and Modula, Addison-Wesley, 1992