

# Konstruktion anpassbarer Software

O.Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Pree

Software & Systems Research Center  
[cs.uni-salzburg.at](http://cs.uni-salzburg.at)

# Inhalt

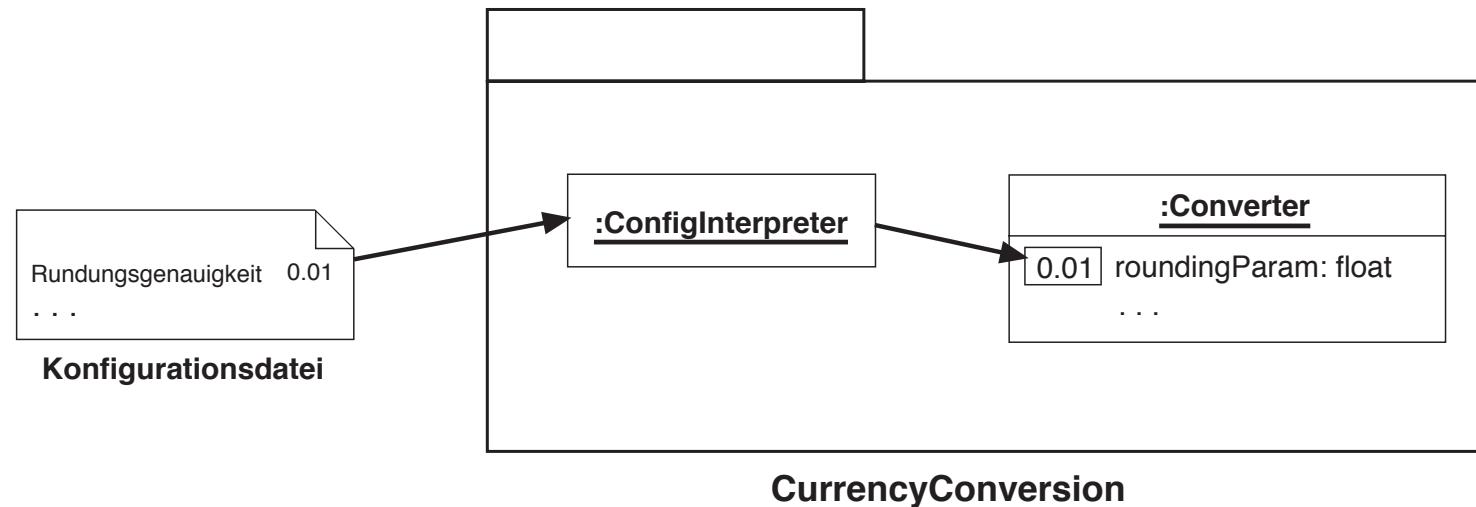
- Konfigurationsparameter
- Konzepte und Konstruktionsprinzipien für anpassbare, objektorientierte Produktfamilien
- Konstruktionsprinzipien und Entwurfsmuster

# Konfigurationsparameter

# Definition

- Konfigurationsparameter sind in *Parametrierungsdateien* (= *Konfigurationsdateien*) abgelegt.
- Die *Konfigurationsparameter* entsprechen persistenten, globalen (= statischen) Variablen.

# Beispiel



Legende:



Softwarekomponente

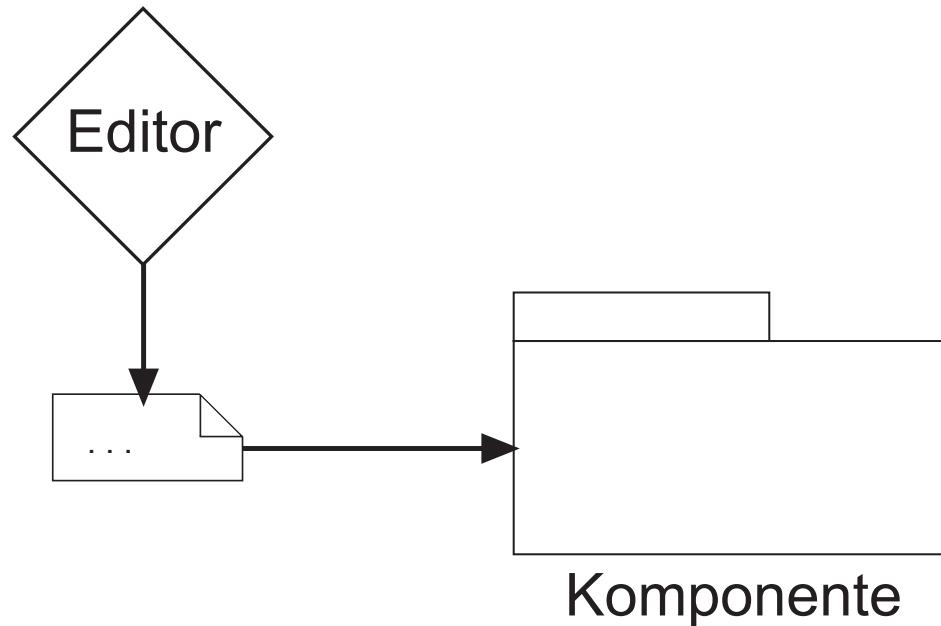


Objekt



externe Datei

# Erstellen der Konfigurationsdatei mit einem Editor

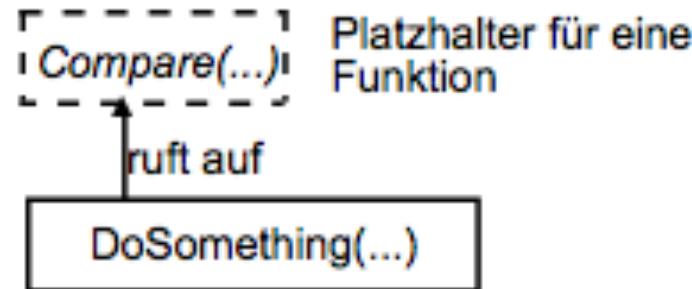


Beispiel:

GUI-Konfigurationsdatei= Ressource-Datei

Visuell, interaktive Erstellung mit Hilfe eines Ressource-Editors.

# Callback-Style of Programming (I)



Eine als Parameter übergebene Funktion wird von DoSomething() aufgerufen. Das erklärt, warum dafür der Begriff Callback-Style of Programming eingeführt wurde:

Es lässt sich 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)**.

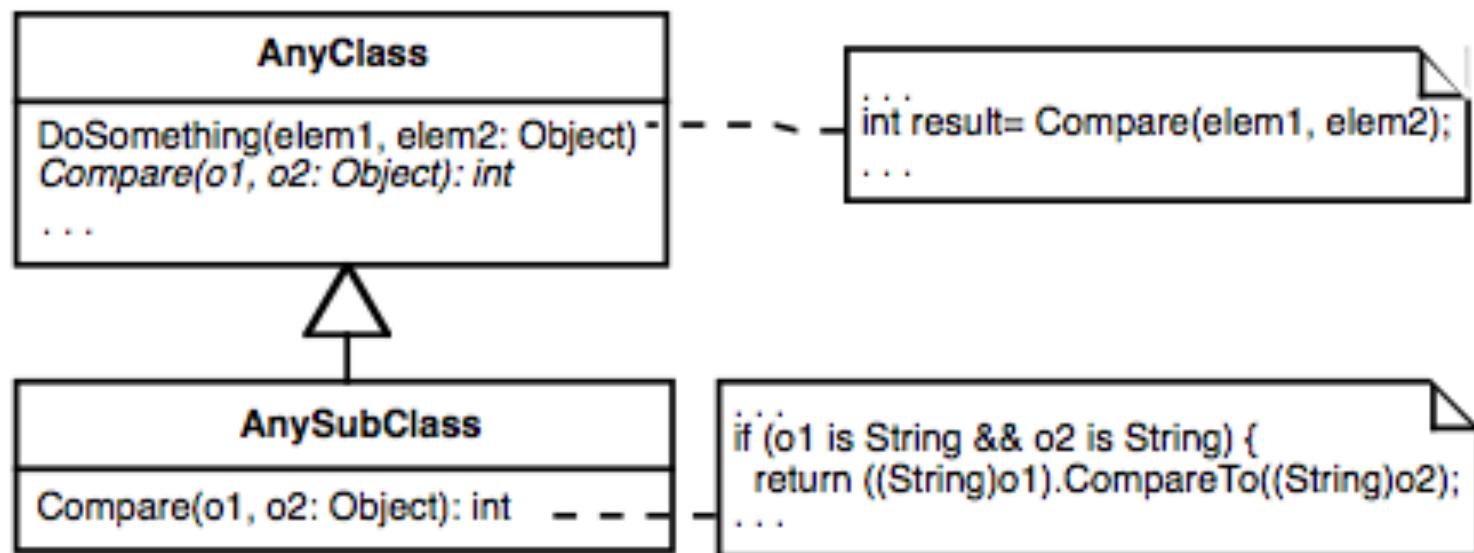
# Callback-Style of Programming (II)

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

```
int StringCompare(void* string1, void* string2) {  
    return strcmp( // C-Bibliotheksfunktion strcmp  
                  (char*)string1,  
                  (char*)string2  
    );  
} // StringCompare
```

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

# Callback-Style of Programming (III)

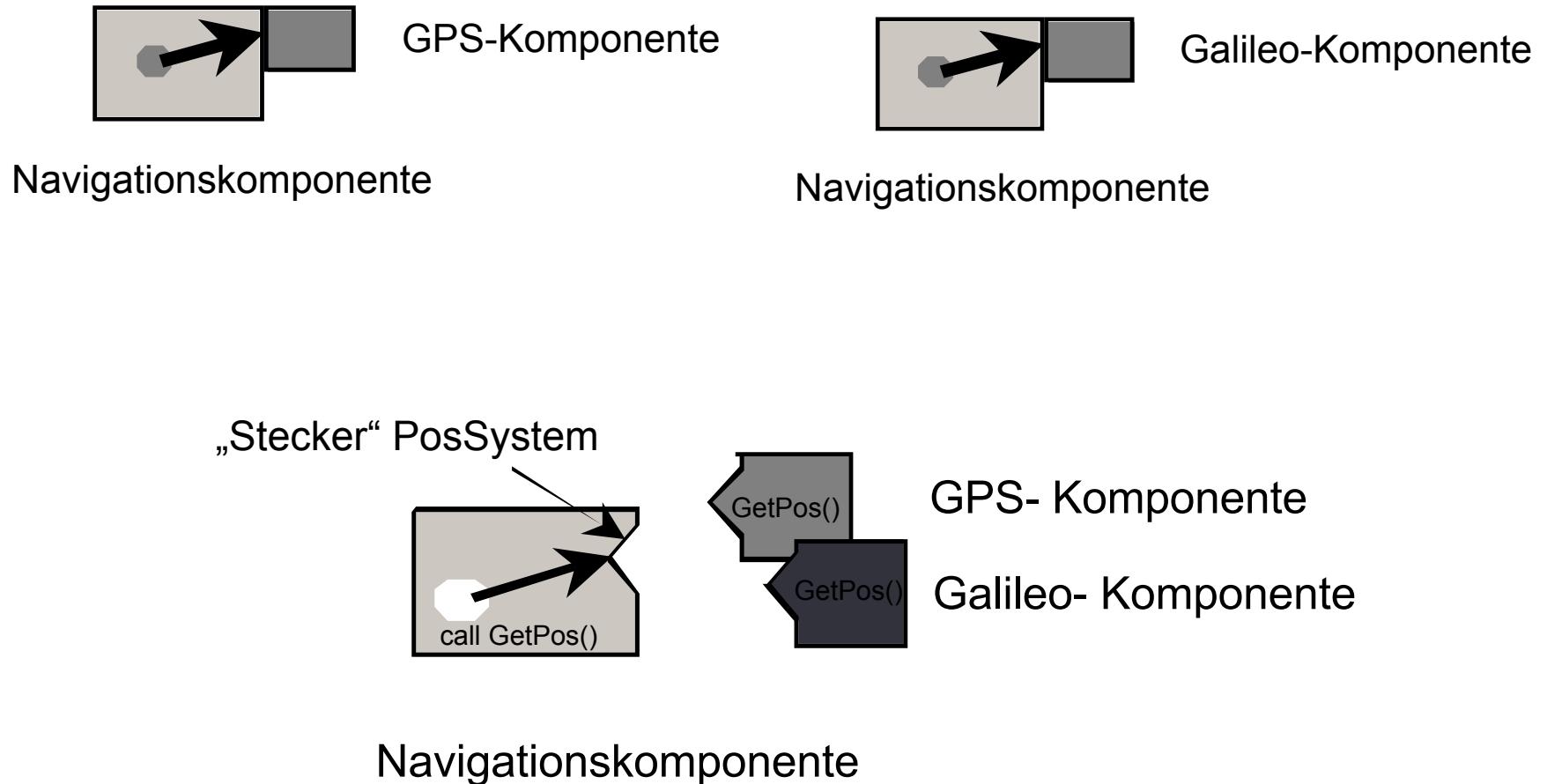


# Konzepte und Konstruktionsprinzipien für anpassbare, objektorientierte Produktfamilien

# Definition Produktfamilie = Framework = PlugIn-Architecture

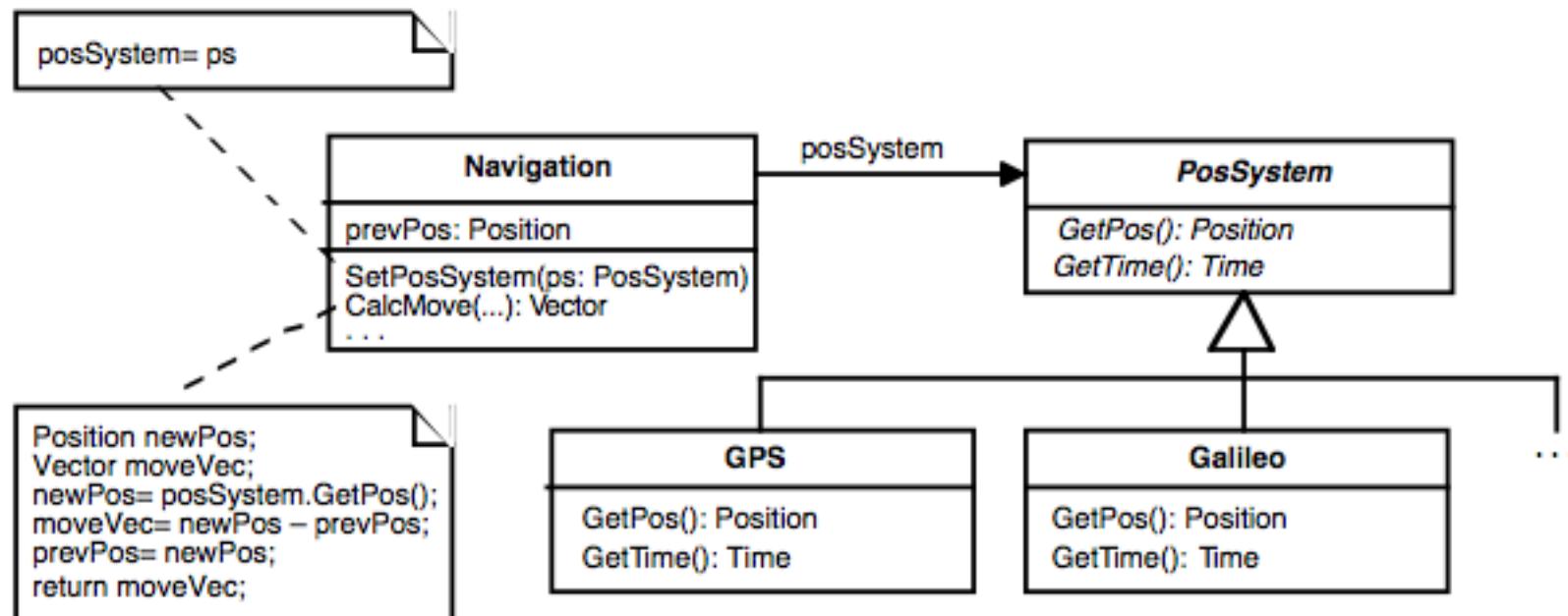
*ein Stück Software, aus dem durch den Callback-Style-of-Programming verschiedene Ausprägungen gebildet werden können, d.h. dessen Verhalten dadurch veränderbar und/oder erweiterbar ist.*

# Abstrakte Kopplung

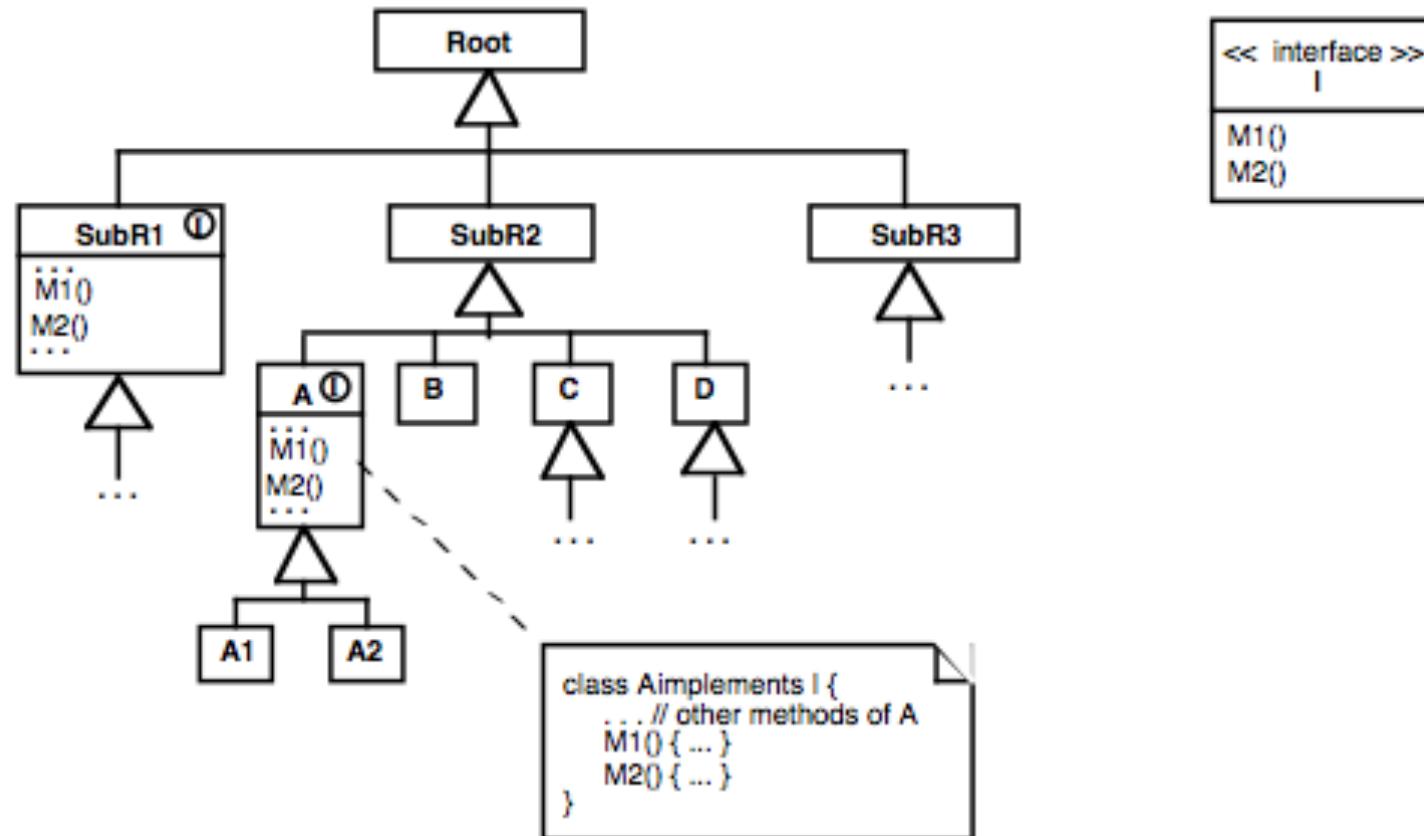


# Abstrakte Kopplung durch abstrakte Klassen

Beispiel Navigationssystem:

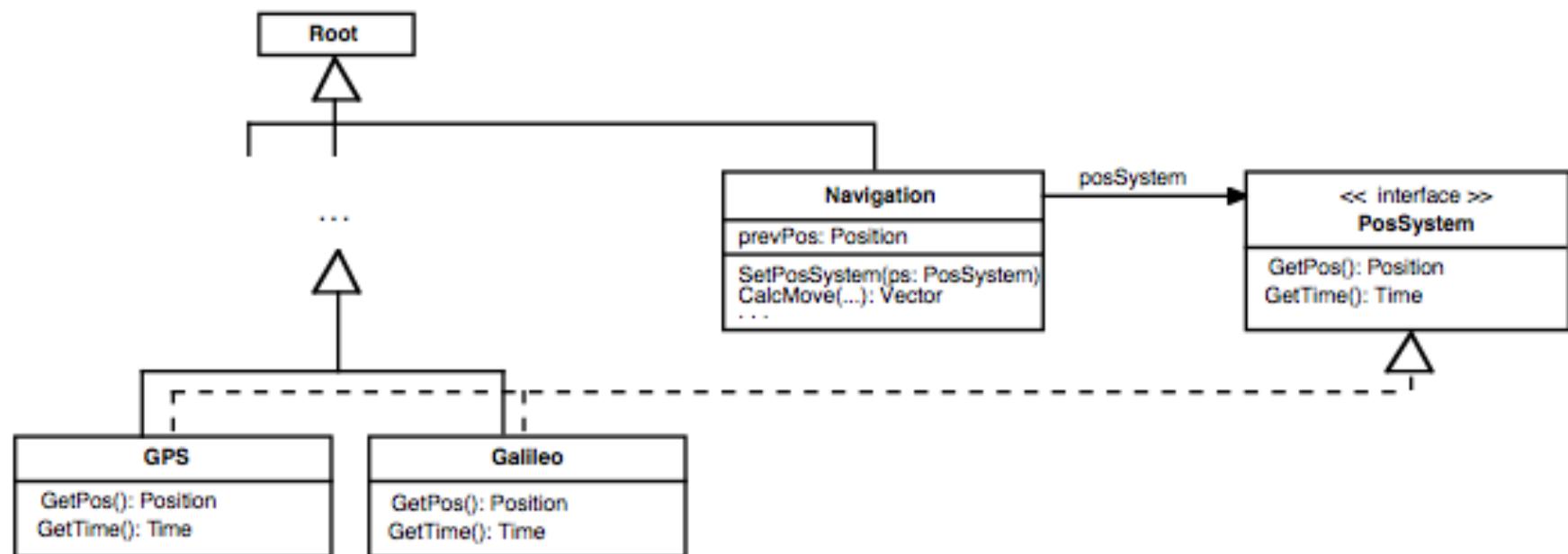


# Alternative: Schnittstelle als „Typstempel“



# Abstrakte Kopplung durch Schnittstellen

Beispiel Navigationssystem:



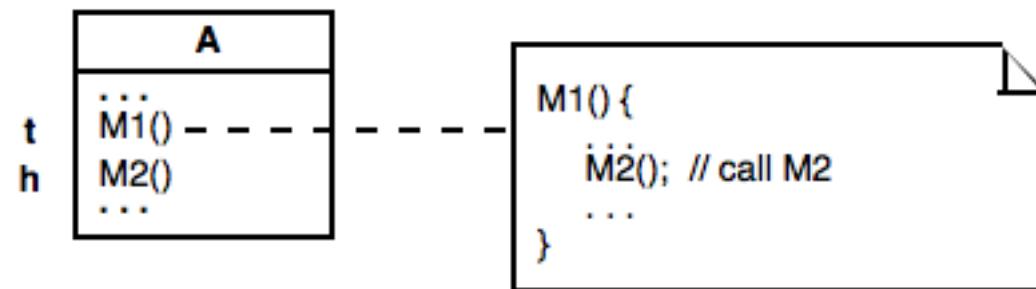
# Template- und Hook- Methoden

# Definition

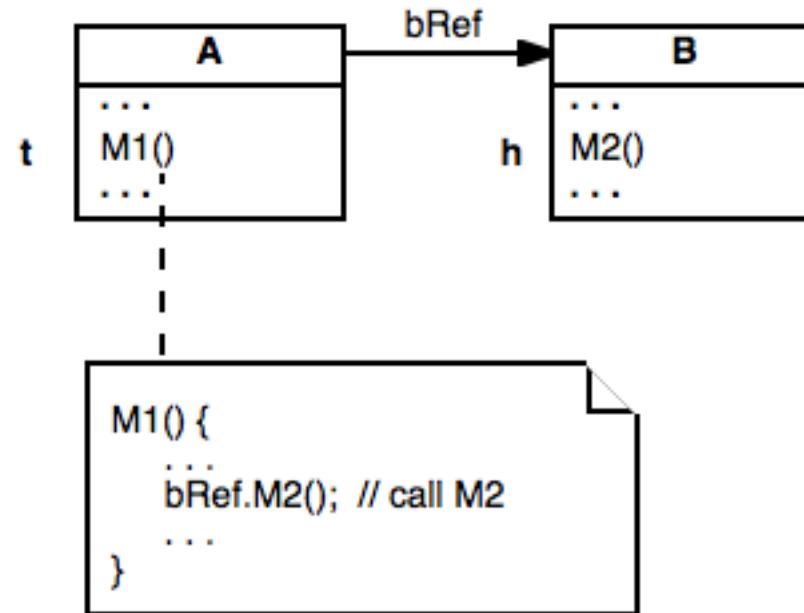
Wenn in einer Methodenimplementierung eine andere Methode aufgerufen wird, nennen wir die **aufrufende Methode die Template-Methode** und die **aufgerufene Methode die Hook-Methode**.

Man beachte, dass die hier angesprochene Template-Methode nichts mit dem C++ Sprachkonstrukt template zu tun hat.

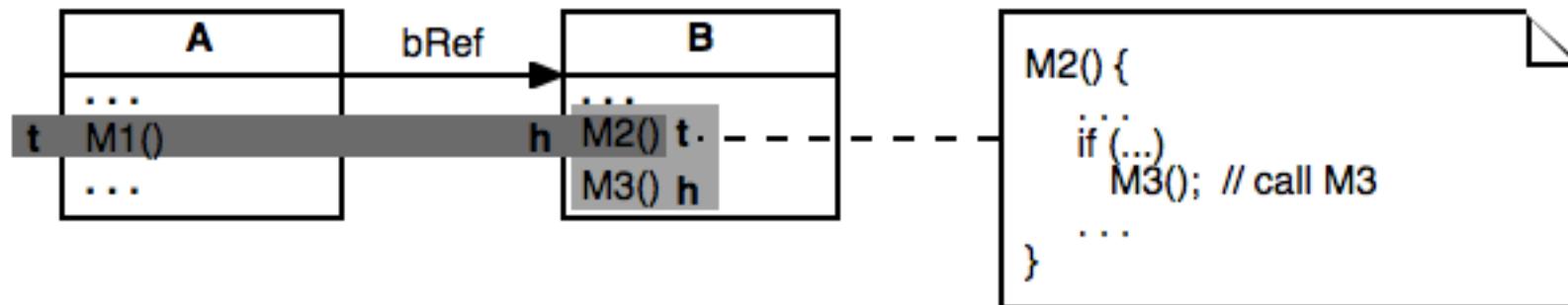
# Beide Methoden in einer Klasse



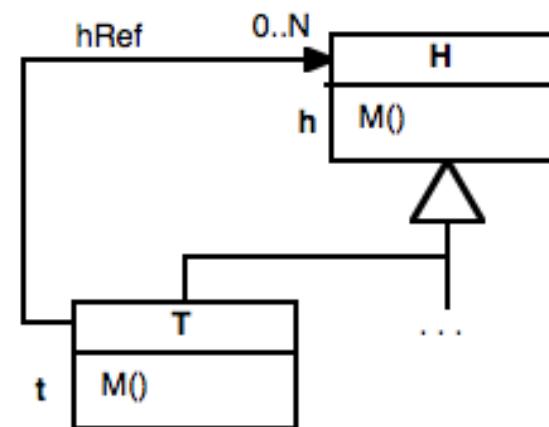
# Template- und Hook-Methoden in verschiedenen Klassen



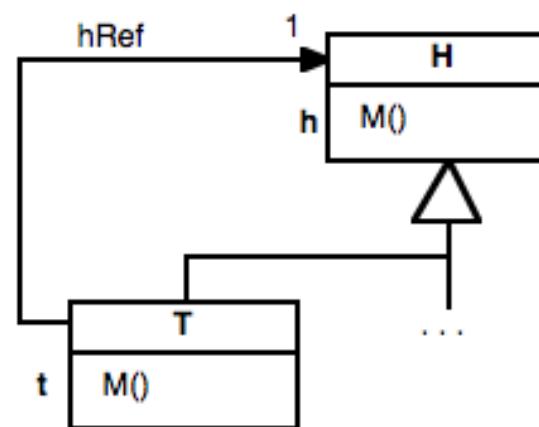
Welche Methode eine Template- und welche eine Hook-Methode ist, hängt vom Kontext ab



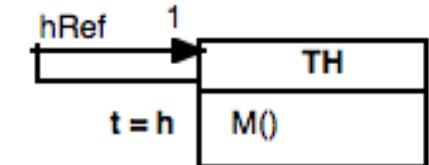
# Kombinationen mit rekursivem Charakter



Composite



Decorator

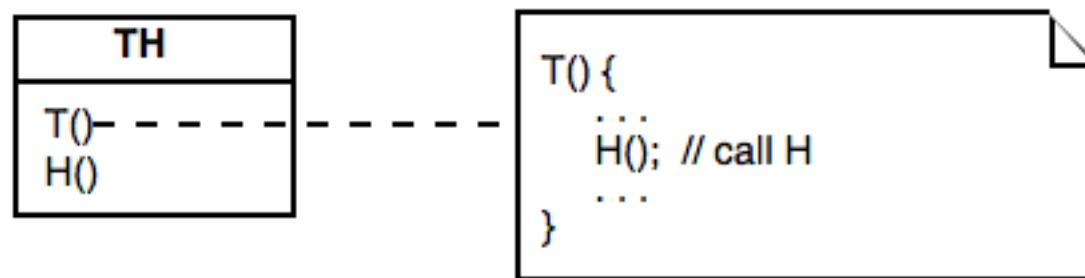


Chain-Of-  
Responsibility

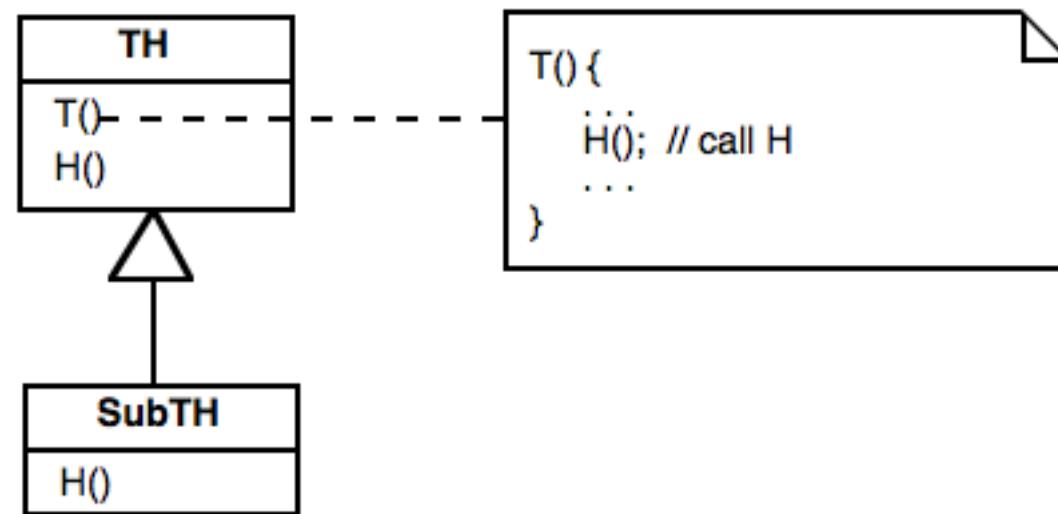
# Hook Method

# Konstruktionsprinzip

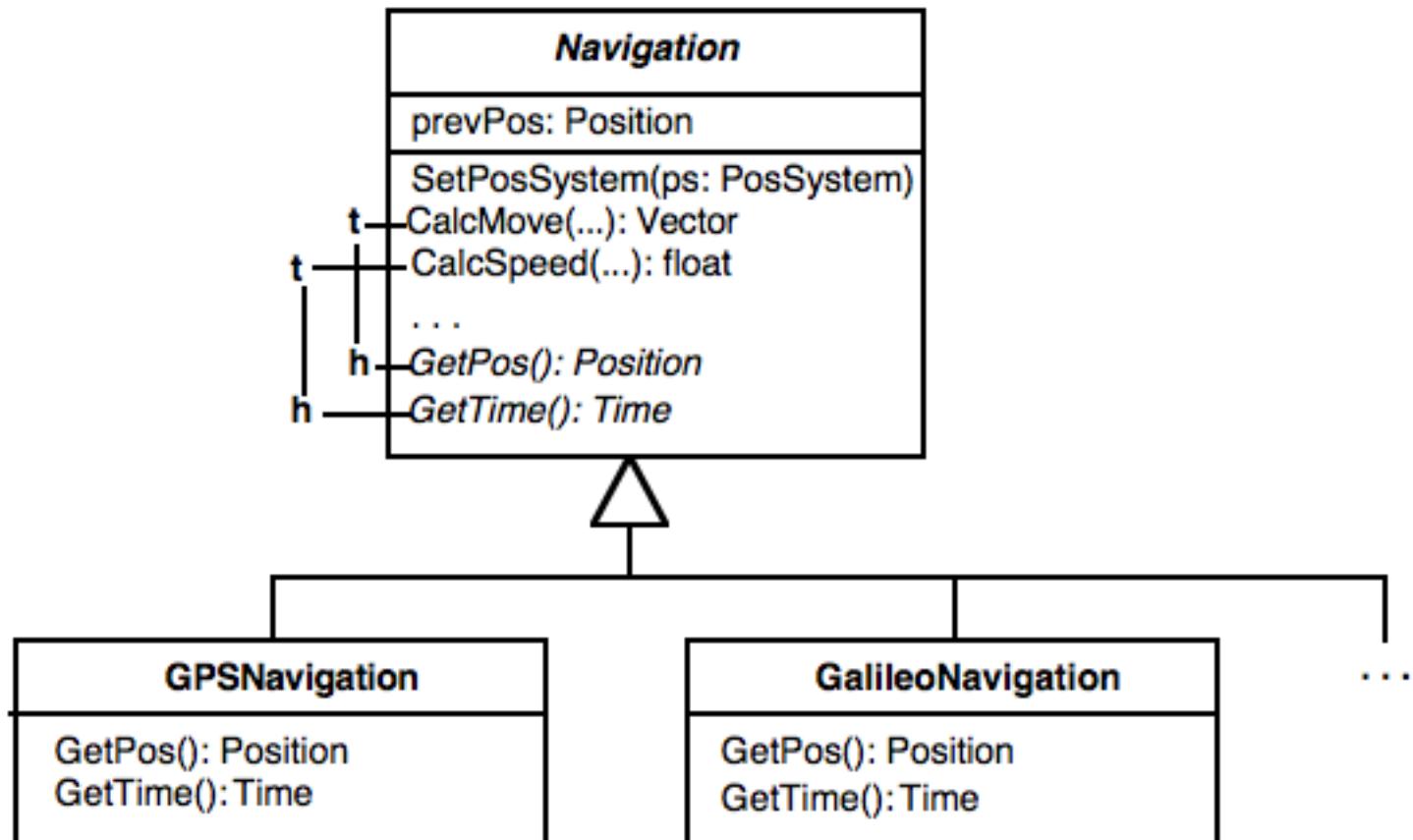
# Hook Method: Anpassung von T() durch Überschreiben von H()



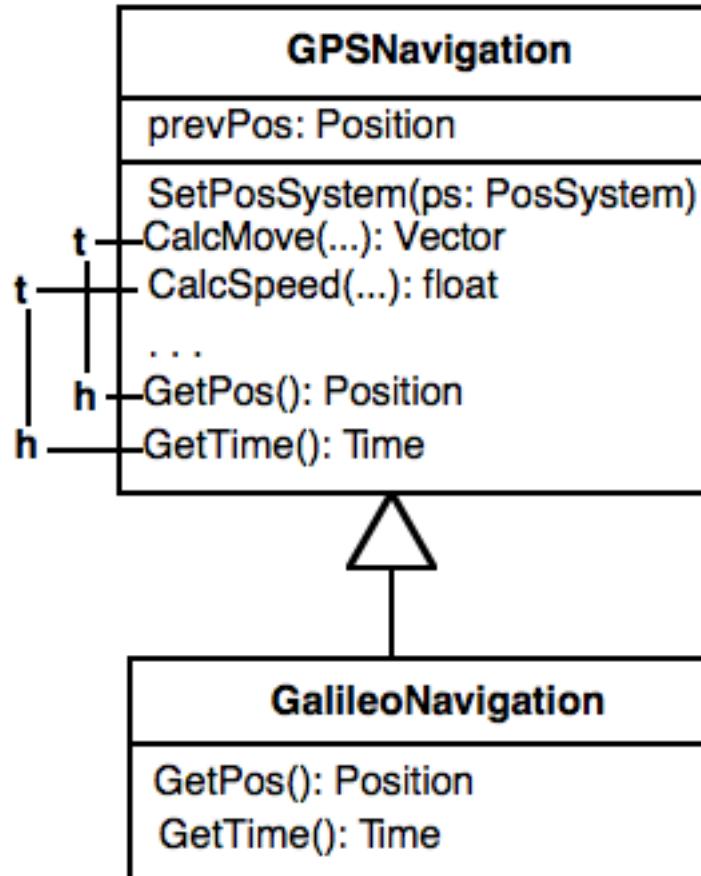
# Anpassung durch Überschreiben der Hook Method H()



# Beispielanwendung: Navigationssystem (I)



# Beispielanwendung: Navigationssystem (II)



Problem: Galileo  
ist keine Spezialisierung  
von GPS!

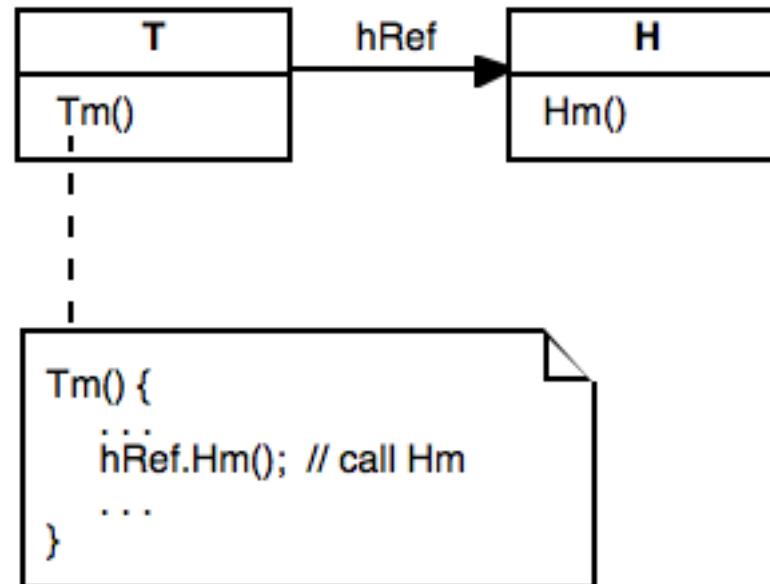
# Zusammenfassung *Hook Method*

- + Einfachheit: Es muss lediglich eine Hook-Methode für das Verhalten, das anpassbar sein soll, vorgesehen werden.
- Anpassung erfordert eine Unterklassenbildung und das Überschreiben der Hook-Methode

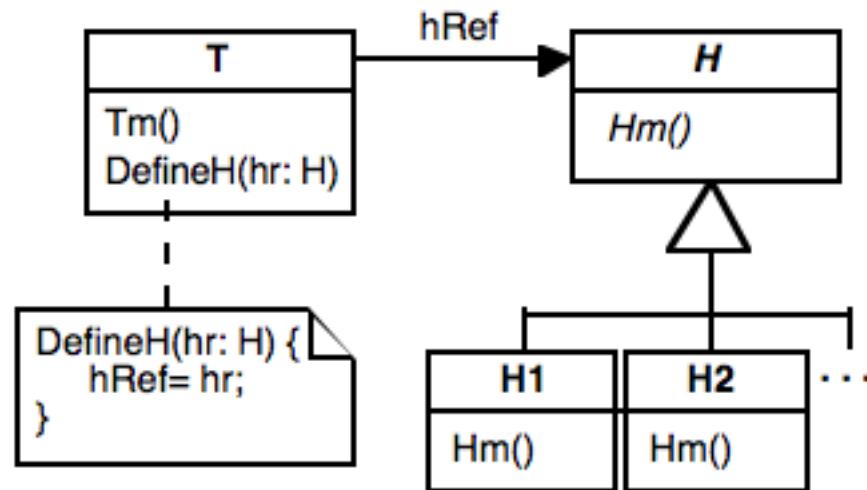
In vielen Fällen ist aber die durch das Hook-Method-Konstruktionsprinzip gebotene Flexibilität für Anpassungen ausreichend.

# Hook Object Konstruktionsprinzip

# Hook Object: Anpassung von T() durch „Einsticken“ eines H-Objektes



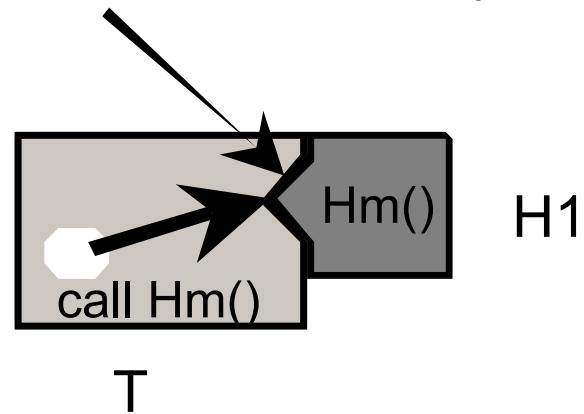
# Anpassung durch Komposition (I)



=> Anpassbarkeit zur Laufzeit

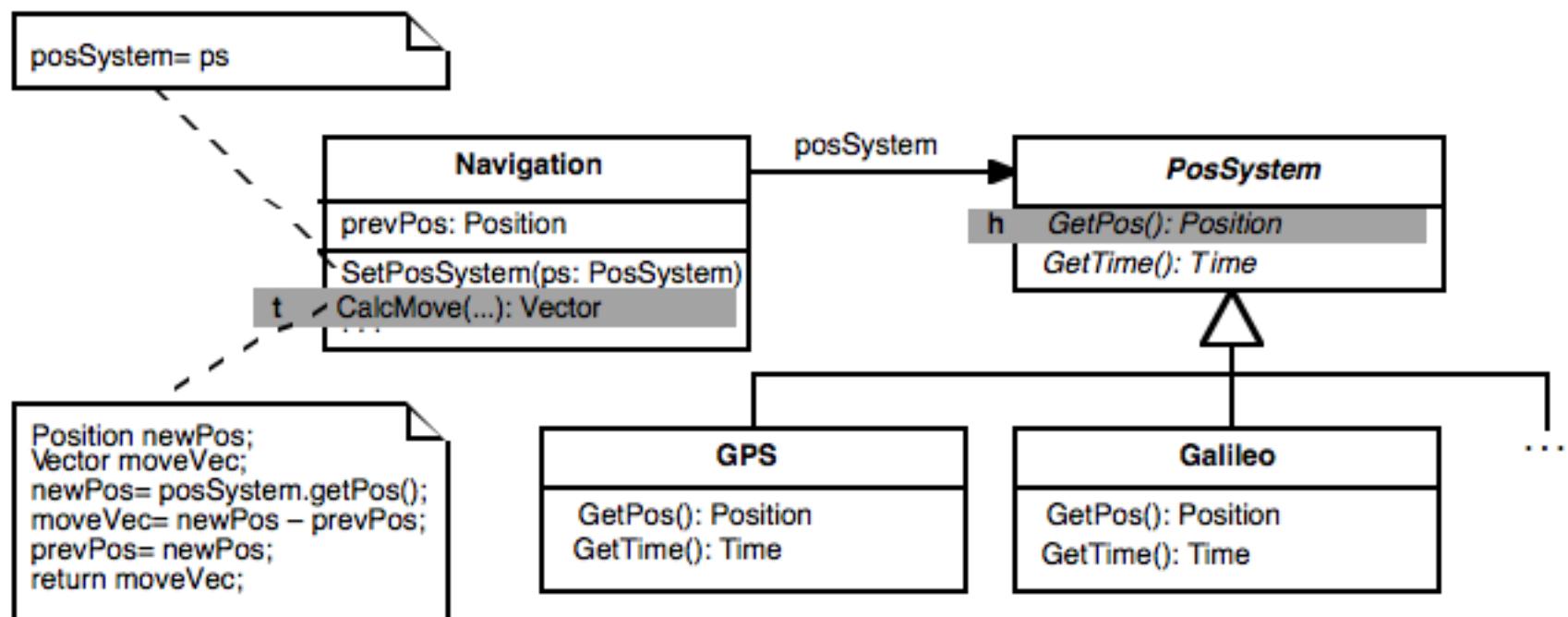
# Anpassung durch Komposition (II)

„Stecker“ vom statischen Typ H

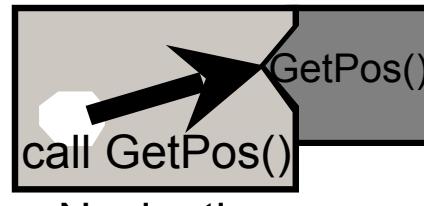


```
T sampleT= new T();  
sampleT.DefineH(new H1());
```

# Beispielanwendung: Navigationssystem (I)

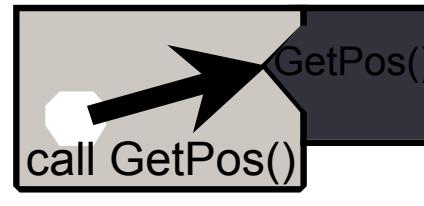


# Beispielanwendung: Navigationssystem (II)



GPS

Navigation  
(a)

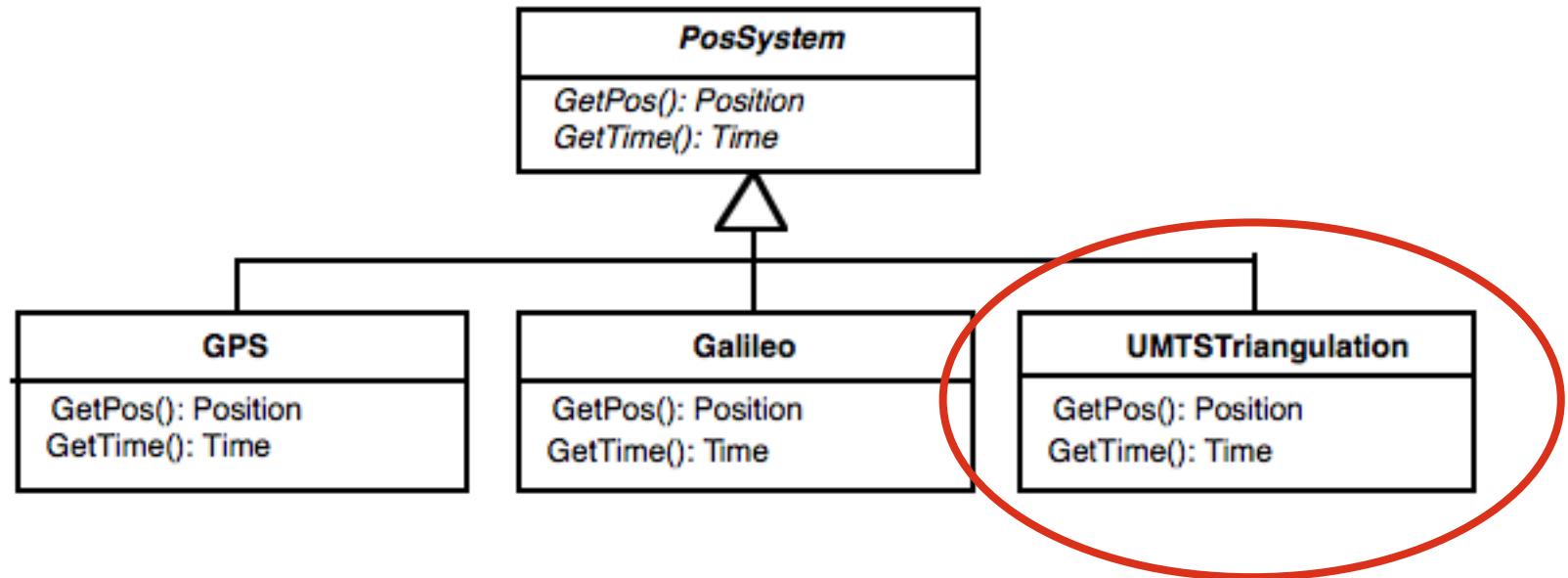


Galileo

Navigation  
(b)

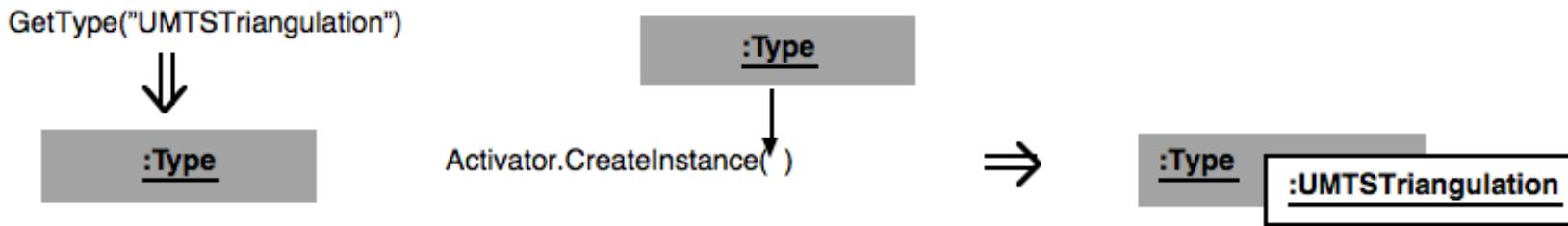
Komposition zu einem GPS-basierten (a) und  
einem Galileo-basierten (b) Navigationssystem

# Erweiterung der einsteckbaren Komponenten zur Laufzeit?



```
Navigation navigation= new Navigation(...);
String nameOfAddtlClass= "UMTSTriangulation";
Object anObj= new nameOfAddtlClass; // so nicht möglich;
// korrekte Lösung folgt
navigation.SetPosSystem((PosSystem)anObj);
```

# dazu nötige *Reflection* in .NET/C#



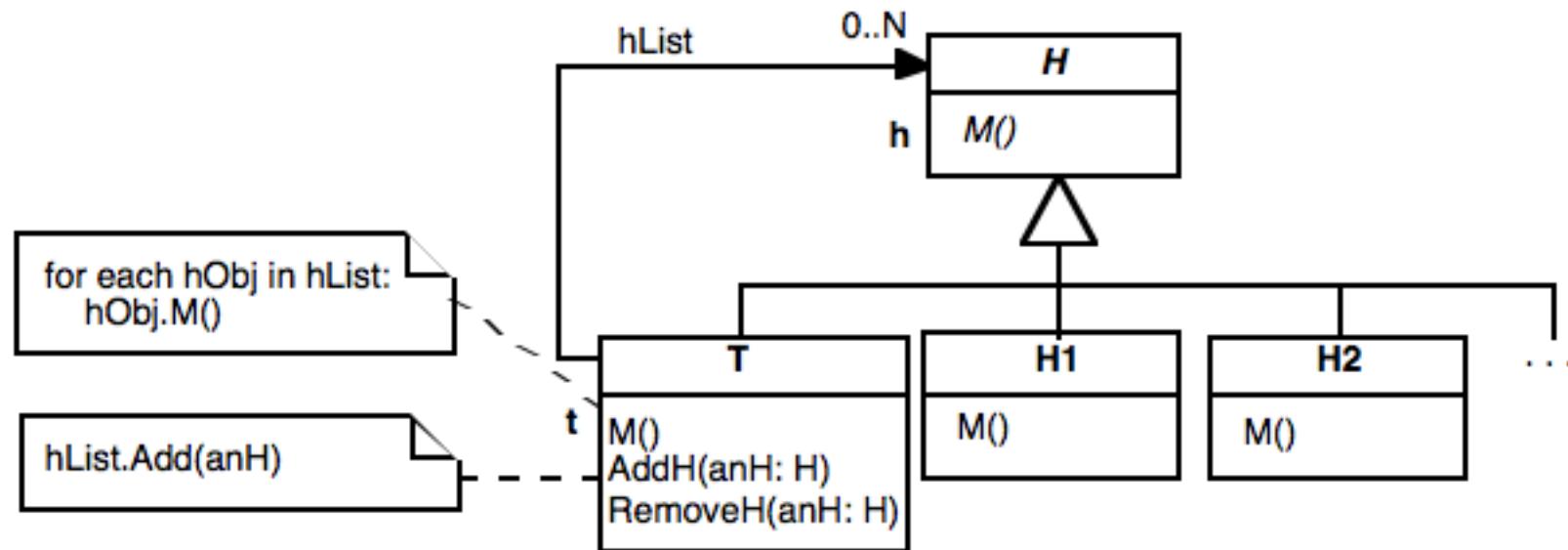
```
Navigation navigation= new Navigation(...);  
...  
String nameOfAddtlClass= "UMTSTriangulation";  
Type typeOfAddtlClass= Type.GetType(nameOfAddtlClass);  
Object anObj;  
PosSystem posSys;  
  
if (typeOfAddtlClass != null) {  
    anObj= Activator.CreateInstance(typeOfAddtlClass);  
    if (anObj != null && anObj is PosSystem)  
        posSystem= (PosSystem) anObj;  
    else ... // error handling  
}  
navigation.SetPosSystem(posSys);
```

# Zusammenfassung *Hook Object*

- + einfache Konfiguration, auch zur Laufzeit
- höhere Komplexität des Entwurfs und der Implementierung als beim Hook-Method-Konstruktionsprinzip

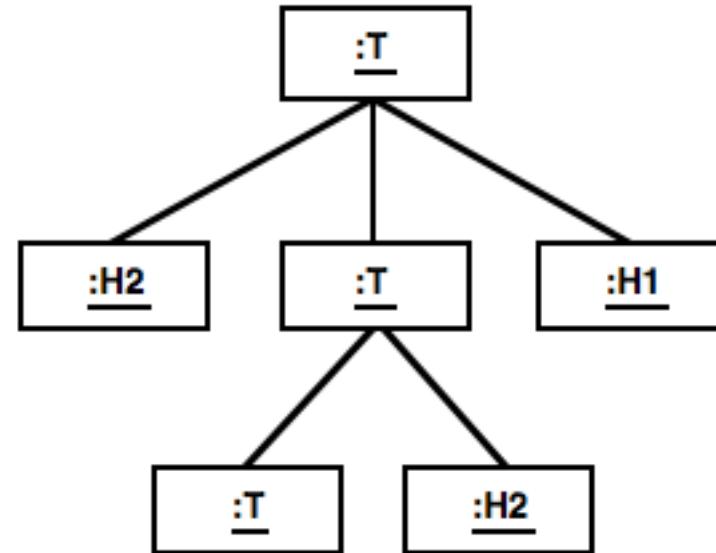
# Composite Konstruktionsprinzip

Composite: ein Baum von Objekten kann wie ein einzelnes Objekt verwendet werden



- Namen der Template- und Hook-Methoden sind gleich
  - Objektverwaltung mit AddH() und RemoveH()

# Beispiel: Definition einer Objekthierarchie



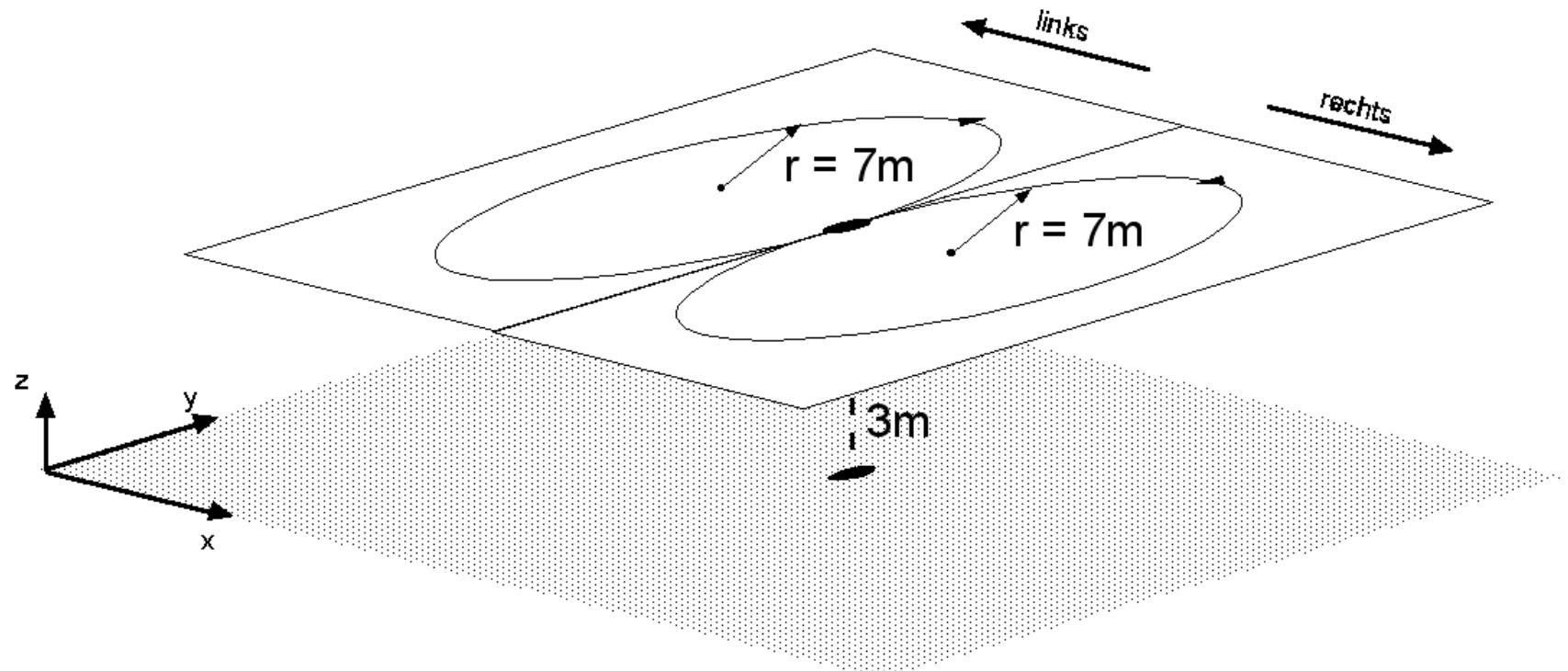
```
T root= new T();
T subRoot= null;
root.AddH(new H2());
subRoot= new T();
root.AddH(subRoot);
root.AddH(new H1());
subRoot.AddH(new T());
subRoot.AddH(new H2());
```

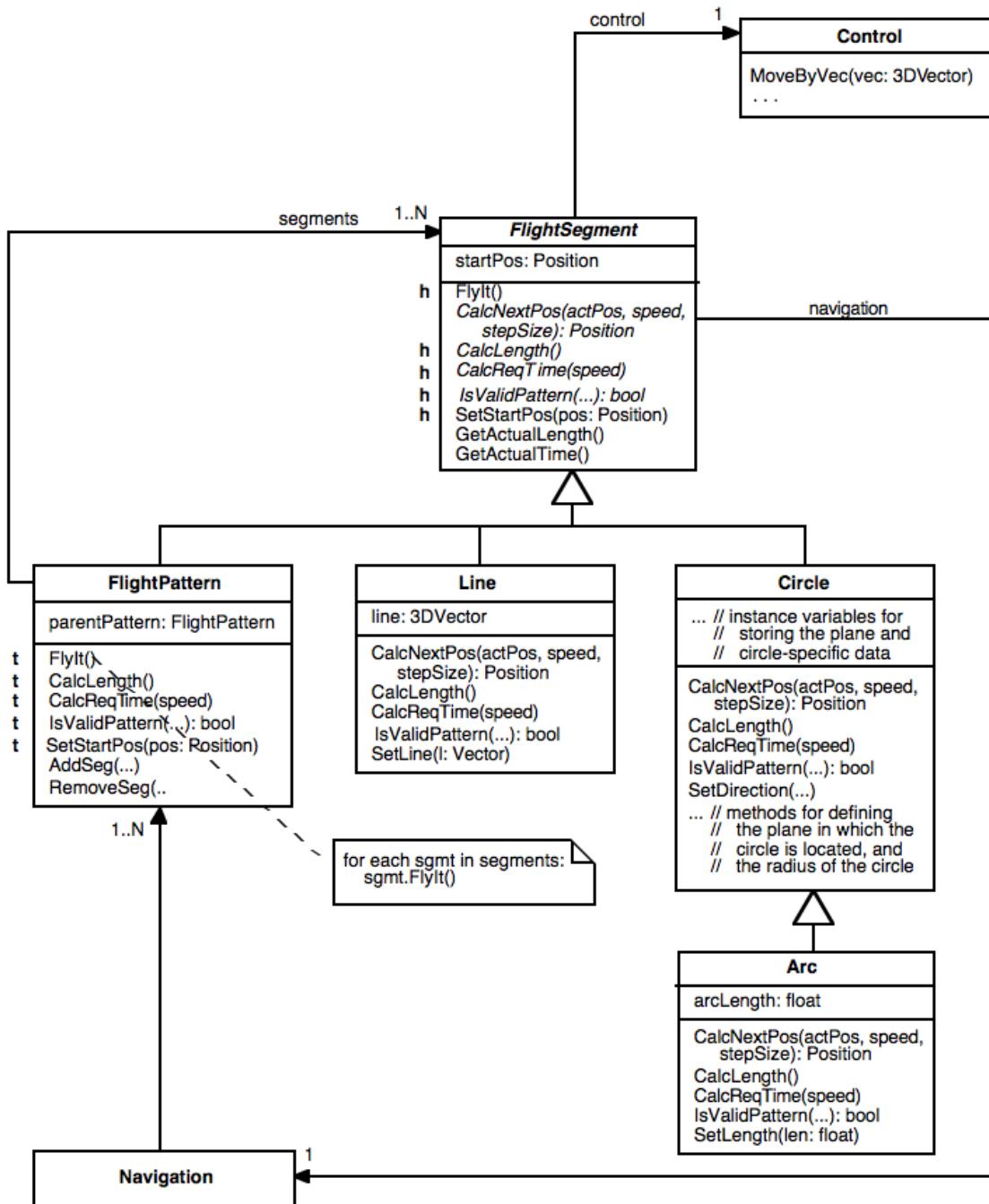
Durch die Struktur der Template-Methode kann die Objekthierarchie wie ein Objekt verwendet werden

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

M() ist keine rekursive Methode; sie operiert jedoch auf einer rekursiven Datenstruktur (Baum).

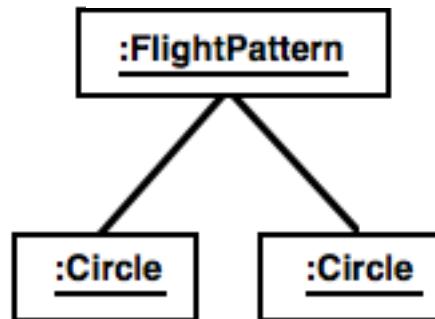
# Beispiel: Komposition eines 8er-Flugmusters aus Segmenten



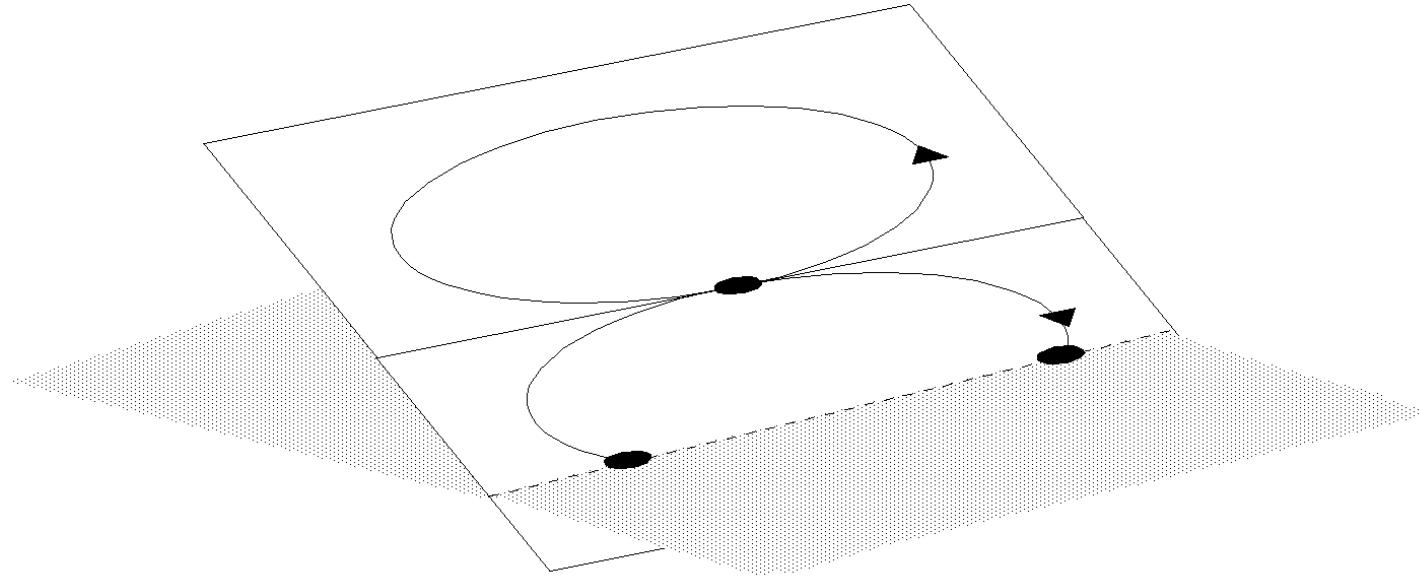


## 8er Schleife

```
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.
```



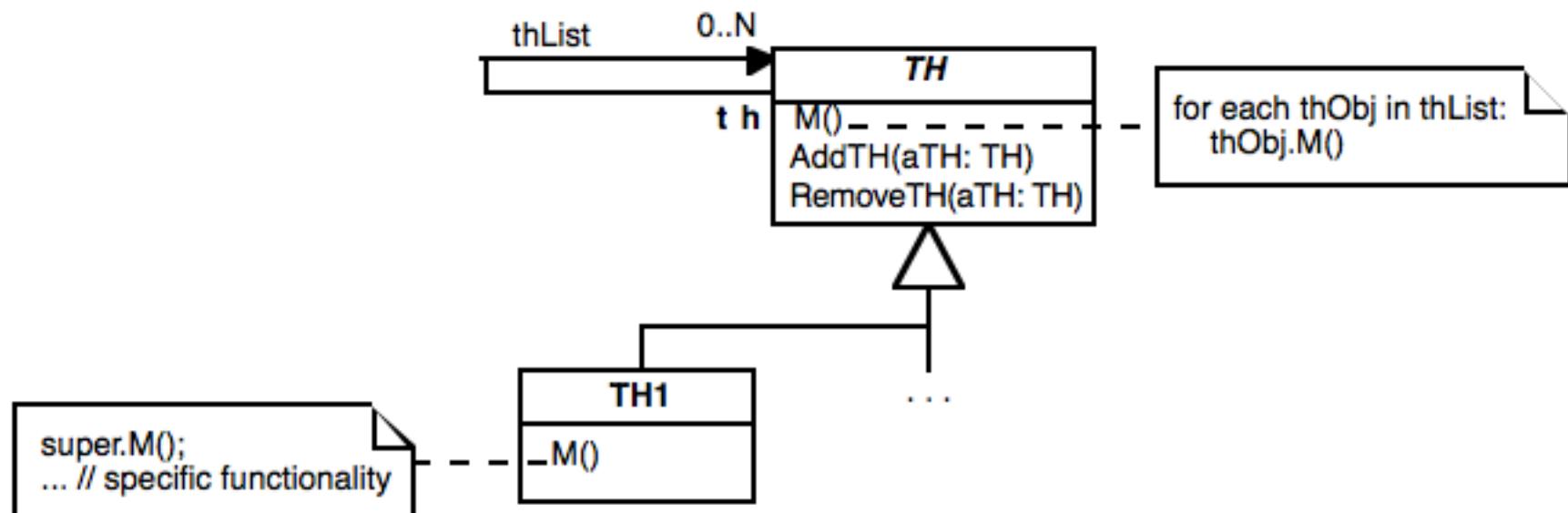
# IsValidPattern(): prüft, ob ein Flugmuster zu einer Bodenberührung führt



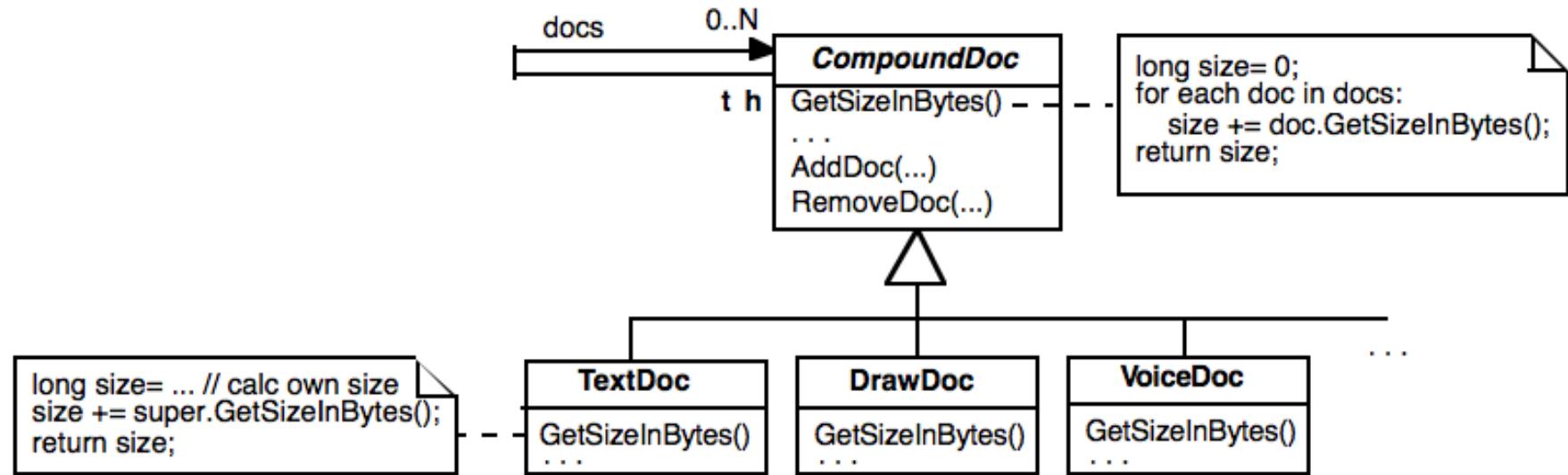
- IsValidPattern() ist in FlightPattern gemäß der Composite-Template-Methode implementiert
- analog: FlyIt(), CalcLength(), CalcReqTime(), CalcLength(), CalcReqTime()
- FlyIt() ist bereits in FlightSegment implementiert  
-> CalcNextPos()

# Composite-Variante: Verwaltung und Funktionalität in einer Klasse

- T- und H- Klasse verschmolzen
- Semantik der Komposition ändert sich
- die grundlegende Eigenschaft, eine Objekthierarchie definieren zu können, bleibt erhalten



# Beispiel: “zusammengesetzte” Dokumente



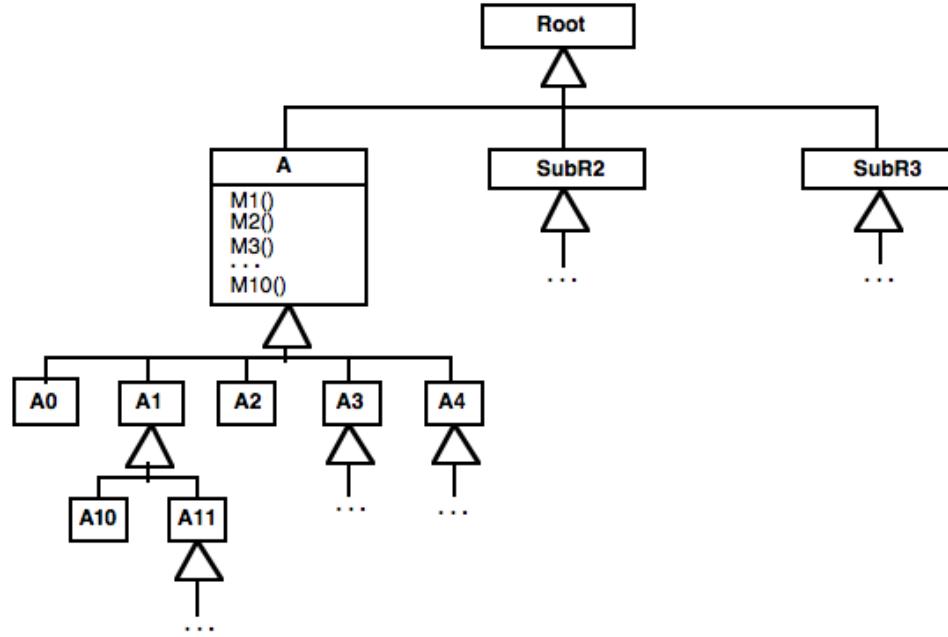
Ein Textdokument, das verschiedene andere Dokumente wie zum Beispiel Zeichnungen, Ton- oder Videosequenzen enthält, sei für die Verwaltung der enthaltenen Dokumente zuständig und bietet zum Beispiel zusätzlich Funktionalität zum Editieren dieser eingebetteten Dokumente an.

# Zusammenfassung *Composite*

- + einfach erstellbare und verwendbare Objekthierarchien können gebildet werden
- + neue Elemente (Unterklassen der Hook-Klasse) ohne Änderung der Template-Klasse
- Komplexität der Interaktionen zwischen den in der Hierarchie angeordneten Objekten, um die automatische Iteration über die Baumhierarchie durchzuführen.  
Objekthierarchien kommen sehr häufig und in vielen Anwendungsbereichen vor: in Fenstern gruppierte GUI-Elemente, Stücklisten, Workflows

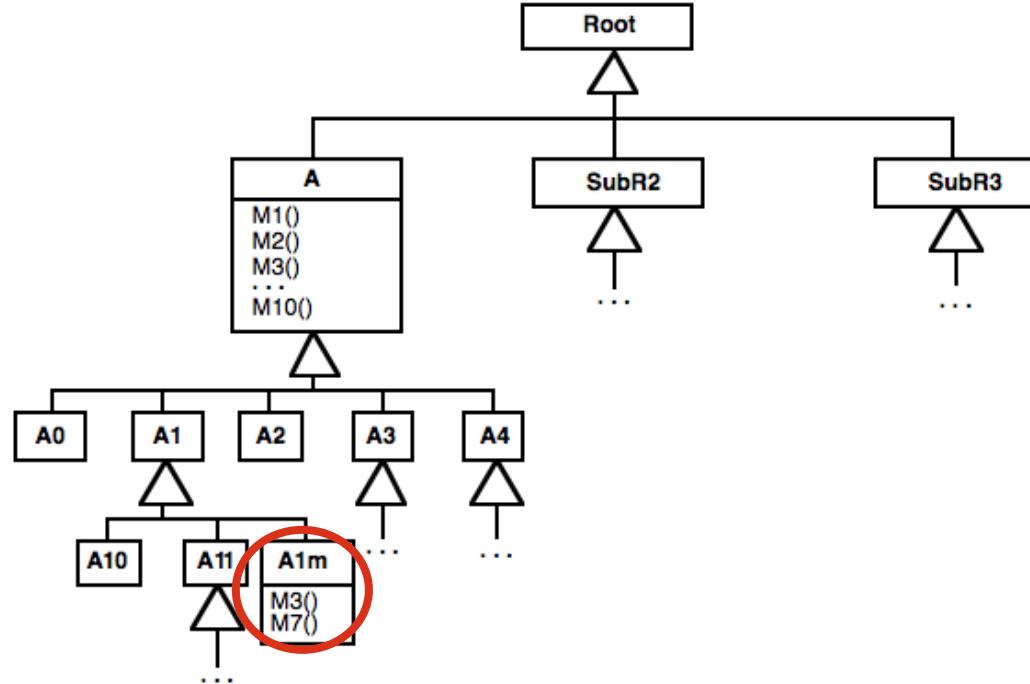
# Decorator Konstruktionsprinzip

# Motivation: Änderungen einer Klasse mit vielen Unterklassen (I)



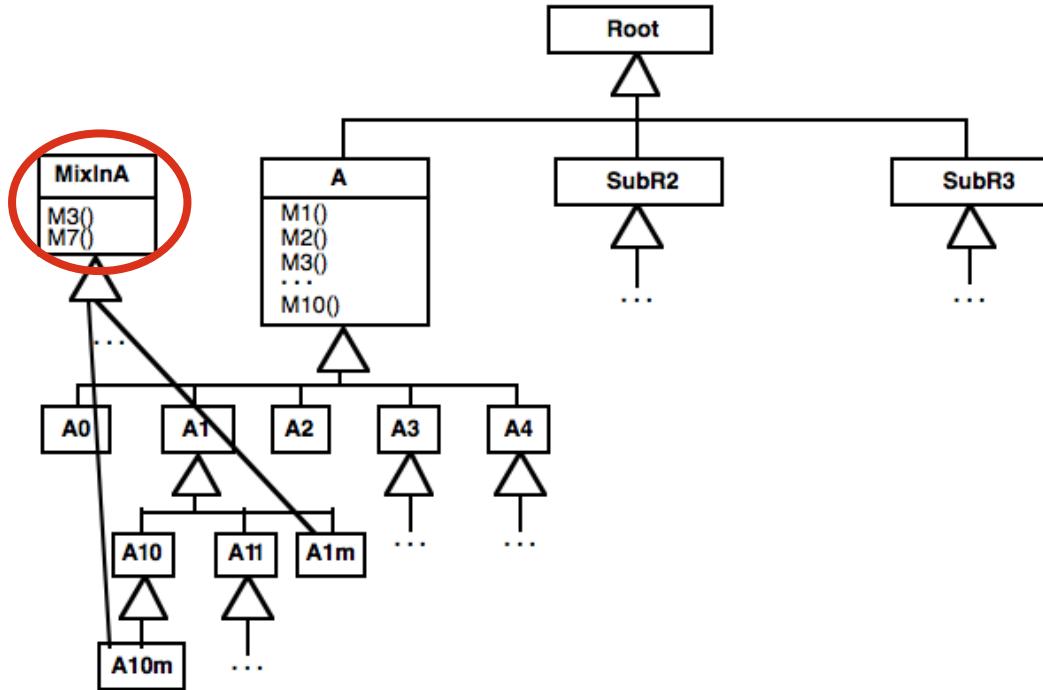
- Änderungen von M3() und M7() in Klasse A erforderlich
- Quelltext von A, wenn vorhanden, ändern?
- Änderung durch Vererbung?

# Motivation: Änderungen einer Klasse mit vielen Unterklassen (II)



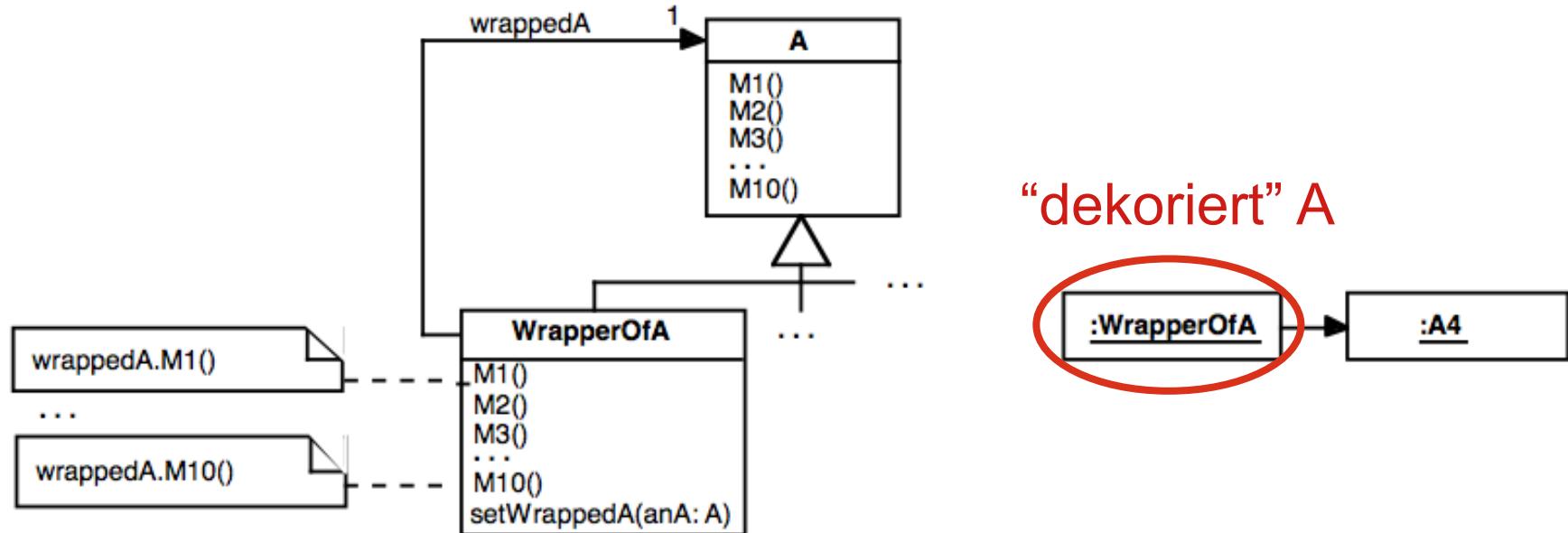
- für eine Klasse (zB **A1m**) ist die Anpassung sinnvoll
- für alle Unterklassen von **A** ist das aufwändig

# Motivation: Änderungen einer Klasse mit vielen Unterklassen (III)



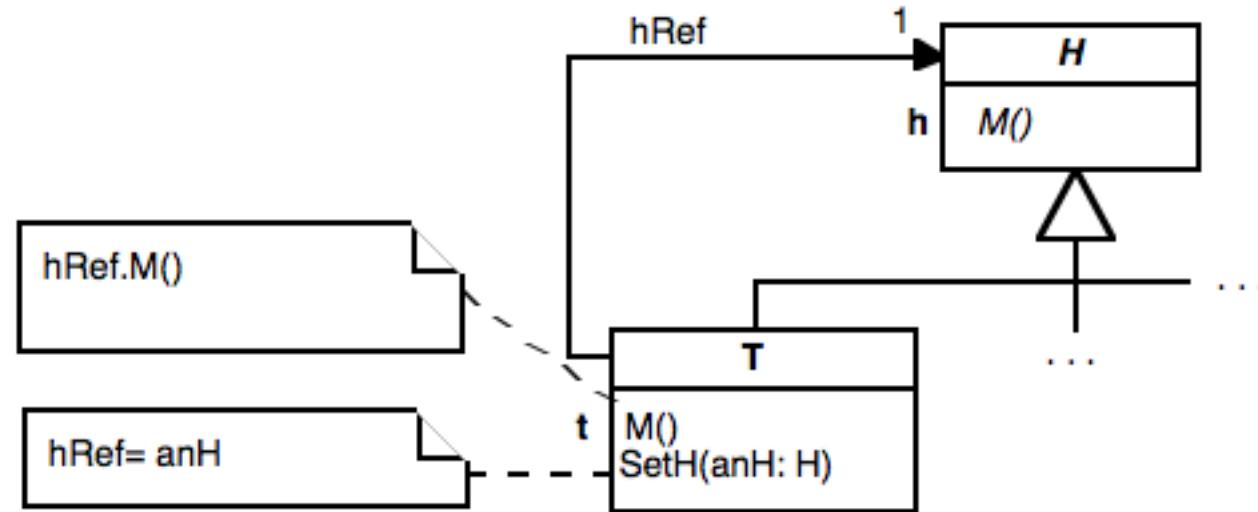
- In Programmiersprachen, die mehrfache Vererbung unterstützen, können sogenannte Mix-In-Klassen definiert werden
- dennoch muss für jede Klasse, deren Verhalten angepasst werden soll, eine Unterklasse gebildet werden

# Änderung einer Klasse durch Komposition statt durch Vererbung



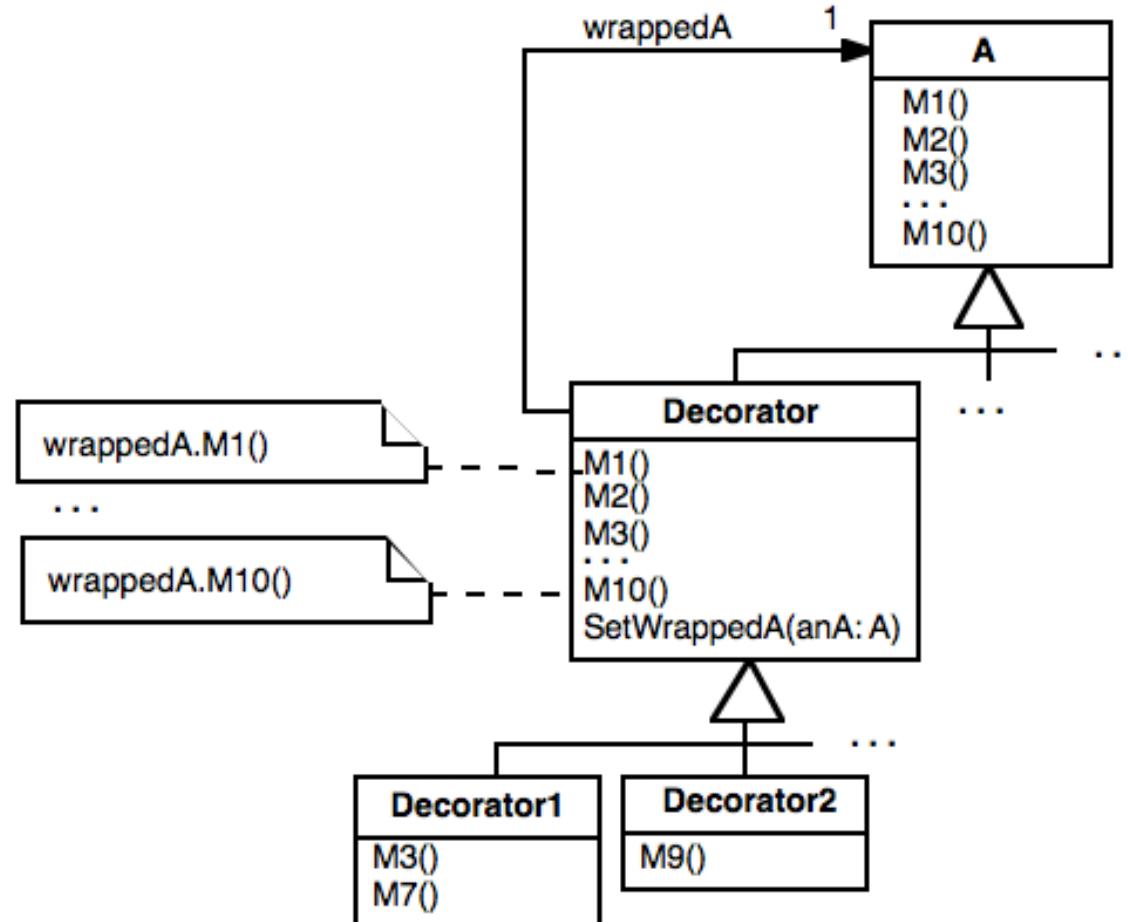
- In der Klasse **WrapperOfA** werden alle Methoden von **A** überschrieben, indem der Methodenaufruf jeweils an ein über die Instanzvariable **wrappedA** referenziertes Objekt delegiert wird – mit Ausnahme jener, die geändert werden.
- Da **WrapperOfA** eine Unterkelas von **A** ist, kann überall dort, 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 Unterkelas von **A** verweisen.

# Decorator: Anpassung durch Komposition mit beliebig vielen Objekten statt durch Vererbung

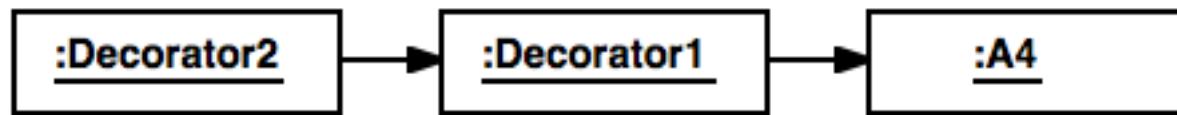
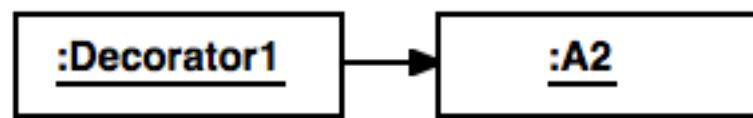


- Namen der Template- und Hook-Methoden sind gleich
- Hinzufügen von Decorator-Objekten mit SetH()
- Eine Instanz des Decorators (Filters) T zusammen mit der Instanz einer Unterklasse von H kann von Klienten wie ein H-Objekt verwendet werden. Allerdings wird das Verhalten des H-Objektes dadurch modifiziert.
- H + Decorator(s) sind als “ein Objekt” verwendbar (vgl. Composite)

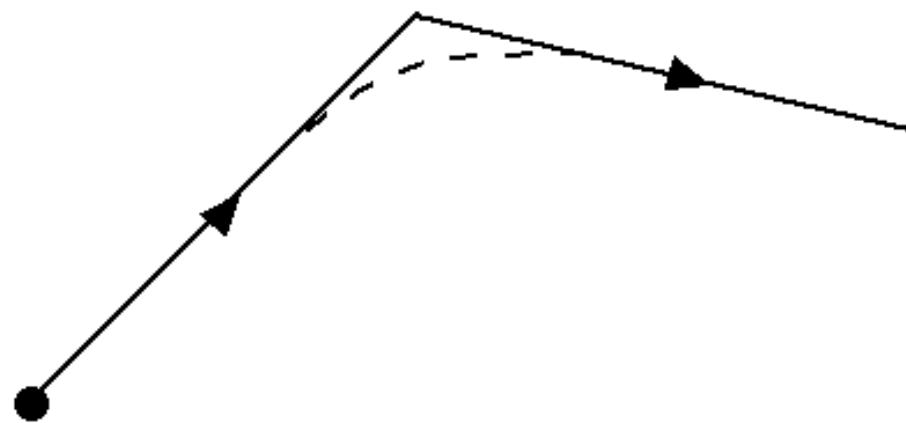
# Vorschlag für einen Entwurf, wenn mehrere Decorator-Klassen verwendet werden

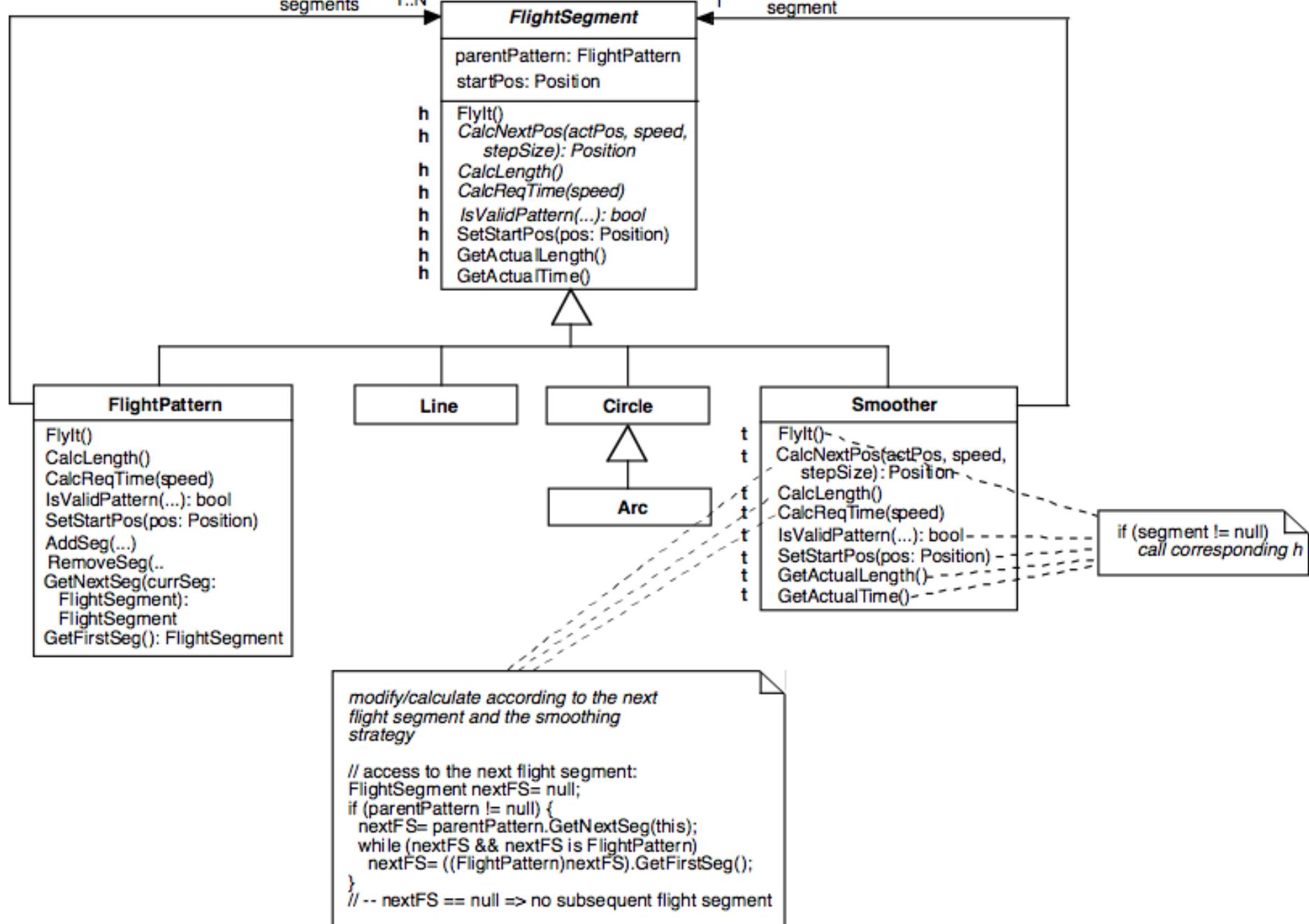


# zwei exemplarische Kompositionen



# Beispiel: Abrunden von Flugmustern





## Verwendung der Decorator-Klasse Smoother

```
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 (I)

- 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. (vgl. Decorator Smoother)

## Rahmenbedingungen für die Anwendung des Decorator-Konstruktionsprinzips (II)

In unserem Beispiel müssen daher 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));
```

# Anwendung des Decorator-Konstruktionsprinzips zum Entwurf "leichtgewichtiger" Wurzelklassen

- 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 daher sowohl beim (Erst-)Entwurf einer Klassenhierarchie als auch bei der Erweiterung von Klassenhierarchien nutzbringend eingesetzt werden.

## Beispiel: Clipping-Mechanismus bei GUI-Bibliotheken

- Clipping-Mechanismus: das Zuschneiden eines GUI-Elements auf dessen festgelegte Größe
- Da der Clipping-Mechanismus 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. (vgl. Decorator-Beispiel in Gamma et al., 1995)

# Zusammenfassung *Decorator*

- + einfache Anpassung durch Objektkomposition
- + neue Decorator-Elemente (Template-Klassen, die Unterklassen der Hook-Klasse sind) können definiert werden, ohne die Unterklassen der Hook-Klasse verändern zu müssen;
- + „leichtgewichtige“ Klassen können damit elegant realisiert werden
- allerdings sollte die Hook-Klasse die erwähnte Rahmenbedingung erfüllen (faktorisiert Verhalten aus allen Unterklassen heraus)
- zusätzliche Indirektion bei Methodenaufrufen
- komplexe Interaktionen zwischen beteiligten Objekten

# Zusammenfassung der Merkmale der OO Konstruktionsprinzipien

# Charakteristika von Template- und Hook-Methoden

		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.		T erbt von H	T = H	

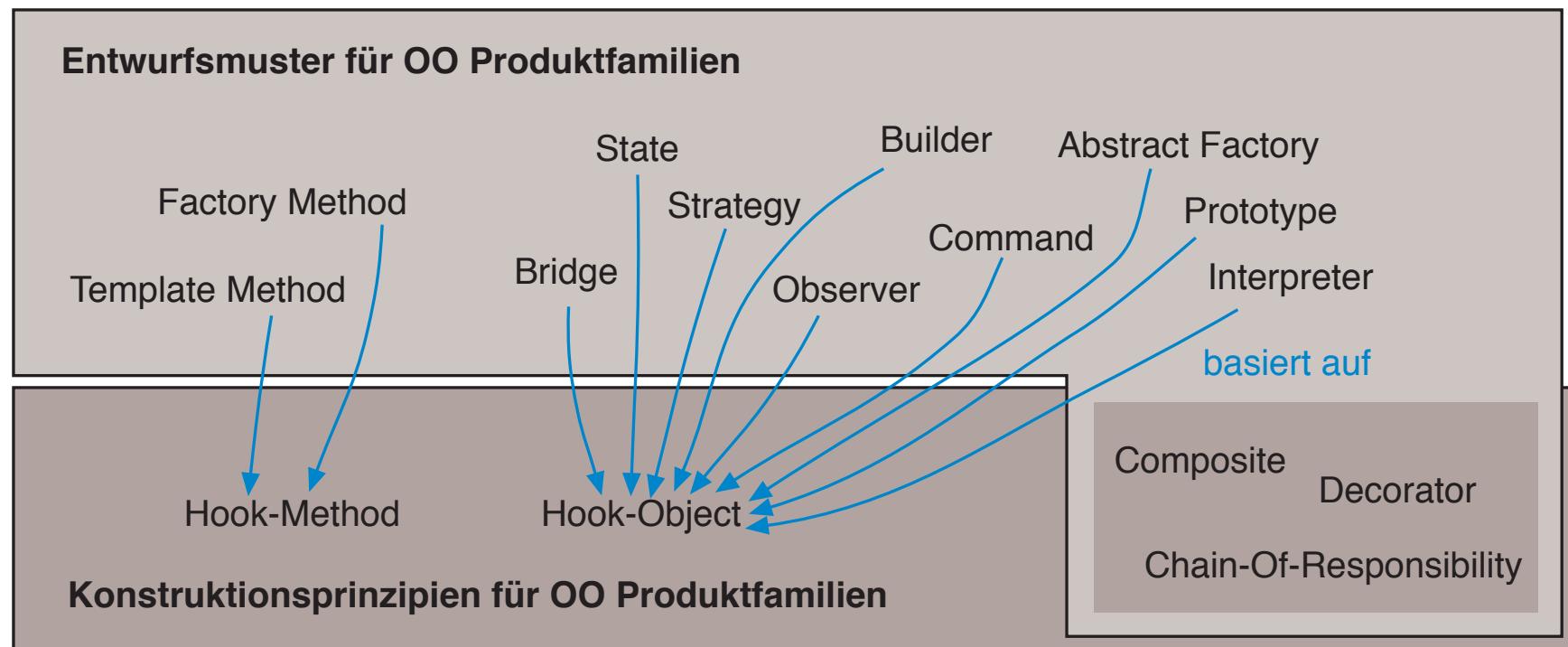
# Anpassung

Konstruktionsprinzipien					
	Hook-Method	Hook-Object	Composite	Decorator	
Anzahl der beteiligten Objekte	1	$1(T) + 1(H)$ oder $1(T) + N(H)$	N Objekte, die wie ein Objekt verwendet werden		
Anpassung	durch Vererbung und Instanziierung der entsprechenden Klasse	durch Komposition (bei Bedarf zur Laufzeit)			

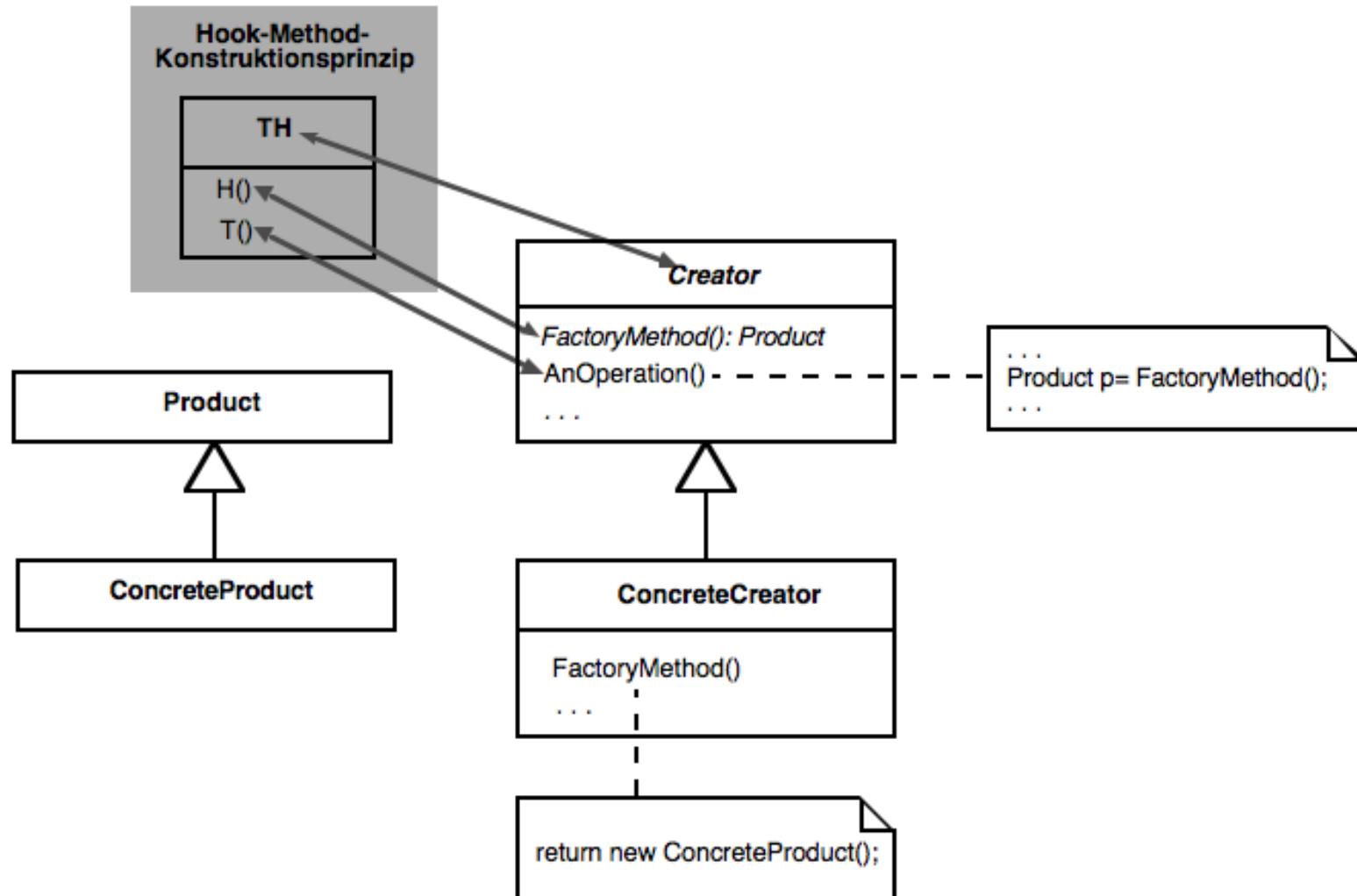
# Konstruktionsprinzipien und Entwurfsmuster (Design Patterns\*)

\* E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Addison-Wesley, 1995

# 14 der 23 Design Patterns von Gamma et al. beziehen sich auf OO Produktfamilien



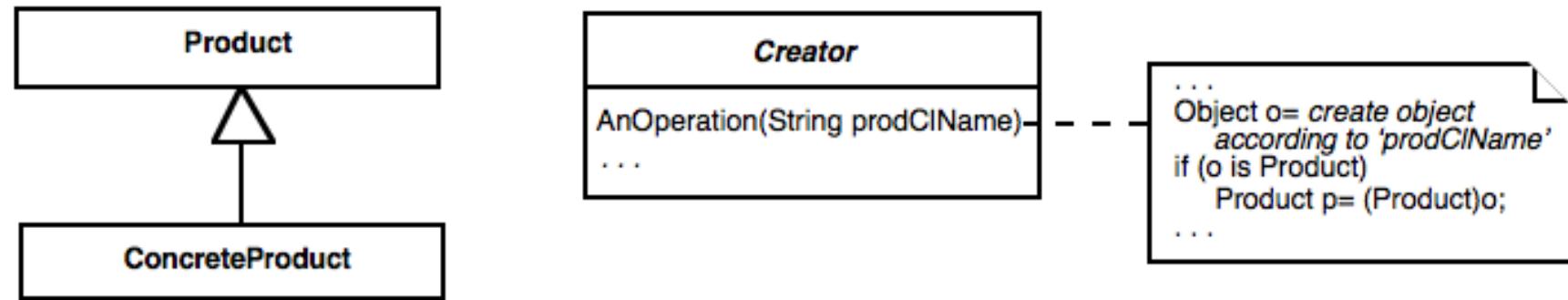
# Template- und Hook-Methoden im Entwurfsmuster *Factory Method*



# Semantik der Hook-Methode/-Klasse ist die Basis für die Namensgebung bei Design Patterns

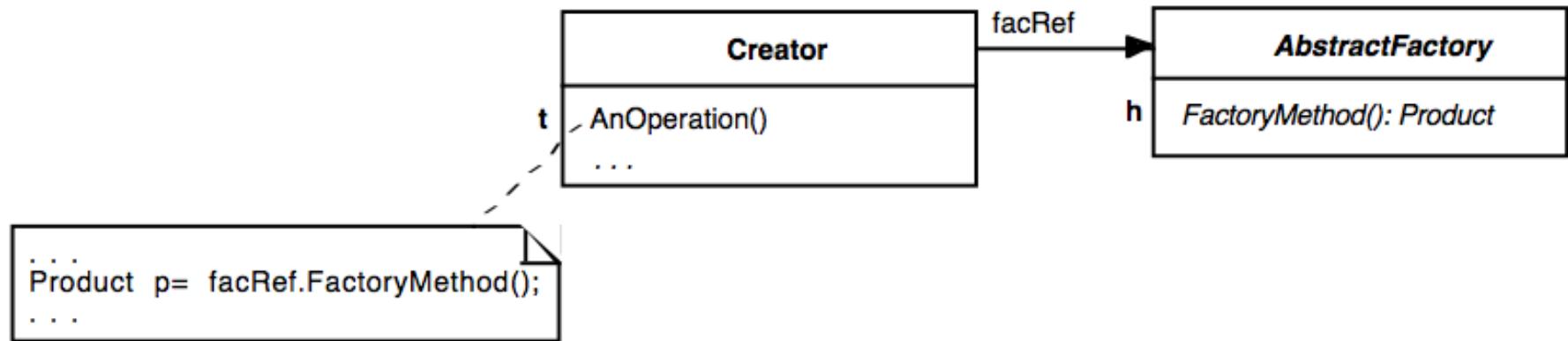
- Der Name und die Funktionalität der Hook-Methode beziehungsweise der Hook-Klasse drücken aus, welcher Aspekt bei einem Entwurfsmuster flexibel gehalten wird.
- Im Beispiel **Factory-Method** geht es darum, die **Objekterzeugung** flexibel zu halten.
- Analoges gilt für die Entwurfsmuster Abstract Factory, 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.

# Flexible Objekterzeugung auf Basis von Meta-Informationen (zB in Java und C#)



- + keine Unterklassen erforderlich
- Umgehung der statischen Typprüfung

# Factory Method (Hook Method) → Abstract Factory (Hook Object)



- die Hook-Methode **FactoryMethod()** wird einfach in eine separate Klasse oder Schnittstelle verschoben