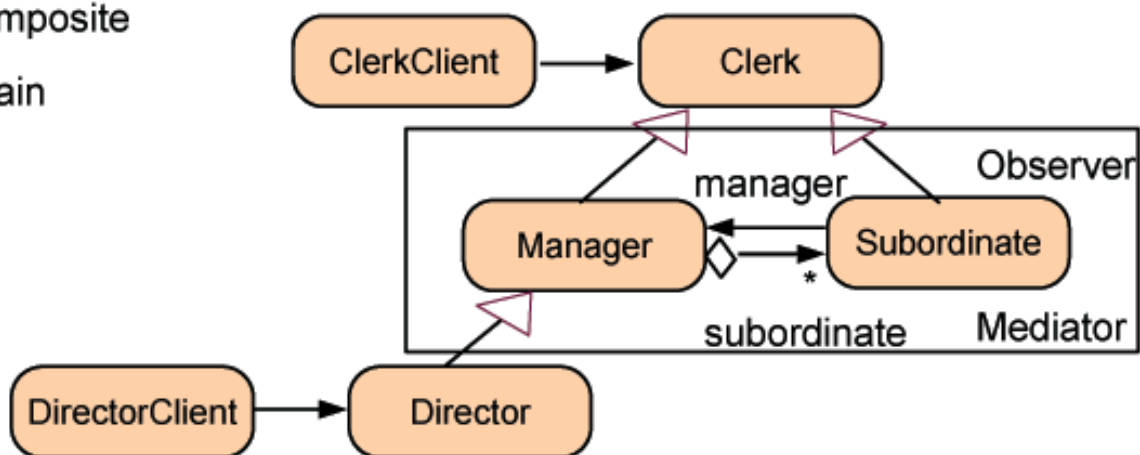


Aßmann's Law

Kleine Fibel zu Design Patterns und Frameworks

Composite
Chain



Vorlesungsskript zur Vorlesung
Design Patterns und Frameworks
im Wintersemester 2007/2008
von Prof. Uwe Aßmann

Zusammenstellung von Dietrich Kammer

Vorwort	3
0 Einleitung	5
0.1 Standard Probleme	5
0.2 Geschichte	7
0.3 Einige Definitionen	8
1 Design-Patterns	11
1.1 Einfache Patterns für Variabilität	11
1.2 Creational Patterns für Variabilität	16
1.3 Einfache Patterns für Erweiterbarkeit	18
1.4 Überbrückung von Architektur-Inkompatibilität	23
1.5 Usability von Design Patterns	26
2 Rollenbasierter Entwurf	29
3 Frameworks und Produktlinien	35
3.1 Framework Variation Patterns	35
3.2 Framework Extension Patterns	40
3.3 Tools and Materials Pattern-Language	45
3.4 Eclipse und Framework Extension-Languages	48
3.5 SAP R/3 Framework	50
3.6 San Francisco Framework für Business Applications	52
3.7 Framework-Dokumentation	54
3.8 Trustworthy Framework Instantiation	55
3.9 Binäre Kompatibilität von Framework-Plugins	58
4 Refactoring and Beyond	59

Vorwort

Der vorliegende Almanach dient in erster Linie der persönlichen Prüfungsvorbereitung, indem der Stoff der Vorlesung **DESIGN PATTERNS AND FRAMEWORKS** an der TU Dresden in Fließtext zusammengefasst wird. Dafür werden unterschiedliche Quellen genutzt, die nicht explizit ausgewiesen werden. Ausgangspunkt ist das Vorlesungsskript in Form der Folienpräsentation von Prof. Dr. Uwe Aßmann aus dem Wintersemester 2007/2008. Zudem wird auf Vorlesungsmitschriften zurückgegriffen, sowie auf das Buch **DESIGN PATTERNS** der Gang Of Four (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides).

Ausgewählte und in der Vorlesung referenzierte Forschungsartikel finden ebenso Verwendung. Die in englischer Sprache gehaltene Vorlesung wird hier weitestgehend übersetzt. Prominente Ausnahme ist der Begriff der Entwurfsmuster, der hier getreu der Vorlesungsüberschrift weiterhin als Design-Pattern erscheint. Des Weiteren finden Eindeutschungen gemäß des gesunden Menschenverstands statt und sollen nicht übertrieben eingesetzt werden.

Größere Konzepte, welche noch nicht erläutert und neu eingeführt werden, sind durch **FETTE, ENGE SCHRIFT MIT KAPITÄLCHEN** ausgezeichnet. Noch nicht erklärte Design-Patterns, sowie wichtige Begriffe und Namen von Forschern sind durch **NORMALE ENGE SCHRIFT MIT KAPITÄLCHEN** hervorgehoben. Quellcode wird durch `nicht-proportionale Schrift` ausgezeichnet.

Alle Angaben und Inhalte werden ohne Gewähr zur Verfügung gestellt.

0 Einleitung

Die Vorlesung beschäftigt sich damit, unterschiedliche Patterns mit ihren jeweiligen Anreizen (Incentives) darzustellen. Dabei werden Patterns für Variabilität und Erweiterbarkeit von Systemen behandelt, mit dem Ziel besonders Frameworks und Produktlinien zu verstehen. Als Beispiel können hier Frameworks für die Produktlinien von Autos gesehen werden. In modernen Autos wird bereits 90% der Innovation durch die integrierte Software erreicht. Wer es noch nicht getan hat, kann mit dem Praktomat des Lehrstuhls für Software-Technologie die Programmiersprache JAVA erlernen.

Strukturen für große Systeme ab 100 KLOC werden dabei unter Nutzung von Layered Frameworks und Facetten modelliert. Dazu kommt die Beschreibung eines anderen Wegs für den objektorientierten Entwurf, der rollenbasierte Entwurf.

Die Gliederung geht dabei ausgehend von grundlegenden Patterns zu deren Zusammenspiel mit Frameworks. Darauf aufbauend werden Pattern Languages besprochen und konkrete Frameworks betrachtet.

0.1 Standard Probleme

Eine Reihe von Standard-Problemen in der Software-Technologie können mit Hilfe der sogenannten Design-Patterns gelöst werden.

Unter **VARIABILITÄT** wird der Austausch von Teilen eines Software-Systems verstanden. Es kann sich dabei um Variationen mit komplexer Parametrisierung handeln. Es wird statische und dynamische Variabilität unterschieden. Von statischer Variabilität ist die Rede, wenn zur Entwurfs- oder Kompilierungszeit verschiedene Anpassungen vorgenommen werden. Im dynamischen Fall wird der Austausch zur Laufzeit betrachtet.

Grundbegriffe der Variabilität sind also Variation, Austausch und Parametrisierung. Es müssen Gemeinsamkeiten und die Variabilität formuliert werden. Auf diese Weise erhalten wir einen **COMMON PART**, der meist in einem Framework liegt. Durch die Parametrisierung an **VARIATION POINTS** kann eine Spezialisierung erfolgen. Die hier verwendeten Design-Patterns arbeiten zum Beispiel mit Templates und Hooks, beispielsweise **TEMPLATE METHOD**, **TEMPLATE CLASS** und **DIMENSIONAL CLASS HIERARCHIES** oder auch **BRIDGE**. Ebenso können Creational-Patterns eingesetzt werden, diese sind **FACTORY METHOD**, **FACTORY CLASS** und **BUILDER**. Ausgetauscht werden dabei die Kommunikation, Policies oder auch Material.

ERWEITERBARKEIT trägt dem Grundsatz Rechnung, dass Software sich ändern muss. Dabei werden neue, unvorhergesehene Varianten unterstützt. Diese Evolution beschreibt einen dynamischen Wandel. Dem trägt die **OBJECT RECURSION** im Gegensatz etwa zur **TEMPLATE METHOD** Rechnung. Weitere Patterns sind **OBJECTIFIER (STRATEGY)**, **DECORATOR** oder **PROXY**, **COMPOSITE** oder **CHAIN OF RESPONSIBILITY**, **VISITOR** und **OBSERVER**. Mit Facetten können dabei parallele, disjunkte Klassenhierarchien zum Einsatz kommen, durch die geschichtete Frameworks aufgebaut werden können. Mit dem unter Variabilität genannten Template und Hook kann gegebenenfalls Erweiterung betrieben werden, nicht nur Variation.

Das **GLUEING** beschreibt die Überbrückung, Adaptierung und Verbindung von Komponenten unterschiedlicher Architekturen, falls diese nicht übereinstimmen (Architectural Mismatch). Die Kopplung von unabhängiger Software wird so möglich. Architekturunterschiede werden zum Beispiel bei Kommunikationsprotokollen überbrückt. Ebenso erfolgt die Behandlung heterogener Komponenten, mit ihren Repräsentationen, Orten (bei verteilten Architekturen) und Kontrollflussstrukturen. Eine anonyme und skalierbare Kommunikation ist zentraler Bestandteil beim Glueing.

Ein weiteres Problem ist die **OPTIMIERUNG** von Software, welche oft Effizienz auf Kosten der Flexibilität bringt. Der Fluss von Daten und Kontrollanweisungen in interaktiven Anwendungen kann auch durch Pattern erfasst werden.

Das Basiswerkzeug der **ROLLENMODELLIERUNG** beschäftigt sich mit den verschiedenen Rollen, die ein Objekt in der Anwendung spielen kann. Dies gibt auch Auskunft darüber, wie die Entwurfsmuster auftauchen. Design-Patterns können zudem

durch Rollenmodelle statt durch beispielhafte Klassendiagramme strukturell erfasst werden. Rollenmodell-Komposition, sowie Rollen-Mapping führt dabei zum Entwurf von Klassendiagrammen gemäß der Entwurfsmuster.

Im besonderen Feld der **FRAMEWORK-PATTERNS** wird Variabilität mit unterschiedlichen Hooks erreicht. Durch diese wird das Framework instantiiert. Geschichtete Frameworks können beispielsweise durch das **ROLE OBJECT** Pattern modelliert werden.

Die im Verlauf betrachteten **TOOLS AND MATERIALS** ermöglichen die Strukturierung einer Anwendung anhand von Metaphern.

0.2 Geschichte

Zu Beginn der 70er Jahre kamen mit dem Window und dem Desktop die ersten konzeptionellen Patterns in Smalltalk bei Xerox auf. 1978 entwickelte Reenskaug das MVC-Pattern (Model-View-Controller). 1987 wurde erstmals der Begriff der Pattern-Language für die Software-Entwicklung, speziell in der objektorientierten Programmierung, entdeckt. Dies geschah durch die Herren CUNNINGHAM, unter anderem für Wikis verantwortlich, sowie BECK, welcher an der Entwicklung der Extreme-Programming-Methode beteiligt war. 1991 beendet GAMMA seine PhD-These zu Design Patterns. 1994 haben wir es mit der ersten PLOP zu tun, der Konferenz zu **PATTERN LANGUAGES OF PROGRAMMING**. 1995 erscheint das Buch zu Design-Patterns der **GANG OF FOUR**, kurz **GOF**. 1997 stellt RIEHLE die Verbindung zwischen Rollenmodellen und Design-Patterns her. 2005 findet das Konzept der **COLLABORATIONS** Einzug in die UML, womit Klassen-Rollen-Modelle erstellt werden können.

Als geistiger Vater der Design-Pattern-Ideologie wird CHRISTOPHER ALEXANDER betrachtet. In seinem 1978 erschienenem Buch **THE TIMELESS WAY OF BUILDING** doziert er über Muster und Mustersprachen in der Architektur. Dabei wird festgestellt, dass die **QUALITY WITHOUT A NAME**, also kurz die Schönheit von Entwürfen, nicht erfunden werden kann, sondern aus Pattern-Languages generiert werden muss.

0.3 Einige Definitionen

Ein Design-Pattern ist die Beschreibung einer Standard-Lösung für ein Standard-Problem in einem bestimmten Kontext. Es wird abgezielt auf die Wiederverwendung von Entwurfsinformationen. Nötig ist eine strenge Mißachtung von Originalität.

Das Muster enthält einen **BAD SMELL**, ein Problem in der Struktur oder einem Graphen. Der **GOOD SMELL** ist die Standardlösung für die Struktur, während mit **FORCES** der Kontext und die Bedingungen umschrieben werden. Die Transformation geht vom Bad Smell geleitet von den Forces zu einem Good Smell. Die Beschreibung aus dem GOF-Buch enthält für je ein Pattern den Namen, Motivation, Einsatz, Lösung, Bekannter Einsatz, Verwandte Pattern.

KONZEPTIONELLE PATTERNS beschreiben mit Hilfe von Begriffen aus der Anwendungsdomäne. Beispiele sind der Desktop und Tools and Materials.

Der Nutzen von Patterns besteht in einer verbesserten Kommunikation zwischen Entwicklern und sonstigen eingeweihten Personen. Es entsteht eine Art Ontologie des Software-Entwurfs. Die Dokumentation abstrakter Entwurfskonzepte wird unterstützt. Eine allgemeine Vereinfachung schlägt sich in den Bereichen Systemkomplexität, Testbarkeit, Evolution, Dokumentation und Wiederverwendung nieder.

Ein **DESIGN-PATTERN** kann als Übereinanderlegen einer einfachen Struktur auf die statische oder dynamische Semantik eines Systems verstanden werden. Es handelt sich um ein **NUGGET OF INSIGHT** (nach **APPLETON**).

Es kann sich dabei um folgende Arten von Mustern handeln. Die sogenannten **IMPLEMENTIERUNGS-PATTERNS** sind Programming-Patterns, auch Idiome genannt. Sie sind abhängig von der Programmiersprache und werden durch Konstrukte der Programmiersprache beschrieben. Workarounds sind Idiome, die ein nicht vorhandenes Programmierkonstrukt nachahmen. Die sogenannten **FUNDAMENTAL DESIGN PATTERNS** werden nicht als Sprachkonstrukt beschrieben. **ARCHITECTURAL PATTERNS**

geben die grobe Struktur von (Sub-)Systemen an. Die speziellen **FRAMEWORK-INSTANTIIERUNGSPATTERNS** werden noch behandelt.

Ein **ANTI-PATTERN** ist ein defektes Pattern mit einem Bad Smell innerhalb der Anwendung. Es kann sich um Spaghetti-Code, If-Kaskaden, überall verteilte Casts oder Nicht-Modularität handeln.

Hier steht etwas Text, da zwei aufeinanderfolgende Definitionen typographisch ziemlich blöd aussehen.

PROZESSPATTERNS sind immer im Begriff etwas zu tun. Sie treten bei Automatisierung z.B. in Workflows auf.

Weiterhin werden Re-Engineering-Patterns identifiziert, die im Durchführen bestimmter Testvorgänge bestehen, z.B. das Lesen des gesamten Codes in einer Stunde. Organizational Patterns sind etwa die wohlbekannte Linien- oder Matrixorganisation.

Eine **PATTERN LANGUAGE** ist eine Menge von Patterns für verwandte Probleme in einer bestimmten Domäne. Beispiel ist Tools and Materials.

1 Design-Patterns

Das erste Kapitel beschäftigt sich mit der Beschreibung der grundlegenden Technologie für Frameworks und Produktlinien, den Entwurfsmustern.

1.1 Einfache Patterns für Variabilität

Grundlage für diese Patterns ist das konzeptionelle **TEMPLATE AND HOOK** Pattern. Damit können viele Produkte aus einer Code-Basis erstellt werden. Beispiele sind das Office-Backbone mit den Ausprägungen Word, Powerpoint, Excel usw. Der Anreiz des Patterns ist es, Gemeinsamkeiten und Unterschiede auszudrücken. Dies erfolgt durch ein für viele Produkte festes Template, in welches spezifische Implementierungen eingehängt werden können. An den definierten Holes, Empty Spaces, Hooks, Slots oder Hotspots werden konkrete Werte gebunden.

TEMPLATE METHOD implementiert Template und Hook in der gleichen Klasse. Variables Verhalten wird somit erreicht, indem die Invariante vom varianten Teil eines Algorithmus getrennt wird. Der Hook kann dabei nicht-erweiterbar oder erweiterbar gestaltet werden. Binding the Hook erfolgt durch das Ableiten von der abstrakten Superklasse, sowie Implementierung der Hook-Methode.

Der Unterschied zwischen Variabilität und Erweiterbarkeit liegt darin, dass entweder nur ein einmaliges Binding erfolgt, oder der Hook mehrfach gebunden wird. Letzteres ergibt für Produktlinien eine größere Variationsmöglichkeit.

Das **OBJECTIFIER** Pattern beruht auf einfacher Polymorphie mit Objekten, also Delegation. Die Hook-Methode wird aus der Klasse mit der Template-Methode herausgenommen und eine abstrakte Klasse mit einer abstrakten Methode geschaffen, die entsprechend implementiert wird, um Variabilität zu erreichen. In C++ gibt es für diese Funktionalität auch sogenannte Functional Objects. Einfacher Austausch wird so auch zur Laufzeit möglich, auf Kosten von mehr Dispatch.

TEMPLATE CLASS implementiert Hook-Methode und Template-Methode in unterschiedlichen Klassen. So wird der konsistente Austausch von mehreren Methoden eines Algorithmus gewährleistet. Dieser Austausch kann auch zur Laufzeit erfolgen. Das Pattern bildet die Basis für weitere: Bridge, Builder, Command, Iterator, Observer, Prototype, State, Strategy und Visitor. Binding the Hook erfolgt durch das Ableiten einer konkreten Klasse von der abstrakten Hook-Superklasse, sowie Implementierung der Hook-Methode.

Mit Generics bei GENERIC TEMPLATE CLASS ist auch eine statische Instantiierung möglich, was zu weniger Runtime-Dispatch, Typsicherheit und damit zu effizienterem Laufzeitverhalten führt. Der Polymorphismus wird dabei jedoch ausgeräumt, aufgrund der statischen Vorauswahl. Bei der Bounded Genericity wird die Instantiierung mit Generics insofern begrenzt, dass Vererbung nur von einem bestimmten Typ erfolgen kann.

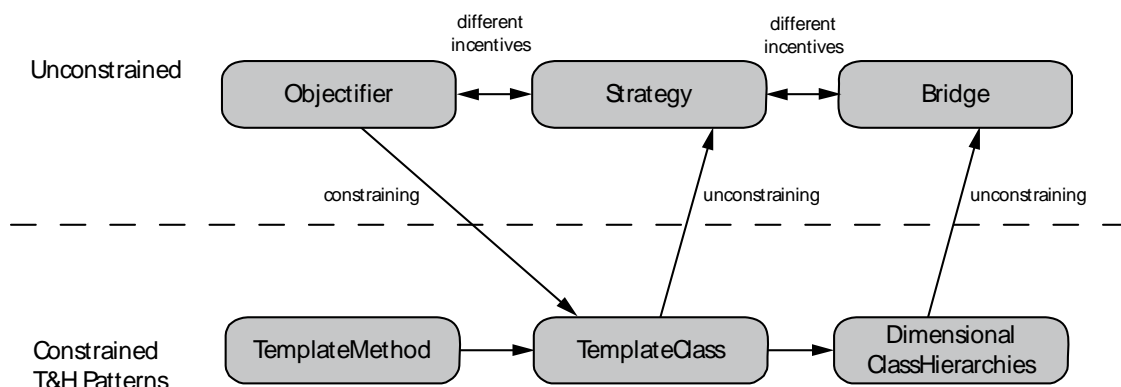


Abb. 1: Zusammenhang der einfachen Variabilitäts-Patterns.

STRATEGY besitzt die gleiche Struktur wie Template Class, jedoch stehen hier die Rollen des Clients und des Algorithmus als Anreiz im Vordergrund. Das Pattern taucht auch im Model-View-Controller auf.

Bei DIMENSIONAL CLASS HIERARCHIES ist auch die Variation des Templates zugelassen. Dadurch ist also auch die Adaptierung des Template-Algorithmus möglich. Im Gegensatz zu der schwachen Veränderlichkeit bei Template Class ist nun auch die

Veränderung der Template-Objekte für die verschiedenen Dimensionen in der Anwendung möglich. Eine Double-Variability auf Caller- und Hook-Seite entsteht, sowie zwei getrennte Klassenhierarchien, die durch Constraints verbunden sind.

Bei Verwendung von **BRIDGE** wird ähnlich wie bei Dimensional Class Hierarchies vorgegangen. Das bedeutet Abstraktion und Implementierung werden unabhängig variiert. Zudem wird auf Template- und Hook-Constraints verzichtet. Werden mehrere Bridge-Patterns kombiniert erhält man Facetten. Durch die Faktorisierung von Klassen in Facetten wird die Wiederverwendung erhöht.

Für konsistente Variation bei vorhandenen Parallelismus-Bedingungen sollte **PARALLEL CLASS HIERARCHIES** eingesetzt werden. Hier werden Bridges mit Constraints modelliert. Dabei sind die Dimensionen folgenderweise nicht unabhängig. Dies kann zum Beispiel mit der Object Constraint Language (OCL) im Modell ausgedrückt werden. Wenn ein bestimmter Teil gewählt wird, so ist ein anderer obligatorisch. Zur Laufzeit sollten nicht erst diese Constraints durchgesetzt werden.

Bei unabhängigen Dimensionen von mehrdimensionalen Systemen wird die **FACETTEN-KLASSIFIKATION** benutzt. Diese wird durch Bridge oder Dimensional Class Hierarchies implementiert. Der kombinatorische Raum wird so effektiv modelliert und die Vererbungshierarchien weniger tief gehalten. Die feste Modellierung mit (mehrfacher) Vererbung ist deswegen schlecht, da die Aufsuche von Eigenschaften sehr kompliziert wird und es zu Überschreibungen kommt. In Java ist zudem nur die Mehrfachvererbung von Interfaces zulässig, wodurch die Wiederverwendung von Code verhindert wird.

Eine **FACETTE** ist eine orthogonale Dimension eines Modells. Die abstrakten Facettenklassen bilden dabei ein eigenes Modell. Die Vererbungshierarchie wird dabei vereinfacht. Alle Facetten sind unabhängig und wissen nicht voneinander. Eine finale Klasse kann als Vereinigung aller Eigenschaften einer Facette modelliert werden.

Wird Bridge verwendet, so kann eine zentrale Facette als Abstraktion gewählt werden, welche alle anderen als Hook-Klassen verwendet. Wird wie in diesem Fall ohne einen Kern gearbeitet, so erhält eine primäre Facette Kenntnis über die ande-

ren Facetten entlang der Bridge-Verbindungen. Dies ist jedoch effizienter hinsichtlich des Speichers. Ein logisches Objekt wird bei der Facetten-Klassifikation mit mehreren physischen modelliert. Dies ist immer angebracht, wenn mehrere unterschiedliche Partitionen in einer Klasse existieren. Der Vorteil liegt in der dynamischen Variation und der Reduktion der Anzahl der Klassen. Nachteile bestehen im Fehlen einer Typprüfung für die Produktklassen, sowie fehlende Kontrolle über illegale Kombinationen. Zudem kommt es in diesem Zusammenhang auch zur Objekt-Schizophrenie. Beim Abfragen der Identität eines physischen Objekts kann kein Rückschluss auf das logische erfolgen. Der Speicherverbrauch für die Allokationen steigt, während die Geschwindigkeit der Anwendung sinkt. Besonders die letzten Punkte sprechen gegen den Einsatz bei eingebetteten Systemen.

Zusammenfassend ist noch einmal festzustellen, dass der Einsatz von Facetten-Klassifikationen bei unabhängigen Dimensionen eines komplizierten Modells anzuraten ist. Bridges sind gerade bei Sprachen ohne Mehrfachvererbung eine interessante Alternative. Bei Frameworks, bei denen Type-Checking von Vorteil ist, sollte auf Facetten-Vererbung zurückgegriffen werden.

Beim soeben beschriebenen Vorgehen entstehen sogenannte geschichtete Objekte (**LAYERED OBJECTS**). Falls Bridges zum Einsatz kommen, ist die Rede von unabhängigen Facetten. Jedoch können sehr wohl Abhängigkeiten existieren. Diese schlagen sich nieder in bestimmten Interfaces oder Verträgen zwischen den Facetten. Eine Modellierung kann hier durch Chain-Bridges erfolgen, womit den Abhängigkeiten von Informationen der unteren Schichten Rechnung getragen wird. Ein Layered Object Space beschreibt den Austausch von Informationen zwischen Schichten in einem gerichteten, azyklischen Graphen. Auch hier ist die Austauschbarkeit der Schichten gegeben, das Konzept ist allerdings breiter anwendbar, aufgrund der Möglichkeiten Abhängigkeiten zu modellieren.

Als Generalisierung für geschichtete Objekte und Facetten-Klassifikation werden **GESCHICHTETE FRAMEWORKS** und Systeme betrachtet. Eine Ausprägung sind Facet-Bridge Dimensional Systems, bei denen alle Dimensionen unabhängig sind und ein Kern verwendet wird, um diese zu aggregieren. Die zweite Variante sind Facet-Bridge Frameworks für facettenbasierte Systeme. Hier werden ein oder mehrere Schichten fest modelliert, während der Rest variabel ist und über Multi-Bridges

aggregiert wird. Dieses Facet-Framework ermöglicht Wiederverwendung auf mehreren Ebenen. Der gegenseitige Aufruf erzeugt den geschichteten Entwurf der Facetten und keinen Raum der Facetten.

Der facettenbasierte Entwurf bei Frameworks ist eine Best-Practice, um große Klassenhierarchien nicht komplett durch Vererbung zu modellieren, wenn diese auf Partitionierung basieren. Zuerst werden Facetten identifiziert und die Vererbungshierarchie in diese Facetten faktorisiert. Die Implementierung erfolgt dann über Bridges, bei abhängigen Facetten werden Schichten eingeführt und es erfolgt eine Implementierung über Chain-Bridges.

Bei geschichteten Frameworks mit Chain-Bridges schneiden die Klassen durch alle Schichten, was aber nicht immer für alle Klassen erfüllt ist. Bei geschichteter Architektur entsteht ein **LAYERED OBJECT FRAMEWORK** mit Chain-Bridges. Dadurch wird Variabilität, Erweiterbarkeit und Parametrisierung möglich. Die Wiederverwendung auf jeder Schicht ist der Grundgedanke. Die Rede ist hier auch von gestapelten Aspekten (**STACKED ASPECTS**). Dabei erfolgt die Parametrisierung der unteren Schichten durch die oberen. Techniken hierfür sind das aspektorientierte Weaving und das View-basierte Weaving, auch Hyperslice-Programming genannt. Von Hand kann die Implementierung mit Chain-Bridges und dem Role-Object-Pattern erfolgen. Prof. Aßmann prophezeit, dass geschichtete Frameworks die wichtigste Zukunftstechnologie in der Software-Technik darstellen.

1.2 Creational Patterns für Variabilität

Um komplexe Objekte effektiv und variabel zu allozieren, können Creational Patterns verwendet werden. Zusätzlich zum Erzeugen der Objekte können auch Konsistenzbedingungen durchgesetzt werden.

Die **FACTORY METHOD**, oder auch Polymorphic Constructor, ist ein Pattern, welches sprachenabhängig ist. Es umgeht die Beschränkung einiger objektorientierter Sprachen hinsichtlich des Austauschs von Konstruktoren. Oft wird dies nicht erlaubt. Grundsätzlich wird wie bei Template Method vorgegangen, wobei eine Produkt-hierarchie hinzukommt, aus der die Hook-Methode eines auswählt. Die Hook-Methode wird damit zu einem polymorphen Konstruktor. Dabei kennen abstrakte Klassen lediglich abstrakte Konstruktoren und erbende Klassen können den Konstruktor spezialisieren.

Neue Subklassen für unvorhergesehene Erweiterungen können über Vererbung und Reflection eingesetzt werden. Damit wird es möglich, beliebige neue Produktinstanzen zu erzeugen. Dieses dynamische Wissen über die Allokation macht die Anwendung allerdings langsamer. Die Möglichkeit einer Default-Implementierung kann über einen Super-Aufruf gegeben werden. So können gemeinsame Features genutzt werden. Die Factory Method wird oft in parallelen Hierarchien genutzt. Sie kreiert Objekte aus einer zweiten Hierarchie auf der gleichen Ebene und setzt so Parallelismus-Bedingungen um. Produkt-Subklassen können auch mit Generics erstellt werden.

Das Prinzip des Information-Hidings wird wie folgt umgesetzt. Abstrakte Klassen wissen nur wann ein Objekt alloziert werden soll, aber nicht welches. Diese Information ist in den Unterklassen gekapselt. Ein etwaiges Framework weiß nicht, mit welchen konkreten Klassen gearbeitet wird, sondern nur wann. Die Anwendung kann so den Typ frei bestimmen.

Soll eine Produktfamilie gemultiplext werden, wird dies über **FACTORY CLASS**, oder auch Abstract Factory erreicht. So kann die gesamte Produktfamilie einer Variante auf die andere wechseln, indem die Factory-Methoden in einer Klasse gruppiert

werden. Die abstrakte Factory bietet dabei ein Interface an, um Familien von verwandten Objekten zu erzeugen. Damit handelt es sich nunmehr um mindestens drei Klassenhierarchien. Die Factory-Hierarchie und zwei oder mehr Produkthierarchien. Die Produkthierarchien sind dabei parallel aufgebaut, das Factory-Pattern garantiert beim Erzeugen die Parallelismus-Bedingung. Eingesetzt wird dies zum Beispiel bei Window-Styles, Office-Systemen und Austausch von Sprachmodulen bei beliebigen Anwendungen.

Eine **PROTOTYPING FACTORY** kennt keine konkreten Factories, es gibt nur einen Prototyp, der vorhandene Produkte klonet. Variabilität wird durch das Klonen von Prototyp-Objekten erreicht. Es werden nur valide Kombinationen der Produkte gespeichert.

Eine weitere Möglichkeit für eine Factory ist die **INTERPRETIVE FACTORY METHOD**. Für das Hinzufügen weiterer Factory Methods wird hierbei eine Factory Method mit einem Parameter-String benutzt, statt alle Factories zu initialisieren.

Bei Nutzung von Factories wird das System unabhängig von der Art und Weise, wie Objekte erstellt werden. Die Konstrukturen werden versteckt, um den Typ-Austausch zu gewährleisten. Der Einsatz liegt nahe, wenn komplette Produktfamilien zusammen erstellt werden müssen. Die Abstract Factory ist damit vergleichbar mit der Hook-Klasse und wird oft als Singleton verwendet. Dabei wird sie mit Prototyp-Objekten parametrisiert.

Soll eine Factory mit Protokoll realisiert werden, wird das **BUILDER** Pattern verwendet. Dabei wird ein strukturiertes Produkt produziert, also ein Ganzes mit Teilen. Es handelt sich hierbei oft um große Business-Objects. Der Builder speichert dabei intern den Fortschritt der Konstruktion des Objekts. Versteckt werden die Struktur der Daten, also das Protokoll, der jeweilige Status und die Implementierung, ähnlich wie bei einem Iterator. Beispiele sind etwa Parser in Compilern und Datenbanken mit Integritäts-Constraints.

1.3 Einfache Patterns für Erweiterbarkeit

Mit den folgenden Patterns werden zum Beispiel parallele Klassenhierarchien als Implementierung für Facetten modelliert, sowie geschichtete Frameworks gebaut. Dynamische Erweiterbarkeit bedeutet, dass zur Laufzeit White-Spots erweitert gefüllt werden können. Bisher wurde nur die statische Erweiterung und Variabilität durch Vererbung usw. behandelt.

Die **OBJECT RECURSION** bildet die Grundlage für viele der Erweiterbarkeits-Patterns. Es findet eine Rekursion in Abhängigkeiten zwischen Klassen statt. Diese sind Vererbung und Aggregation. Je nach Multiplizität der Aggregation ergibt sich eine 1-Recursion, also eine Liste als Laufzeit-Struktur, oder eine n-Recursion, wodurch Bäume und Graphen entstehen. Die Abstrakte Superklasse definiert dabei gemeinsame Bedingungen für Terminatoren und Recursor. Alle können gleich behandelt werden, es werden stets alle Nachbarn oder Kinder bei einer Anfrage auf einem zusammengesetzten Objekt aufgerufen.

Das **COMPOSITE** Pattern ist als Instanz der n-ObjectRecursion anzusehen. Als älteres GOF-Pattern wurde es allerdings vorher beschrieben und ist weniger abstrakt. Der Client weiß nicht, ob ein Leaf oder Composite bearbeitet wird. Es findet ein einfaches Iterieren über die Knoten statt. Dabei stellt die abstrakte Superklasse den Vertrag bereit. Die entstehende Struktur ist dynamisch erweiterbar. Composite ist das Kernstück der funktionalen Programmierung und Attribut-Grammatiken.

Das Lieblingsauto von Prof. Abmann, der Phaeton wird übrigens mit Composite bis zur Schraube modelliert, zum Zweck des Total-Quality-Managements. So wird Tracking für alle Komponenten möglich, alle Teile werden als Business Objects umgesetzt.

Der **DECORATOR** ist wiederum eine Variante des Composite. Es wird durch eine Listenstruktur eine Hülle, oder auch Zwiebelhaut, aufgebaut. Damit handelt es sich bei dieser 1-ObjectRecursion um ein eingeschränktes Composite. Eine Decorator-Klasse ist eine Unterklasse einer Klasse mit einem Kind von genau dieser Klasse. Der letztendliche Kern, also das eigentliche Objekt wird so vor dem Client ver-

steckt. Ein Decorator kann später eingefügt werden, im Gegensatz zum Composite, welcher eine vorherige Design-Entscheidung ist. Mit dem Decorator kann so Legacy-Code weiterverwendet, oder Wrapper für Bibliotheken realisiert werden.

Es findet ein Pre- und Post-Processing statt, während die eigentliche Aktion des Objekts in der Mitte stattfindet. Man kann so von erweiterbaren Objekten sprechen, denen neue Features zur Laufzeit ohne Vererbung hinzugefügt werden. Dies ist geradezu verpflichtend, falls die Vererbungshierarchie zu komplex werden würde, oder in einer Bibliothek unmöglich ist. Eingesetzt wird der Decorator zum Beispiel, um Core-Widgets dynamisch zu dekorieren, wie zum Beispiel Scrollbars und Frames. Ebenso können Objekte zur Laufzeit persistent gemacht werden.

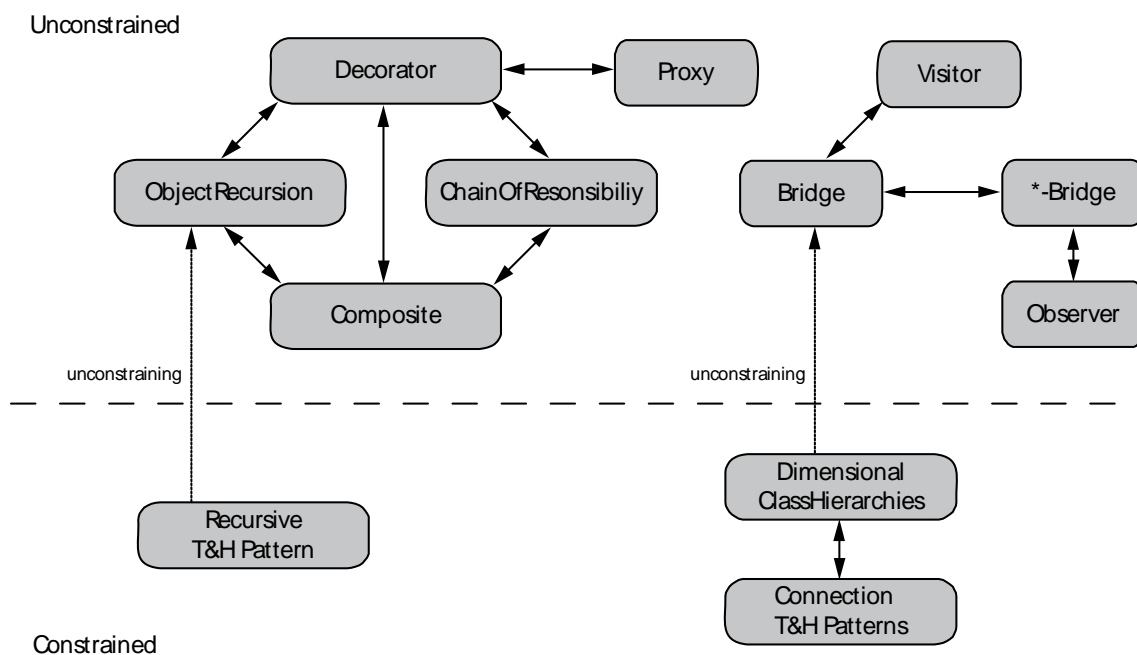


Abb. 2: Zusammenhang zwischen den Erweiterbarkeits-Patterns.

Für ambitionierte Hacker stellt COBOL die Möglichkeit bereit, Funktionsnamen zu überschreiben und bei C kann der Linker ausgetrickst werden, um so zu dekorieren. Sonstige Varianten des Decorators bestehen im Weglassen der abstrakten Superklasse, falls nur eine Erweiterung geplant ist. Des Weiteren kann mit einer De-

corator-Family in einer Hierarchie der gemeinsame Austausch konsistent für alle Objekte erfolgen.

Die Möglichkeit der Verkettung von Decorator-Klassen ermöglicht es, beliebig viele neue Features hinzuzufügen. In der Tat handelt es sich um eine spezielle **CHAIN OF RESPONSIBILITY**, in der zuerst die Decorator-Klassen kommen und zuletzt das eigentliche Objekt.

Die herkömmliche Chain Of Responsibility dient der Delegierung einer Aktion an eine Liste von Delegierten. Es wird das Daisy-Chain-Prinzip genutzt, in dem versucht wird ein Problem zu lösen oder so lange weiterzureichen, bis es gelöst ist. Ganz wie in der Bürokratie wird die Verantwortlichkeit für einen Dienst weitergegeben. Es werden so dynamische Aufrufe unterstützt. Der Empfänger der Nachricht ist zur Laufzeit, sowie zur Zeit der Allokation unbekannt. Dies ist ein Grundprinzip der Service Oriented Architecture (SOA). Die Kette ist zur Laufzeit dynamisch mit neuen Empfängern erweiterbar. Die anonyme Kommunikation setzt voraus, dass die Identität des Empfängers oder mehrerer Empfänger unwichtig ist. Da es keine finale Leaf-Klasse gibt, also keinen Back-Arrow zur abstrakten Superklasse, muss das Ende der Kette mit einem NULL-Test geprüft werden. Alle Beteiligten Klassen wissen nur über ihren Nachfolger Bescheid.

Der **PROXY** ist ein limitiertes Erweiterbarkeits-Pattern, da es nur einmal erweiterbar ist. Eine Proxy-Klasse gilt als Repräsentant eines echten Objekts, welches von ihr versteckt wird. Die Struktur ähnelt dem Decorator, aber es gibt nur einen direkten Pointer zur Schwesterklasse und keine Objekt-Rekursion. Es gibt also keine verketteten Proxies. Stattdessen sammelt der Proxy Referenzen zum echten Objekt oder Facetten-Objekt.

Es existieren viele Varianten: Smart Reference, Indirection Proxy (macht das Subjekt austauschbar), Virtual Proxy (Creation on demand), Remote Proxy, Caching Proxy, Protection Proxy. Ein Proxy gewährleistet kontrolliertes Verhalten, auch wenn das eigentliche Subjekt zerstört wird, indem sogenannte dangling edges vermieden werden. Dies sind Referenzen ins Nirvana. In C++ ist das Überladen des Pfeil-Operators möglich und somit die Möglichkeit für Proxies direkt in der Sprache integriert. Modula 3 bietet mit dem Smart-Pointer-Konzept die Möglichkeit Abstürze zu vermeiden.

Der VISITOR ist laut Prof. Abmann ein Ugly-Pattern. Es umschifft die Beschränkungen moderner objektorientierter Programmiersprachen beim Double-Dispatch. ADA95 bot einen Methodenaufruf nach der Syntax $f(o, p1, p2)$ an, wobei das erste Argument das Receiver-Objekt darstellt. Dies ist eine Smooth-Extension der funktionalen Programmierung. Die übliche Syntax $o.f(p1, p2)$ wird von Puristen als richtig angesehen. Der Polymorphismus auf mehreren Argumenten wird nun mit dem Visitor-Pattern erreicht. Es handelt sich dabei um eine Instanz des Dimensional-Class-Hierarchies-Patterns. Es existiert eine polymorphe Datenstruktur, die Template-Hierarchie. Dazu kommt ein polymorpher Algorithmus, die Hook-Hierarchie. Beide Hierarchien können einfach erweitert werden. Der Double-Dispatch erfolgt zuerst bei der abhängigen Dimension. Dazu wird das Objekt im Produkt-Baum ausgewählt und die Accept-Methode mit einer Visitor-Klasse als Argument aufgerufen. Dann erfolgt die Auswahl des Algorithmus durch den Aufruf der Visit-Methode am übergebenen Visitor durch das Produkt, dabei wird eine Selbstreferenz als Argument übergeben. Dabei arbeitet jeder Visitor mit jedem Element zusammen. Nur durch dieses Konstrukt kann der Kontext dispatcht werden, andernfalls müsste auf Switch-Statements oder eine andere Programmiersprache zurückgegriffen werden, zum Beispiel MultiJava. Hier können Multi-Methoden gruppiert werden.

Beim Hinzufügen von neuen Methoden ist der Aufwand mit dem Visitor-Pattern geringer, da nur in einer Hierarchie Änderungen vorgenommen werden müssen. Der Visitor ist auch als Schicht in geschichteten Entwürfen möglich. Durch die getrennten Hierarchien wird zudem die Wiederverwendung von Code vereinfacht.

Eine *-BRIDGE stellt eine Bridge mit einer Kollektion von Hooks dar. Die Hook-Methoden aller Hook-Klassen werden dabei im Template aufgerufen.

Das OBSERVER Pattern stellt eine Art Event-Bridge dar und ermöglicht die lose Kopplung von Kommunikation. Ein Update bei einem Subjekt feuert ein Ereignis beim Observer, welcher selbst über einen Pull-Mechanismus den benötigten Zustand des Subjekts anfordert. Das Subjekt ist so unabhängig vom Observer, aber der Observer kennt sein Subjekt. Er muss sich anonym beim Subjekt registrieren. Daher sind die Dimensionen nicht komplett getrennt wie bei der Bridge. Multi-Cast und Broadcast können umgesetzt werden. Hier wird Aspektorientierung per Hand imp-

lementiert. Es ist auch möglich einen Push-Mechanismus in der Notify-Methode zu realisieren. Der Observer ist auch ein Link im Model-View-Controller.

Über einen **CHANGE MANAGER**, Event-Bus oder auch Mediator kann ein Mapping von Subjekten zu Observern verwaltet werden. Filterung und Kaskadierung von Ereignissen zwischen ihnen kann realisiert werden.

Dieses Vorgehen ist in Window-Systemen, also interaktiven Frameworks mit Event-Cues ubiquitär. Bei Macintosh-Computern wird im MACH-Betriebssystem-Kern auf diese Weise die Interprozess-Kommunikation gewährleistet. Mit dieser Erweiterbarkeit zur Laufzeit wird auch der Software-Cocktail-Mixer realisiert (ein ambitioniertes Projekt zur Erstellung von Anwendungen on-the-fly). Für mehr Informationen: <http://www.eecs.ucf.edu/~leavens/FoCBS/assmann.html>.

1.4 Überbrückung von Architektur-Inkompatibilität

Bei Inkompatibilitäten von Architekturen werden Brücken zwischen Modulen eingesetzt. Dies wird betrieben bei der Enterprise Application Integration (EAI) und z.B. bei Legacy-Systemen angewendet, bei denen Konnektoren zwischen den Ports entwickelt werden. Ein Beispiel dafür ist AESOP von GARLAN, ALLEN und OCKERBLOOM.

Probleme treten dabei mit den Komponenten und Konnektoren auf. Gründe dafür sind unterschiedliche Annahmen über das Komponentenmodell, die globale Architektur-Struktur, den Konstruktionsprozess (Bibliotheken, Sprache), das Datenmodell (Datumsformat) und das Kontrollmodell. Auch Protokolle können inkompatibel sein, wenn grundlegend unterschiedlich auf asynchrone und synchrone Kommunikation spekuliert wird. Das Komponentenmodell gibt Interfaces und Austauschbarkeit an und beschreibt das Zusammenwirken der Welt mit dem Interface-Modell. Das Typ-System beschreibt die Domänen und damit die Application-View auf die Welt. Die meisten Datenbanken sind grundlegend ill-designed für Anwendungen im Umfeld von Informationssystemen. Sie beschäftigen sich nicht damit Objekte persistent zu machen, sondern treten als Black-Box mit sicherem Input und Output in Erscheinung. Sie sind für sich "master of the universe", statt sich in die Anwendung zu integrieren.

Lösungsansätze beruhen darauf, alle Annahmen explizit zu machen und zu dokumentieren. Leider existieren dafür wenige Formalismen. Implizite Annahmen verletzen das Information-Hiding-Prinzip und verhindern Variabilität. Komponenten müssen generell unabhängiger gestaltet und eine Bridging-Technologie zur Verfügung gestellt werden. Architekturstile müssen unterschieden werden. Für all diese Anforderungen können Design-Patterns genutzt werden.

Die Nutzung von Erweiterbarkeits-Patterns können Verhalten überbrücken. Chain Of Responsibility kann Objekte filtern und Verhalten adaptieren. Proxy kann zum Beispiel zur Übersetzung von Datenformaten herangezogen werden. Observer fügt zusätzliches Verhalten hinzu, indem Ereignisse überwacht werden, während Visitor zum Beispiel neue Algorithmen für eine Datenstrukturhierarchie ermöglicht. Um

Daten-Inkompatibilität zu umgehen ist Decorator verwendbar, der als Wrapper Verhalten adaptiert und überbrückt. Protokoll-Inkompatibilitäten können allerdings nicht korrigiert werden. Die Bridge kann Entwürfe für verschiedene Plattformen faktorisieren. Abstraktions- und Implementierungskomponenten sind unabhängig.

Als neues Pattern wird in diesem Zusammenhang der **ADAPTER** eingeführt. Er ist ein Proxy, der ein Interface auf ein anderes mappen kann. Zudem können Protokolle und Datenformate adaptiert werden. Da der Adapter nur einmal passiert wird, ist die Anpassung des Kontrollflusses allerdings schwierig. Es wird Delegierung genutzt. In Komponentenmodellen werden Standard-Interfaces verwendet. Wenn Klassen mit anwendungsspezifischen Interfaces existieren, können Adapter-Patterns ein Mapping herstellen. Der Adapter erbt vom Interface der Zielklasse und passt das Interface des adaptierten Objekts darauf an. Ein spätes Einfügen eines solchen Adapters ist möglich.

Eine **FACADE** ist ein Objekt-Adapter, der ein komplettes Subsystem versteckt. Sie kann auch als Proxy angesehen werden. Das eigene Interface wird auf das der versteckten Objekte gemappt. Es werden Interfaces, Daten und Protokoll in einer Runtime-Bridge verbunden. Dies ist ähnlich zum Decorator, geschieht aber ohne die Rekursion. Die Facade hat ein exklusives Anrecht auf externe Bibliotheken oder Subsysteme.

Der Class Adapter nutzt zur Anpassung statt Delegierung Mehrfachvererbung. Da dies nicht in allen Programmiersprachen möglich ist, kann zum Beispiel bei Java nur mit Interfaces gearbeitet werden, wodurch die Wiederverwendung von Code verhindert wird.

Der 2-Way Class Adapter stellt einen Role-Mediator dar. Er hat mehr als eine Zielklasse und jede spielt eine bestimmte Rolle eines konkreten Objekts. Auf diese Weise werden ganze Hierarchien gekoppelt. Der 2-Way Decorator und Adapter zum Koppeln von Hierarchien ergeben eine Kette von Strategien, zum Beispiel für einen Daten-Generator.

Mit einem **MEDIATOR**, oder auch Broker, wird ein n-Way Proxy zur Kommunikation in Verbindung mit Bridge aufgebaut. Dies ist ein zentrales Konzept bei CORBA, wo ein Web Service Broker Dienste anbietet, deren Interfaces mit der Web Service

Description Language (WSDL) beschrieben sind. Ermöglicht wird dadurch eine anonyme Kommunikation und der Aufbau anonymer Kommunikationsnetze. Ein Proxy-Objekt versteckt die Kommunikation zwischen Partnern, welche alle ein Mediator-Objekt benutzen. Mit einer Bridge werden die Kommunikationspartner verbunden. Mediator und Partner-Hierarchien sind unabhängig voneinander veränderbar. Der **OBSERVER WITH CHANGE MANAGER** verbindet dabei noch Observer mit dem Mediator Pattern.

Das Pattern Repository Connector erlaubt die Kollaboration von Werkzeugen ohne gegenseitiges Wissen. Dies geschieht über die Kopplung ihrer Repositories, wie von STÖLZEL 2005 mit Hilfe von Lazy-Indirection-Proxies beschrieben. Es wird ein Master-Werkzeug ausgewählt, die anderen sind Slave-Werkzeuge, in deren Repositories das Master-Repository lediglich gespiegelt wird. Das eigentliche Repository wird vom Slave-Tool entfernt und eine Adapterschicht eingezogen. Die echten Klassen werden dann on-demand im Master-Repository erstellt.

Unterschiedliche Annahmen sind also mit Erweiterbarkeitspatterns überbrückbar.

1.5 Usability von Design Patterns

Der Einsatz von Design Patterns ist meist domänenspezifisch und erfolgt in Firmen die sich zum Beispiel mit Bank- oder Telekommunikations-Software beschäftigen. Pattern können entweder für die Problems- oder Lösungsdomäne geschrieben werden. Die verwendeten Begriffe kommen dann entweder aus der Anwender-Domäne oder aus der Entwickler-Domäne.

Wichtig ist der Aufbau eines Katalogs für die Firma, einer sogenannten **EXPERIENCE FACTORY**. In ihr sind sowohl domänen- als auch firmenspezifische Patterns enthalten. Es wurden drei Gruppen von Pattern-Benutzern identifiziert. Pattern-Gurus können sie beschreiben, während Pattern-Benutzer sie nur erkennen und einsetzen können. Pattern-Ignoranten können allgemein nichts damit anfangen.

Die Schwierigkeiten beim Umgang mit Patterns kommen daher, da der Fokus auf die Gemeinsamkeiten gelegt werden muss. Dies ist schwierig, aber sehr wichtig. Das **PATTERN-MINING** ist ein iterativer Prozess, um Pattern zu identifizieren und zu beschreiben. Zentrale Fragen sind: Warum wurde so entworfen? Ist die Komplexität wirklich nützlich? Welche Annahmen liegen zu Grunde? Sind die Annahmen realistisch? Was passiert in 6 Monaten, wenn neue Features gebraucht werden (Variabilität und Erweiterbarkeit des Patterns)? Alle diese Dinge sollten in Interviews mit den Domänenexperten erörtert werden. Das resultierende Pattern muss kurz, kompakt und aus vorhanden Entwürfen aufgebaut sein. Es muss im Rang einer Best-Practice sein.

Pattern führen zu einer verbesserten Kommunikation, aber die Auswirkungen sind nur schwer messbar. Ein weiterer interessanter Nebeneffekt beim Pattern-Einsatz ist es, dass vergessene Anforderungen identifiziert werden können. Dies geschieht, wenn die Beschreibung eines Pattern nur teilweise mit den Anforderungen übereinstimmt.

Folgende Tabelle wurde aus dem GOF-Buch übernommen und stellt die möglichen Variabilitäts-Aspekte der darin enthaltenen Design-Patterns in einer Übersicht dar.

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory	Families of product objects
	Builder	How a composite object gets created
	Factory Method	Subclass of object that is instantiated
	Prototype	Class of object that is instantiated
	Singleton	The sole instance of a class
Structural	Adapter	Interface to an object
	Bridge	Implementation of an object
	Decorator	Responsibilities of an object without subclassing
	Facade	Interface to a subsystem
	Flyweight	Storage costs of objects
	Proxy	How an object is accessed; its location
Behavioral	Chain of Responsibility	Object that can fulfill a request
	Command	When and how a request is fulfilled
	Interpreter	Grammar and interpretation of a language
	Iterator	How an aggregate's elements are accessed, traversed
	Mediator	How and which objects interact with each other
	Memento	What private information is stored outside an object, and when
	Observer	Number of objects that depend on another object; how the dependent objects stay up to date
	State	States of an object
	Strategy	An algorithm
	Template Method	Steps of an algorithm
	Visitor	Operations that can be applied to object(s) without changing their class(es)

Tabelle 1: Entwurfsaspekte, die mit Design-Patterns variiert werden können (GOF, Seite 30)

2 Rollenbasierter Entwurf

In diesem Kapitel sollen die Unterschiede zwischen Rollen und Objekten, Rollentypen und Klassen dargestellt werden. Zentrale Punkte sind das Rollen-Mapping auf Klassen und die Rollenmodell-Komposition. Design-Patterns lassen sich übrigens auch als Rollenmodelle ausdrücken, die dann in Klassenmodelle übertragen werden. Zusammengesetzte Design-Patterns lassen sich durch Rollenmodellierung besser verstehen. Rollentypen werden als semantisch nicht-rigide Typen angesehen, geschichtete Frameworks lassen sich als Rollenmodelle ausdrücken. Insgesamt lässt sich der Ansatz des rollenbasierten Entwurfs zur Optimierung von Frameworks und Design-Patterns einsetzen.

REENSKAUG, der an Smalltalk und dem Model-View-Controller-Pattern arbeitete, sagt "UML did it all wrong". Inzwischen bietet UML über kleine Lollipops die Möglichkeit, benötigte und bereitgestellte Interfaces auszudrücken. Dies kann auch dazu verwendet werden, bestimmte Rollen von Objekten zu modellieren.

Der Begriff der **ROLLEN** befindet sich auf dem Objekt-Level in der Objektorientierung. Die Rolle ist eine dynamische Sicht auf ein Objekt. Sie ist veränderlich und stellt einen bestimmten Aspekt oder Service dar.

ROLLENTYPEN hingegen befinden sich auf dem Klassenlevel. Sie sind die Service-Typen eines Objekts, damit auch dynamische Sicht-Typen. Sie können sich dynamisch ändern. Ein Objekt spielt eine Rolle eines Rollentyps über einen bestimmten Zeitraum. Der Rollentyp ist Teil des Protokolls einer Klasse und wird oft als Interface umgesetzt. Er gründet sich auf einer Kollaboration mit einem Partner.

Ein **ROLLENMODELL** ist eine Menge von Objekt-Kollaborationen und wird durch eine Menge von Rollentypen beschrieben. Es ergibt sich eine bedingte Beschreibung für Klassen und Objekt-Kollaborationen.

Das **KLASSEN-ROLLENTYP-DIAGRAMM** ist ein UML-Klassendiagramm, welches durch Ovale in den Klassen, welche Rollen darstellen, augmentiert wird. Es können belie-

big viele aktiv sein. Eingesetzt werden zwischen den Rollen sogenannte Rollen-Constraints.

Die zentrale Modellierungsfrage lautet: Welche Rollen spielt ein Objekt im gegebenen Kontext? Äquivalent ist die Frage nach den Verantwortlichkeiten des Objekts und welchen Zustand es für einen Kontext besitzt. Bei Einsatz der CRC-Karten-Methode (Class-Responsibility-Collaboration) entsprechen die Rollen den Responsibilities. Die Methode eignet sich für Rollen allerdings nur, wenn sehr viele Kollaborationen vorhanden sind.

Beim **ROLLENBASIERTEN ENTWURF** mit Rollenmodellen liegt der Schwerpunkt immer auf den Kollaborationen. Jeder Kollaborateur ist eine Rolle. Dabei teilen die Rollen Klassen in kleinere Stücke. Der Schwerpunkt liegt auf der Kontextabhängigkeit. Man kann auch von der Separierung der Aspekte oder Angelegenheiten (Concerns) sprechen. Die getrennte Wiederverwendungsmöglichkeit des Rollenmodells vom Klassenmodell ist besonders hervorzuheben. Das Vorgehen beim rollenbasierten Entwurf ist wie folgt. Zuerst werden die Rollenmodelle entwickelt. Die getrennt herausgefundenen Rollen werden durch gegenseitige Bedingungen in Verbindung gebracht. Dabei ist wichtig, welches Objekt welche Rolle ausführt. Im zweiten Schritt werden die Rollenmodelle zusammengeführt. Dabei werden auch neue Bedingungen zwischen den Rollen der Modelle hergestellt. Als drittes erfolgt das Mapping von Rollenmodell auf Klassendiagramm. Dabei werden Rollen auf Klassen abgebildet, unter Berücksichtigung der Bedingungen. Hier wird auch vom Weaving gesprochen. Zu beachten ist, dass aus einem Rollenmodell mehrere unterschiedliche Klassendiagramme hervorgehen können. Es existiert eine Variabilität bereits im Modell.

Für Design Patterns ist die Komposition ihrer Rollenmodelle in gleicher Gestalt möglich. Dabei ist zu beachten, dass Klassen statisch sind und Rollen kombinieren, diese wiederum sind dynamisch. Klassenmodelle kombinieren Rollenmodelle, die auch als Dimensionen und Aspekte bezeichnet werden können. Partielle Klassenmodelle mit offenen Rollen erlauben spätere, weitere Komposition. In der modellgetriebenen Architektur (MDA) kann die Rollenmodellierung und das Mapping auf Klassen als zusätzlicher Transformationsschritt verstanden werden. Das Business-

Modell wird auf das Klassenmodell abgebildet. Die Rollen sind dabei plattformunabhängiger als Klassen, da erstere logisch sind und letztere physisch.


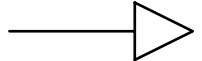
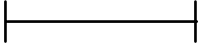
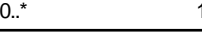
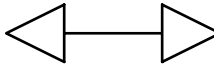
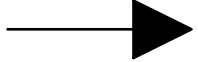
- o **Role-Use:**
Eine Klasse benötigt eine andere Rolle 
- o **Role-Implication:**
Ein Objekt welches eine Rolle spielt, muss auch eine andere spielen 
- o **Role-Prohibition:**
Ausschlussbedingung zwischen zwei Rollen 
- o **Role-Association:**
Ein Objekt, welches eine Rolle spielt, kennt ein Objekt, welches die andere Rolle spielt 
- o **Role-Equivalence:**
Objekt muss beide Rolle spielen, wenn es eine bereits spielt 
- o **Role-Implication-Inheritance:**
Betonung, dass Rolle vererbt werden kann 

Abb. 3: Constraints bei der Rollenmodellierung

Die manuelle Implementierung von Rollen kann über Interfaces erfolgen. Dabei muss der Code allerdings per Hand geschrieben werden, ohne die Möglichkeit der Wiederverwendung. Bei Sprachen mit Mehrfachvererbung werden zwei Schichten von Klassen realisiert, Standardklassen und Rollen. Bei Verwendung von Mixin-Klassen, welche von einigen Programmiersprachen unterstützt werden, sind die Rollen Mixins, die in eine Klasse komponiert werden. Eiffel bietet hier die Include-Inheritance als Sprachkonstrukt. Mit Multi-Bridges wird ein Rollen-Objekt für eine Rolle angelegt, ein Kern aggregiert dann alle zusammen. Der Unterschied zwischen Rollen und Facetten liegt darin, dass eine Faceted-Klasse beliebige viele Rollendimensionen haben kann. Jede Facette entspricht dabei einem Rollentyp. Sie sind voneinander unabhängig, aber statisch. Die Facette bleibt über die gesamte Lebenszeit des Objekts bestehen.

Werden Design-Patterns als Rollenmodelle ausgedrückt, kann man sich auf die Participant-Sektion im GOF-Buch berufen. Schon hier fällt auf, dass die Struktur bei vielen Patterns sehr ähnlich ist. Oft werden optionale Rollen angegeben, da die Beschreibung nicht standardisiert ist. Ein Ingenieur identifiziert und benennt die Rollen im Struktur-Klassendiagramm für das Design Pattern konkret um, was dem Mapping von Rollenmodell auf Klassenmodell gleich kommt.

Das Erstellen von größeren Patterns ist möglich durch die Komposition der Rollenmodelle. Als Beispiel dient dafür das BUREACRCY Pattern, welches für Organisationen mit Baumstrukturen verwendet wird (siehe Abbildung auf dem Cover dieses Dokuments). Weiterhin kann das Model-View-Controller-Pattern durch Rollenmodell-Komposition verstanden werden. In den Unterlagen zur Vorlesung wird dies im Detail belegt.

Das **RIEHLE-GROSS-GESETZ** für zusammengesetzte Design-Patterns besagt: Das Rollenmodell eines zusammengesetzten Design-Patterns besteht aus den Rollenmodellen der beteiligten Design-Patterns.

Aus dem Riehle-Gross-Gesetz lässt sich schlussfolgern, dass komplexe Patterns leicht zerlegbar in Einfachere sind. Durch das Verständnis eines Design-Patterns als Rollenmodell werden auch Varianten ein und desselben leichter vergleichbar. Zum Beispiel lässt sich das konzeptionelle Pattern Template and Hook dadurch erklären. Es wird hier ausgehend von einem Mikro-Pattern, **DERIVED METHOD**, zusätzliche **TEMPLATE METHOD** und **HOOK METHOD** Rollen ergänzt. Bei Template Class wird eine Vererbungshierarchie für den Hook impliziert.

Mit unterschiedlichen Rollenmodellen lassen sich die feinen Unterschiede zwischen Patterns also auch syntaktisch ausdrücken. Der Sinn des Patterns wird aufgeschrieben. Der Entwurf wird expliziter, präziser und formaler. Auch das Pattern Mining kann mit Rollen erfolgen.

Im Vergleich mit **HYPERSLICES**, oder auch Views, welche ein ähnliches Konzept umsetzen, weil sie auch auf anderen Abstraktionen als Klassen und Eigenschaften arbeiten, fällt auf: Rollenmodelle brauchen keine Dimensionen und Schichten, sie

sind damit freier. Das Mergen von Views erfolgt aber genauso wie bei Rollen, bei Rollen sind allerdings auch Constraints möglich.

Der Vergleich mit Facetten zeigt, dass eine Facette ein Produktverband ist und sich mit einem logischen Objekt beschäftigt. Rollenmodelle können mehrere Objekte durchschneiden. Tatsächlich ist die Voraussetzung für die Existenz von Rollen eine Kollaboration, die immer zwischen mindestens zwei Objekten stattfindet. Eine Facette ist wie eine Rolle eines Objekts aus einer endlichen Menge von Facettendimensionen, die über die gesamte Lebenszeit des Objekts hinweg existiert.

Wichtig ist weiterhin die Unterscheidung in **RIGIDE TYPEN** und gegründete, sogenannte **FOUNDED TYPEN**. Für die ersten steht zum Beispiel der Typ "Buch", wenn ein Objekt einen semantisch rigiden Typen hat, kann es nie aufhören diesen zu haben, ohne die Identität zu verlieren. Der Typ gehört zur Klasseninvarianten. HEIDEGGER und KANT würden es "Das Ding an sich" nennen. Semantisch nicht rigide Typen kennzeichnen den dynamischen Zustand eines Objekts. Gegründete Typen, wie zum Beispiel der des "Lesers", sind Typen eines Objekts, welche immer kollaborieren, oder assoziiert sind mit einem anderen Objekt.

Rollentypen sind gegründet und nicht-rigide, Natürliche Typen sind nicht-gründet und semantisch rigide. Sie können für sich alleine stehen.

Rollenmodellierung hat folgende Effekte. Delegierung kann skaliert werden. Standardmäßig werden alle Rollen über eine Klasse gelegt, sie können aber auch separat bleiben. Das feinkörnige Zerlegen von Objekten in so kleine Teile wie möglich bietet später alle Freiheiten für das Zusammensetzen. Geschichtete Frameworks teilen alle Rollen auf Role-Objects auf. Wenn die Schichten zusammengelegt werden sinkt die Variabilität und die Effizienz steigt. Dies folgt aus dem Abnehmen von Delegierungen, Allokationen und Flexibilität. Die Gesetzmäßigkeit wird in Abmann's Law zementiert.

Das **GESETZ VON ABMANN** zur Optimierung von Design Pattern sagt aus: Falls eine Variante für ein Design Pattern gesucht wird, die effizienter ist, untersuche das Rollenmodell und versuche Verschmelzung der Klassen für die Rollen.

Bei der Umsetzung von Rollen-Bedingungen ist zu beachten, dass die Role-Implication sich in zwei Klassen, einer Klasse, oder Ober- und Unterklassen implementieren lässt und somit große Freiheiten bietet. Role-Implication-Inheritance erlaubt allerdings nur Vererbung. Die Wahl von monolithischen oder gesplitteten Objekten ist eine sehr wichtige Entwurfsentscheidung.

Das objektorientierte Framework San Francisco von IBM wurde für Enterprise Resource Planning ausgelegt, scheiterte aber auch trotz Java-Basierung wegen fehlender von Rollenmodellierung. Es wurde nur Einfachvererbung eingesetzt. SAP berücksichtigt in ihrem Framework Laufzeit-Variabilität und Flexibilität. Offene Rollentypen können in Frameworks als Hotspots verstanden werden und erlauben zudem das Zusammenfügen mehrerer Frameworks.

Die Zahl der Laufzeit-Objekte, Referenzen und Fragmentierung ist dabei abhängig vom Mapping. Die Berechnung der physischen Objekte kommt einem Tailoring gleich. Das Klassenmodell ist also nicht die heilige Wahrheit, sondern die Implementierungs-Wahrheit. Tatsächlich existiert eine Variabilität in den übergeordneten Modellen, die zum Beispiel bei der MDA verwaltet wird.

Am 11. Dezember 2007 hat Stefan Hermann seine Programmiersprache ObjectTeams an der TU Dresden vorgestellt. Dort wird ein Rollenkonzept umgesetzt. Die Assoziationen werden in Klassen aufgelöst und Referenzen beim Übergang in die Implementierung. Viele Entwurfsmuster werden hier bereits überflüssig durch die Arbeit mit dieser Sprache. Composite und Chain existieren zwar auch hier als strukturelle Patterns, werden aber vereinfacht, da Mediator und Observer verschwinden. Informationen unter objectteams.org.

3 Frameworks und Produktlinien

Die im vorigen Kapitel beschriebenen Design-Patterns sind Grundlage für das Verständnis von Frameworks. Es handelt sich hierbei um Software-Konstrukte, die für eine allgemeine Funktionalität ausgelegt sind und es erlauben, Spezialisierungen vorzunehmen. Ein Framework ist selbst noch kein eigenständiges Programm und muss an definierten Punkten instantiiert und erweitert werden.

3.1 Framework Variation Patterns

Ziel dieses Kapitels ist es, Framework-Hook-Patterns zu verstehen, welche auf der Variabilität des Template-and-Hook-Konzeptes aufbauen. Es wird mehr zu geschichteten Frameworks erläutert.

Der Zusammenhang mit Design Patterns besteht darin, dass sie zuerst während der Framework-Entwicklung entdeckt wurden. Design Patterns sind somit Bauteile von Frameworks, einige sind verwendbar als Framework-Variations-Punkte für Variabilität, Erweiterbarkeit und damit für Produktlinien.

FRAMEWORK-INSTANTIIERUNG erfolgt dabei über OFFENE ROLLEN (Hotspots) nach den Arbeiten von RIEHLE und GROSS. Frameworks werden als Klassenmodelle mit offenen Rollen modelliert, welche als Hotspots funktionieren. Es handelt sich dabei um Rollentypen, die noch nicht zu Klassen zugeordnet wurden. Die Menge dieser Rollentypen wird auch als "integration role type set" bezeichnet. Die Instantiierung geschieht nun durch die Bindung der Rollentypen an Klassen, unter Berücksichtigung von Constraints. Das Rollenmodell ist also die Verbindung zwischen Framework und dem Client. Im Gegensatz zu den Hotspots existieren auch Frozen-Spots, in Form von gebundenen Rollentypen.

Eine Verschmelzung von Frameworks ist möglich, indem die Rollentypen von einem Framework an die Klassen eines anderen gebunden werden. Wieder müssen

die Constraints dabei berücksichtigt werden. Rollenmodelle sind also die Verbindung zwischen Frameworks, oder Framework-Schichten. Diese einfache Rollenbindung kann auch noch elaboriert werden.

Prees **FRAMEWORK-HOOK-PATTERNS** beschreiben die Rollen von Klassen am Rand des Frameworks. Es handelt sich um Variations-Punkte mit dem Template-and-Hook-Pattern (T&H) als konzeptionelles Pattern. Pree nennt diese Meta-Patterns, Prof. Aßmann bezeichnet sie als Rollenmodelle. Es werden die Arbeiten von Pree mit denen von Riehle und Gross kombiniert. Das T&H-Pattern besteht im Rollenmodell aus zwei Rollen, einem Template in Role-Use-Beziehung mit einem Hook. Es wird ein fester von einem flexiblen Teil getrennt. Durch Overlaying mit einem anderen Pattern kann zum Beispiel das Observer-Pattern damit nachempfunden werden. Dadurch können mehr Constraints formal angegeben werden und somit Informationen über das Pattern explizit gemacht werden. Pree hat 7 Framework-Hook-Patterns identifiziert, mit Riehles Open-Role-Hook. Das Template gehört dabei immer zum Framework und der Hook zur Anwendung.

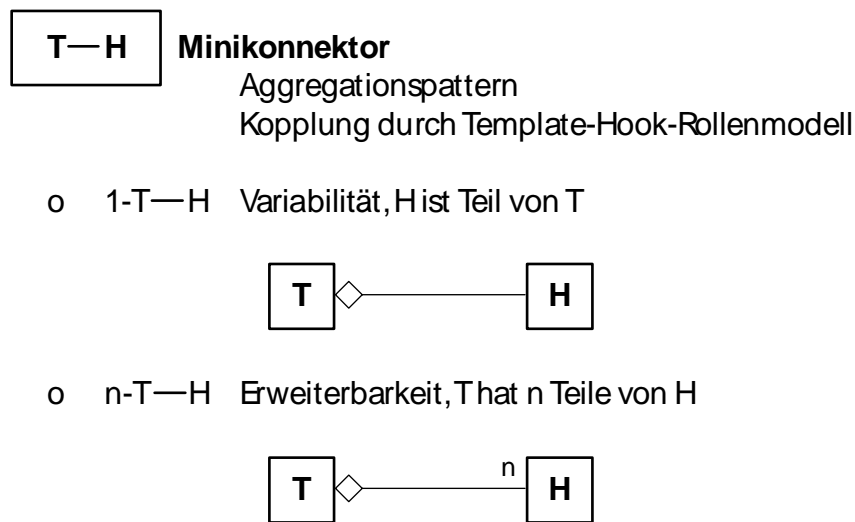
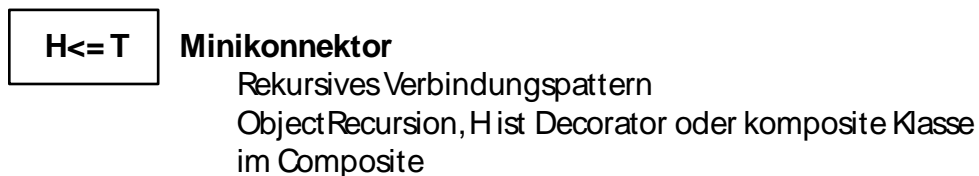


Abb. 4: Der T—H Minikonnektor.

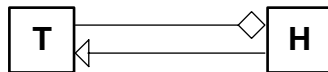
Die Template-Method-Rolle definiert dabei ein Template, welches die Hook-Method aufruft. Framework-Hook-Patterns bieten im allgemeinen also Design-Patterns am Rand des Frameworks an und legen das T&H-Rollenmodell darüber.

T&H META-PATTERN + STANDARD DESIGN PATTERN = **FRAMEWORK-HOOK-PATTERN**

Der T—H Minikonnektor ist ausdrückbar mit den Patterns Template Class und Dimensional Class Hierarchies in der 1-T—H Variante. Für die n-T—H Variante kann ebenfalls Dimensional Class Hierarchies verwendet werden. Zudem Observer, Bridge für geschichtete Frameworks. Der Minikonnektor steht für Blackbox-Frameworks, da die Hauptbeziehung Delegation ist und somit das Framework unangetastet bleibt.



- o 1-H<=T Decorator, deep list extension



- o n-H<=T Composite, deep graph extension

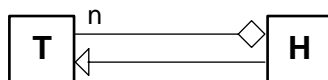


Abb. 5: Der H<=T Minikonnektor.

Der H<=T Minikonnektor lässt die Hook-Klasse von der Template-Klasse erben, das Template ist zudem Teil vom Hook. Das Pattern befindet sich zwischen Black-Box und White-Box, da die Template-Klasse im Framework von der Hook-Klasse in der Anwendung abgeleitet und aufgerufen wird.

Das Unifikations-Pattern mit dem TH Minikonnektor vereinigt die Template- und Hook-Klasse. Chain Of Responsibility kann als 1-TH umgesetzt werden. Rekursion erfolgt dabei auf der abstrakten Superklasse. Jede Klasse im Vererbungsbaum weiß dadurch, dass sie eine andere versteckt, anders als bei Object Recursion. Dies kann bei Event-Handlern eingesetzt werden.

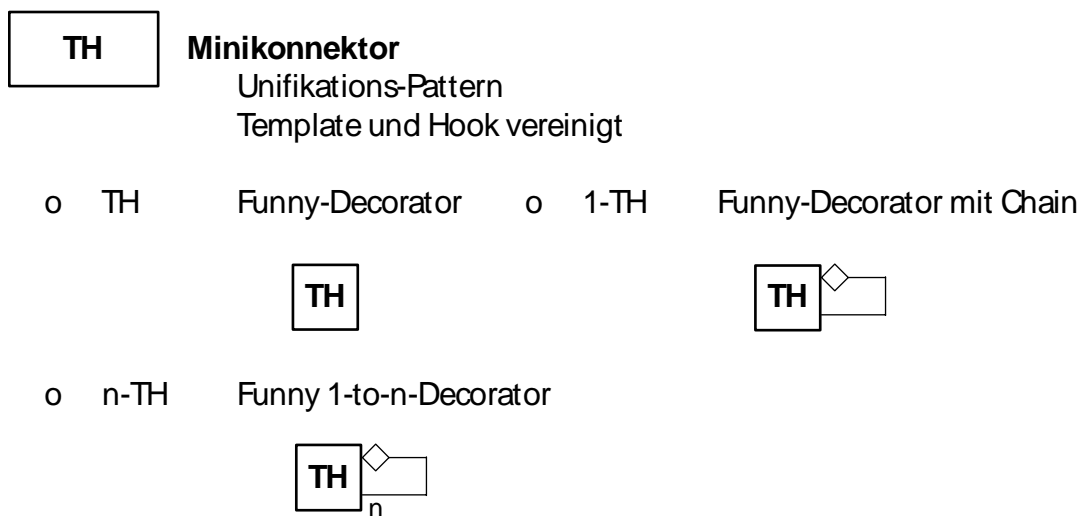


Abb. 6: Der TH-Minikonnektor.

Der Austausch von Template-Methoden und Hooks kann nur zusammen erfolgen. Die Template-Methoden sind nicht abstrakt und referenzieren konkrete Hooks. Das Verschmelzen von Rollentypen wie bei dem TH Minikonnektor ergibt schnellere, aber dafür weniger flexible Anwendungen. Eine monolithische Laufzeit-Struktur resultiert daraus, sowie eine enge Zusammenarbeit zwischen den Layern. Es kann von einem "degenerierten Decorator" gesprochen werden.

Das H<T Vererbungspattern ist nur für Whitebox-Frameworks geeignet. Es erfolgt das Öffnen der Oberklasse und Modifikationen. Insgesamt wird mit allen Minikonnectoren eine präzise Notation erreicht.

PREES ERSTES GESETZ für Framework-Instantiierung: Basiert ein Framework-Hook-Pattern auf einem Design-Pattern für Variabilität, so wird das Framework variiert, nicht erweitert.

PREES ZWEITES GESETZ für Framework-Instantiierung: Basiert ein Framework-Hook-Pattern auf einem Erweiterbarkeits-Pattern, so wird das Framework erweitert, aber nicht variiert.

Aus den beiden Gesetzen folgt, dass wenn ein Framework nicht auf eine feste Menge von Variationen beschränkt werden soll, Erweiterbarkeits-Patterns genutzt werden müssen.

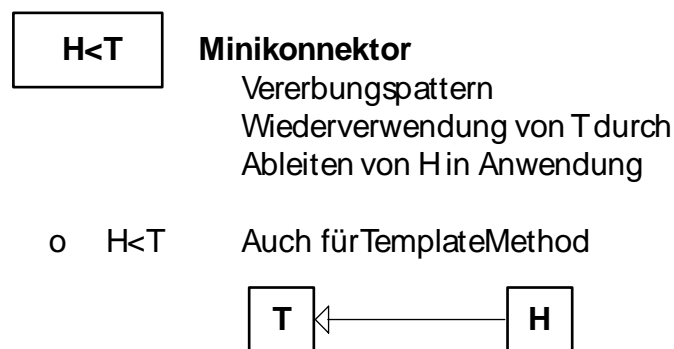


Abb. 7: Der H<T Minikonnektor

Es existiert ein sehr großer Markt für Framework-Instantiierung. Beispiele sind das SAP-Framework und Gebos von der Züllighoven-Gruppe. Letzteres kommt bei über 1000 Volksbanken und Sparkassen zum Einsatz. Viele Firmen verfügen über In-House-Ingenieure, die Extensions und Extension-Points für sich selbst entwerfen. Externe Plug-Ins werden ebenso produziert. Software-Produkt-Ingenieure haben ein anderes Profil als Framework-Ingenieure.

SAP bietet für die Lightweight-Instantiierung des Frameworks das sogenannte "Business by Design", wodurch es einfacher für die Solution Manager Implementation (SMI) wird, In-House-Hosting und Instantiierung zu gewährleisten.

3.2 Framework Extension Patterns

Da Erweiterbarkeit für Frameworks obligatorisch ist, werden im folgenden die entsprechenden Pattern besprochen, die dies gewährleisten. Der Aufbau von geschichteten Frameworks resultiert daraus.

Das **EXTENSION OBJECTS** Pattern behandelt komplexe Objekte mit fakultativen Erweiterungen, die allerdings von einem Client hinzugefügt und entfernt werden. Für lediglich eine Erweiterung muss keine Benennung erfolgen, ansonsten wird für mehrere ein Dictionary oder Map verwaltet. Hier werden Namen auf die entsprechenden Extension-Objects abgebildet. Der Vorteil dieses Patterns liegt darin, dass Objekte in einfache Teile aufgeteilt werden können, optionale Rollen sind modellierbar. Erweiterungen können außerdem dynamisch hinzugefügt werden. Der Nachteil besteht darin, dass die Clients für das Management verantwortlich sind und dadurch selbst komplexer werden. Extension-Objects führen außerdem zur Objekt-Schizophrenie. Dies bezeichnet den Umstand, dass ein Teil eines Objekts nicht die Identität des gesamten, logischen Objekts kennt.

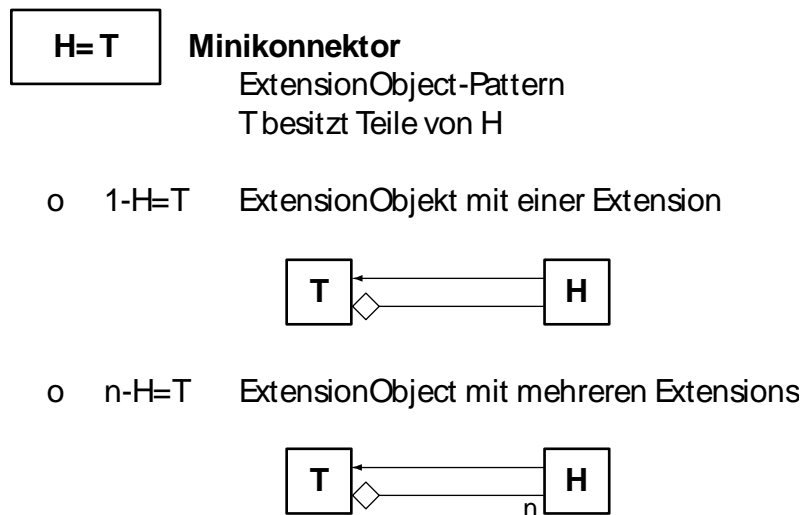


Abb. 8: Der H=T Minikonnektor

Das Extension-Objects-Pattern kann auf das **ROLE OBJECT** Pattern generalisiert werden. Dies ist dann der Fall, wenn mehrere Objekte die gleichen Erweiterungen als Rollen spielen. Ein neuer H=T Minikonnektor wird eingeführt.

Ein Beispiel für Frameworks mit Schichten ist das bereits erwähnte **GEBOS-SYSTEM**, welches Grundlage von Bankanwendungen für 450 Banken ist. Es existieren Application-Layer, Business-Section-Layer, Business-Domain-Layer. Zudem gibt es den Desktop-Layer und den Technical-Kernel-Layer. Die Kopplung zwischen dem Framework und Anwendungssystemen wurde minimiert. Verschiedene Facetten für Produkte, Business-Domänen werden modelliert. Das Role-Object-Pattern wird hierfür eingesetzt, wodurch die Kopplung zwischen den Schichten minimiert wird. Eine Produktlinie mit Variation wird für die Volks-Raiffeisenbanken, sowie die Sparkassen-Gruppe mit ihren getrennten IT-Abteilungen umgesetzt. Das gesamte Konzept ist auch für andere Domänen übertragbar.

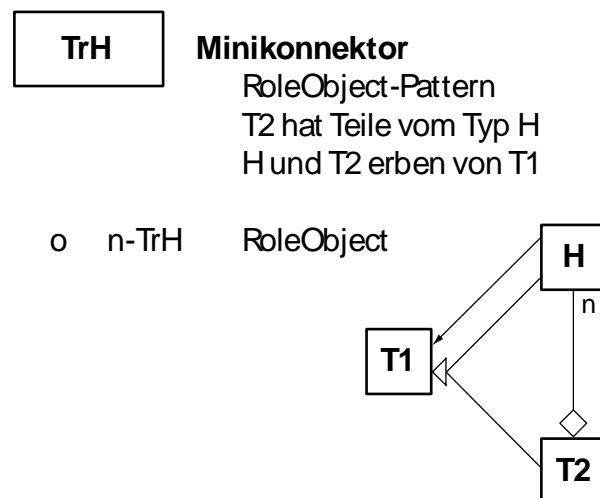


Abb. 9: Der TrH Minikonnektor

Das verwendete Role-Object-Pattern dient für Erweiterbarkeit und Variabilität und realisiert einen Dispatch über alle Layer in der Anwendung. Es existiert eine gemeinsame Abstraktion von Kern und Rollen. Die Umsetzung kann in einer ersten Variante durch einen Stack oder Decorator erfolgen, wodurch eine Kette von Rol-

len entsteht. Eine zweite Variante ist die Umsetzung in einen Kern mit seinen Rollen, bei denen eine Rolle nicht die andere kennt. Dabei ändern sich die nicht-rigid Typen, also die Rollen bei Zustandsänderungen. Bei der Variante mit Decorator kann dies über einen Mediator erfolgen. Eine Kernschicht in Form einer Blackbox mit Kernobjekten wird mit Delegierten-Layern erweitert. Jedes konzeptionelle Objekt wird über alle Schichten verteilt. Dabei gehören der Kern und seine Rollen konzeptionell zusammen, die Rollen bieten je eine Sicht auf das konzeptionelle Objekt und können sich ändern. Der entstehende TrH-Minikonnektor beruht auf dem $n-H \leq T$ -Minikonnektor, also der n -Object-Recursion und Composite.

Rollenobjekte können auch so verstanden werden, dass sie ein Kernobjekt dekorieren. Es handelt sich aber um keinen Decorator, da es zu einem Kern mehrere Rollen gibt. Im Vergleich mit Vererbung ist festzustellen, dass einfache Vererbung nur eine Instanz einer Subklasse zu einer Zeit bietet. Durch Polymorphie kann sich diese ändern. Beim Role-Object-Pattern gibt es viele Instanzen, die sich alle ändern können. Man kann von Rollen-Polymorphismus sprechen. Dabei haben nur die Änderungen in der Basisschicht Auswirkungen auf die anderen Schichten. Dies bedeutet, dass eben nicht nur Variabilität, sondern auch unvorhergesehene statische und dynamische Erweiterbarkeit möglich ist. Dies erfolgt über das Einführen neuer Layer. Wird zwischen den Schichten Vererbung eingeführt, so erschwert sich der Austausch einer einzelnen Schicht. Dies kann auch ein Grund für das Scheitern des San-Francisco-Frameworks von IBM gewesen sein. Stattdessen sollten Minikonnectoren über Rollen mit Template und Hook eingesetzt werden.

Bei der *-Bridge gibt es keine Vererbung zwischen dem konzeptionellen Objekt und Kern, sowie Rollen. Beim Role-Object-Pattern ist Vererbung auch zwischen den Rollen-Objekten möglich. Eine Unabhängigkeit wird nicht erzwungen. Es erfolgt auch keine Zerschneidung eines konzeptionellen Objekts mit Facetten, sondern die Realisierung des konzeptionellen Objekts als primärer Dimension, dem Kern und sekundären Dimensionen, den Rollen-Objekten. Facetten sind unabhängiger, die Schichten müssen sich nicht kennen bei der Umsetzung mit Multi-Bridges.

Daraus ergibt sich der Dispatch auf allen Schichten. Dies ist nativ in einer Programmiersprache nur möglich, wenn Multi-Methoden unterstützt werden. Dies ist

zum Beispiel bei CLOS, oder Multi-Java der Fall. Mit Generics kann zudem eine Hierarchie durch Generic-Expansion flacher gemacht werden.

Das GENVOCA Pattern von BATORY und SMARAGDAKIS benutzt das Mixin-Konzept. Es handelt sich dabei um Klassenerweiterungen mit Klassen-Fragmenten, die durch die Superklasse parametrisiert werden. Ein Mixin wird einer Klasse aufgesetzt und damit Felder hinzugefügt. Auf diese Weise sind auch Rollen und Facetten implementierbar. Die Verschachtelung mehrerer dieser Mixin-Parametrisierungen macht das GenVoca-Pattern aus. Nun können Produktlinien umgesetzt werden, indem verschiedene Varianten für eine Abstraktionsschicht, die mit Rollenschichten korrespondieren, angeboten werden. Da Mixins eingebettet werden, haben sie eine Identität immer für das gesamte logische Objekt, es kommt zu keiner Objekt-Schizophrenie und effizienterem Laufzeit-Verhalten. Dies folgt aus Aßmanns Gesetz, da die hier Rollen auf ein Objekt vereinigt werden. Die Rollenkomposition ist also statisch und lässt keine dynamische Variation zu. Jede Abstraktions- oder Mixin-Schicht steht dabei für Variabilität, neue Schichten für Erweiterbarkeit. Es erfolgt der Austausch des gesamten Layers, um ein anderes Produkt zu schaffen. Es ergibt sich eine Matrix mit variierbaren Rollen. Auf der vertikalen Achse befinden sich die Objekte, horizontal die verschiedenen Varianten der Rollen-Schichten. Große Produktlinien sind umsetzbar. Implementierung mit Inner-Classes erlaubt die getrennte Parametrisierung aller Rollen-Mixins und nicht der gesamten Schicht als Ganzes.

Mixins sind nicht in Standardsprachen enthalten. Für Sprachen mit vollem Genericity-Konzept wie C++ kann ein Workaround verwendet werden. Eine Layerkonfiguration wird dabei in einer Klasse über Template-Programming definiert.

Geschichtete Mixin-Frameworks grenzen sich zu geschichteten Rollen-Objekt-Frameworks wie folgt ab. Es handelt sich um das gleiche Konzept für Variabilität und Erweiterbarkeit, der Unterschied liegt nur im Minikonnektor, der einmal das Role-Object-Pattern verwendet, das andere Mal das GenVoca-Pattern.

Geschichtete Frameworks stehen für den Traum von einem einzigen Entwurf abzugehen. Zum Beispiel im Semantic-Web sollen Aspekte getrennt werden. Die Architektur wird in ein austauschbares Komponentenmodell gegossen. Die Anwendung selbst besteht nur aus Benutzer- und Anwendungs-Constraints. Die Kern-

konzepte werden durch eine Komponentensprache realisiert. Der Austausch von Ontologie-Sprachen, Architekturstilen und Benutzeranforderungen wird so zum Kinderspiel.

3.3 Tools and Materials Pattern-Language

Die Intention dafür, Anwendungen anhand dieser Pattern-Language zu gestalten liegt darin, dass herkömmliche Programme den Nutzer oft zu sehr einengen. Handlungsspielräume und Freiheiten in der Abfolge der Tätigkeiten sind oft nicht vorhanden. Mit dem **TOOLS AND MATERIAL** Ansatz (TAM) sollen sich Menschen kompetent fühlen, bestimmte Aufgaben durchzuführen. Das bedeutet, es wird kein fester Workflow vorgegeben, sondern flexible Arrangements mit Werkzeugen angeboten. Der Nutzer entscheidet so über die Organisation der Arbeit und der Umgebung. Es kann so inkrementelle Arbeit realisiert werden.

Man kann TAM als detailliertere Umsetzung der 3-Tier-Architektur ansehen. Letztere ist noch zu grobkörnig für interaktive Anwendungen. Die zentrale Metapher sind das Werkzeug und das Material aus dem Handwerk. Das bedeutet Menschen nutzen Werkzeuge als Mittel der Arbeit. Zudem arbeiten Menschen mit Material. Die Umgebung ist der handwerkliche Arbeitsplatz.

MATERIAL sind passive Entitäten, auf die kein direkter Zugriff möglich ist. Es gibt zum einen Werte, zum Beispiel Datum und Geld. Diese Werte sind zeitlos und haben keine Position und Identität. Ihre Gleichheit wird nach Wert festgestellt und sie sind domänenspezifisch und nicht änderbar. Zum anderen können auch Objekte Material sein. Diese sind zeitbehaftet, haben eine Position und konkrete Identität. Hier wird die Gleichheit anhand des Namens bestimmt. Geteilter Zugriff kann über Referenzen geschehen, Änderungen können insofern an Objekten vorgenommen werden, dass die Identität bestehen bleibt.

WERKZEUGE sind aktive Entitäten. Sie enthalten Wissen darüber, wie effizient mit bestimmten Material gearbeitet werden kann. Werkzeuge können als Sicht auf ein Material bezeichnet werden und bieten Feedback für den Nutzer über den derzeitigen Zustand. Im besten Fall sind Werkzeuge transparent und leichtgewichtig, ohne dabei komplett zu verschwinden. Sie können außerdem strukturiert werden. Viele Werkzeuge können auf einem Material eingesetzt werden. Werkzeug ist aktiv und hat Kontrolle, das Material liefert die Daten.

Als letztes Element im TAM-Ansatz kann die **ARBEITSUMGEBUNG** angesehen werden. Sie dient zur Organisation von Werkzeugen, Materialien und Kollaborationen. Es können Werkzeug-Factories und Material-Factories angeboten werden. Die Arbeitsumgebung ist Workshop und Desktop für die Planung und Arbeit.

Das **TOOL MATERIAL COLLABORATION** Pattern legt die Beziehung zwischen Werkzeug und Material im beiderseitigen Kontext fest. Das Werkzeug hat dabei eine Sicht auf das Material. Diese Rolle eines Materials ist als abstraktes Interface sichtbar für das Werkzeug und definiert notwendige Operationen. Meist folgen die Namen der Rollen dem Namensschema von "Listable", oder auch "Editable". Die Implementierung ist neben Interfaces auch als Object Adapter, Decorator, Role Object, oder über das GenVoca-Pattern möglich. Es entsteht eine Tool-Schicht, eine Tool-Material-Collaboration-Schicht und die Material-Schicht.

Die Komposition von Tool- und Material-Frameworks funktioniert über Rollenkomposition, nachdem Tool- und Material-Schichten als Frameworks modelliert wurden.

Werkzeugkonstruktion ermöglicht die Erzeugung strukturierter Werkzeuge. Diese können atomar, zusammengefügt, oder rekursiv zusammengefügt sein. Letzteres zum Beispiel unter Nutzung des Composite-Patterns. Unterwerkzeuge arbeiten dabei auf eigenem Material oder mit weniger komplexen Rollen auf dem gleichen wie das Oberwerkzeug. Sie sehen nur einfache Rollen und erfüllen damit die Grundsätze des Information-Hiding. Die Trennung der Werkzeuge in Funktion und Interaktion ähnelt der Trennung von Benutzerschnittstelle und Anwendungslogik im 3-Tier-Ansatz. Der **FUNCTIONAL PART** eines Werkzeugs kümmert sich um die Manipulation des Materials und den Zugriff über die Material-Rollen. Der **INTERACTION PART** reagiert auf Benutzereingaben. Falls möglich erfolgt dies ohne einen bestimmten Modus und kann somit unabhängig ersetzt werden. Es handelt sich hierbei um eine Verfeinerung des Model-View-Controller-Patterns. Das Werkzeug enthält damit eine View (Interaction Part), Controller (Functional Part), Managing Part des Modells (Functional Part des Werkzeugs, Materialzugriff, Material). Die Implementierung erfolgt mit einem Observer zwischen Interaction Part und Function Part, der Interaction Part überwacht dabei den Function Part. Zudem wird ein Observer zwischen Subtools und Supertools eingesetzt, das Supertool benachrichtigt dabei

die Subtools über Änderungen. Wird in der Implementierung Function Part und Interaction Part verschmolzen wird die Erweiterbarkeit vernichtet, aber das Laufzeitverhalten verbessert. Es kann mit hardgecodeten Aufrufen statt mit Events gearbeitet werden.

In der Umgebung leben nun die Werkzeuge mit ihren Koordinatoren und Materialien mit Administratoren. Die Umgebung ist verantwortlich für die Initialisierung aller Elemente, der Anzeige und dem Warten auf einen Werkzeugaufruf. Der **WERKZEUGKOORDINATOR** gruppiert Werkzeuge und ihr Material. Es handelt sich dabei um ein globales Objekt mit Tool-Material-Dictionary und einer Werkzeug-Factory. Außerdem existiert ein **CONSTRAINED MATERIAL CONTAINER**, der eine bestimmte Strategy verfolgt, um Abhängigkeiten zwischen Materialien anzugeben. Ein **AUTOMATON** ist ein Werkzeug für automatisierte, wiederholte Aufgaben, also eine Art Makro-Werkzeug, welches im Hintergrund laufen kann. Es kapselt automatisierten Workflow, zum Beispiel Produktion und Speicherung. Dies wird mit State-Charts, Aktivitätsdiagrammen oder Makro-Kommandos beschrieben.

TAM kann als Variante von geschichteten Frameworks angesehen werden. Es gibt dabei unterschiedliche Minikonnektoren zwischen den Schichten.

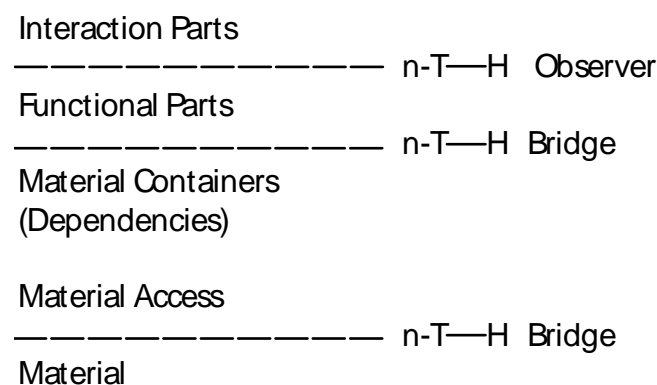


Abb. 10: TAM als geschichtetes Framework

3.4 Eclipse und Framework Extension-Languages

Eclipse ist seit der Version 3 als Menge von Frameworks anzusehen. Diese dienen dazu, IDE-Anwendungen, IDE selbst, GUI-Anwendungen und Rich-Thin-Clients zu entwickeln. Eclipse stapelt Frameworks.

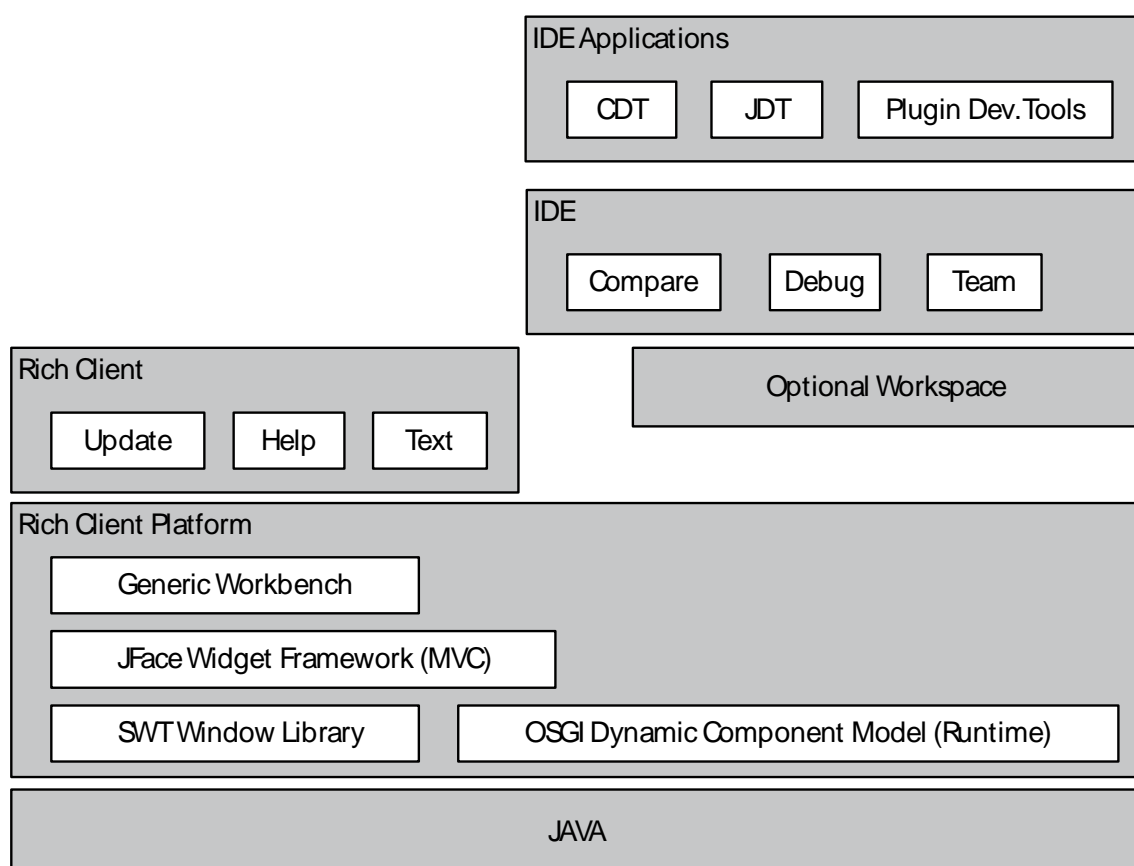


Abb. 11: So stapelt Eclipse Frameworks

Die Besonderheit an Eclipse ist die Möglichkeit Plugins und Extension-Points zu entwickeln und zu definieren. Alle Frameworks besitzen Hooks, die in Eclipse Extension-Points genannt werden. Diese sind kein Konzept für Variabilität, sondern

für Erweiterbarkeit. Extension-Points können Klassen, Menüs, Properties, oder auch Classpath-Einträge sein. Plugins werden dynamisch aus einem bestimmten Verzeichnis geladen. Für die Plugin-Klassen gibt es eine `plugin.xml` als Manifest-Datei. Plugins erweitern die Klasse `Plugin` oder `AbstractUIPlugin`. Die Generic Workbench strukturiert und organisiert die GUI einer Rich-Client-Plattform-Anwendung. Die sprachenkontrollierte Framework-Erweiterung ersetzt die Framework-Hook-Patterns in Eclipse. Ein Core-Interpreter interpretiert die XML-Dateien, um an Extension-Points zu erweitern. Eclipse stellt dafür eine domänenspezifische Sprache für Extension-Points und das Binding bereit. Die bereits definierten Framework-Hook-Patterns können als kleine Sprache für Framework-Extension angesehen werden. Sie sollten auch in Logik ausdrückbar sein und so formal Constraints für Variabilität und Erweiterbarkeit beschreiben.

3.5 SAP R/3 Framework

Die Firma SAP entwickelte ihr Framework für Business-Software in der Version R/2 (Release 2) noch als Mainframe-Lösung. Derzeit ist R/3 immer noch Industriestandard und arbeitet mit dem neuen MySAP und Business-by-Design zusammen. Die neuen Entwicklungen von SAP zielen ungefähr seit dem Jahr 2000 auf Webbasierung ab. Das R/3 ist eine 3-Tier-Architektur für Client und Server. Zur Programmierung wird die hauseigene ABAP-Sprache bereitgestellt. Damit kann auch die Datenbank einfach genutzt werden. Workflow-Spezifikationen werden durch ereignisgesteuerte Prozessketten ausgedrückt. Mit sogenannten Business-Blueprints werden vorgefertigte Prozess für die Installation beim Kunden bereitgestellt. In der Zukunft wird auf SAP NetWeaver gesetzt, der voll Java- und Web-basiert ist.

R/3 verfügt über eine Vielzahl von Modulen. **INTERNAS** mit Financials (FI), Controlling (CO), Anlagenwirtschaft (AM), Projektssystem (PS). **MISCALLENEOUS** mit Office&Communication (OC), Branchenlösungen (IS). **CORE** mit Distribution (SD), Material Management (MM), Production Planning (PP), Quality Management (QM), Instandhaltung (PM) und **HUMAN RESOURCES** (HR). Das System ist intern nicht sehr modern. Jedes Modul kostet und muss adaptiert werden für die spezifischen Geschäftsregeln und Workflows.

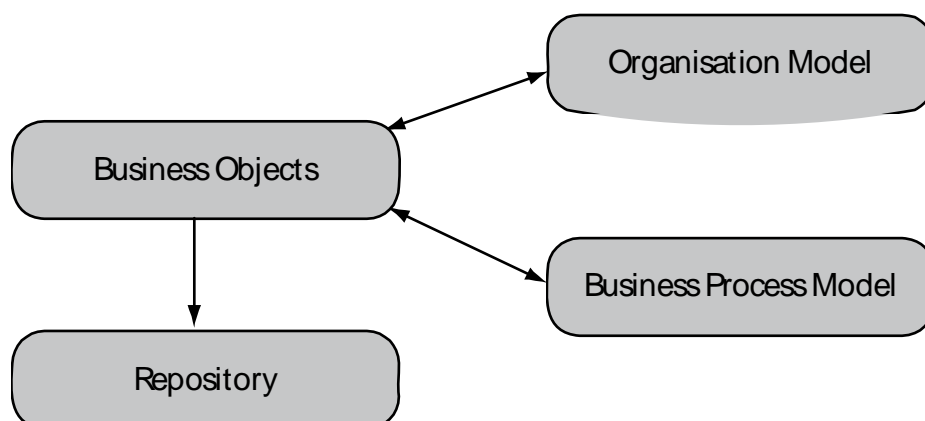


Abb. 12: Anpassung des SAP R/3 Frameworks.

Die ereignisgesteuerten Prozessketten (EPK) werden für die Modellierung von Geschäftsabläufen genutzt. Es handelt sich dabei um einfache Algebra, in der Operatoren auf Ereignissen arbeiten. Prozess liefern dabei Ergebnisse an Operatoren und Ereignisse. Auch die Weitergabe an andere Workflows ist möglich. R/3 ist damit ein Workflow-Framework für dessen Instantiierung EPK-Spezifizierungstemplates eingesetzt werden. Dort werden Gemeinsamkeiten, Erweiterbarkeit (Variabilität) der Workflow-Sprache festgelegt.

Das Repository von R/3 nimmt eine Verteilung der Daten über Application Link Enabling (ALE) vor. Dies ist eine intelligente Middleware-Schicht, durch die ein Repository auf mehreren Datenbanken betrieben werden kann. Die Komplexität der Datenbank wird dabei von ALE versteckt. Es dient zur Speicherung von Business-Objekten mit Interfaces (Business Application Interfaces, kurz BAPI). Die Umsetzung erfolgt mit geschichteten Objekten, intern existiert ein ganzes Business-Object-Model. Die Implementierung ähnelt Role-Objects und Mixins, ist aber einfacher modelliert. Die Menge von Business Objects ist selbst ein geschichtetes Framework.

Das Engineering von SAP-Anwendungen geschieht durch einen Geschäftsprozess-Ingenieur. Geschäftsanwendungen werden durch den Enterprise Integration Server angeschlossen. Dieser verbindet R/3 unter anderem mit Legacy-Systemen, Office-Software, Data-Warehouses und Java.

Die Modellierung von Geschäftsprozessen für die Automatisierung (Billing, Shipping, CRM) ist ein sehr großer Markt. Als Quintessenz sollte mitgenommen werden, dass eine Prozess-Spezifikations-Sprache notwendig ist für ein Prozess-Framework wie SAP R/3. Neben der Lösung mit den ereignisgesteuerten Prozessketten sind auch Petri-Netze (YAFL) dafür einsetzbar, abgestützt auf einer entsprechenden Prozess-Ausführungs-Engine.

Auch für das SAP-Framework ist die Unterscheidung von unvorhergesehenen und expliziten Extension-Points wichtig. Explizite Erweiterungspunkte werden durch die Qualität des Software-Engineerings erreicht und bieten stabile Erweiterbarkeit. Für das Implizite dient die Aspektorientierung.

3.6 San Francisco Framework für Business Applications

Das San-Francisco-Framework wurde von IBM von 1995 bis 1997 entwickelt, mit dem Ziel ein geschichtetes Framework zu schaffen. Verteilte Anwendungen, basierend auf geschäftsspezifischen Design-Patterns sollten realisierbar sein. Ziel war die Flexibilität, die mit objektorientierter Technologie (Java) erreicht werden sollte. Außerdem Wiederverwendbarkeit, Isolation der zu Grunde liegenden Technologie und Konzentration auf den Kern. Eine schnelle und qualitativ hochwertige Anpassung für alle Geschäftsanwendungen, sowie Integrationsmöglichkeiten für andere Sprachen standen ebenfalls auf der Agenda.

Die Architektur besteht aus 3 Schichten, Foundation (mit Infrastruktur und Services), Common-Business-Objects und Core-Business-Processes.

Folgende Techniken finden in San-Francisco Verwendung. Es gibt vordefinierte Business-Objekte. Außerdem ein Komponentenmodell mit sogenannten **USER DEFINED ENTITIES**. Es handelt sich dabei um Material, welches gegebenenfalls auch persistent ist. Es kann mit Konstruktoren, Getter- und Setter-Klassen usw. ausgestattet werden. Globale Handlers werden durch Factories erstellt und in Container abgelegt, ähnlich wie bei Java Entity Beans. Geschäftsprozesse werden über **COMMON FUNCTION FINANCIAL INTERFACE (CFFI)** bestimmt. Hier werden gemeinsame Funktionalitäten definiert, die untereinander genutzt werden. Warehouse-Management fällt darunter, aber auch Order-Management, Accounts und der General Ledger für Journaling.

Erweiterung von San-Francisco funktioniert über die Markierung von Klassen als Extension-Points mittels einer bestimmten Namenskonvention.

`E<number>_<name>`. Business-Objekte können durch Vererbung im Sinne eines White-Box-Frameworks erweitert werden. Unterklassen von der Klasse Property-Container können über ein spezielles Design-Pattern neue Attribute dynamisch, ohne Neukompilation zu Objekten hinzufügen. Der Zugriff auf die Attribute erfolgt dann über eine Hash-Tabelle. Policy-Klassen dienen zum Umsetzen von Geschäftsregeln. Dabei wird Strategy als Extension-Point genutzt und Chain Of Responsibili-

ty um mehrere Policies zu verknüpfen. Mit Composite können Policies mit höherer Priorität eingefügt werden. Durch dynamische Referenzen ermöglicht das Framework die Erweiterung von Wertebereichen.

Lebenszyklus oder Workflow wird durch erweiterbare Statecharts modelliert. Dafür gibt es eine Entscheidungstabelle mit Zustandsübergängen. Die Zeilen enthalten die Bedingungen und Aktionen und ändern den Zustand des jeweiligen Prozesses. Eine dynamische Erweiterung mit neuen Pfaden ist durch Modifikation der Tabelle möglich. Entsprechende Aktionen können definiert werden, die von einer Transition ausgelöst werden. Auch das Löschen von Pfaden ist somit möglich. Statecharts sind endliche Automaten und damit regulär und entscheidbar. Beim Hinzufügen von Sub-Statecharts ist die Frage, ob die Berechenbarkeit gestört wird.

In San-Francisco werden viele spezielle Design-Patterns benutzt. Mit **CLASS REPLACEMENT** kann das Verhalten geändert werden, ohne Klassen- und Anwendungslogik zu ändern. Eine Super-Factory erstellt spezialisierte Factories für spezielle Klassen. Der schon erwähnte **PROPERTY CONTAINER** ermöglicht die dynamische Erweiterung einer Geschäftsobjekts-Instanz um neue Attribute. Es muss nur das Interface implementiert werden, damit neue Properties aggregiert werden können. **BUSINESS PROCESS COMMAND** erlaubt es, ein logisches Objekt durch mehrere physische Objekte zu implementieren, mit dem Ziel einen Geschäftsprozess zu fahren. Command-Pattern kapselt dabei den Prozess wie Facade. Mit **SIMPLE POLICY** werden Geschäftsregeln als Menge von Methoden im Objekt austauschbar gemacht. Dies geschieht durch eine Delegation der Auswertung der Regeln an ein Strategy-Objekt. Die **CHAIN OF RESPONSIBILITY POLICY** modelliert Geschäftsregeln als Kette, somit sind viele Regeln vorhanden, die dynamisch ausgetauscht werden können. Es handelt sich um ein 1-TH-Minikonnektor, kombiniert mit Strategy. Es findet eine Suche nach der passenden Regel für den momentanen Zustand statt.

Für ein Framework sind die Erweiterbarkeits-Mechanismen der wichtigste Teil. Es sollten stets **Pre-Patterns** statt einfacher Vererbung genutzt werden. Die Objekt-orientierte Modellierung ist laut Stefan Hermann außerdem "ill" und sollte von Rollenmodellierung abgelöst werden. Perl 6 verfügt bereits über Rollen. Dies ist eine Gefahr, da Perl damit einen Vorteil hat. Und niemand mag Perl.

3.7 Framework-Dokumentation

Die Dokumentation von Framework ist wichtig, da das Verständnis sehr schwierig ist. Eine gute Dokumentation hilft bei unbekannten Mappings von Domänenkonzepten auf Klassen, Fragen zur Framework-Funktionalität und Architektur- und Integritäts-Verständnis. Außerdem bei technischen Problemen, zum Beispiel der Wahl der Plattform.

Mit dem **PYRAMIDEN PRINZIP** für die Dokumentation wird das Framework auf unterschiedlichen Abstraktionsgraden beschrieben. Es ergibt sich außerdem eine reduzierbare Struktur für die Dokumentation. Auf dem ersten Level wird die **FRAMEWORK SELECTION** in Form einer Kurzbeschreibung erleichtert. Dazu kommt ein Fact-Sheet mit Problem- und Lösungsbeschreibung, sowie Beispiele und Aufzählung verwandter Frameworks. Level 2 beschreibt **STANDARD USAGE**. Es werden hier Anwendungsmuster (Use-Cases) für das Framework angeboten. Darin enthalten sind unter anderem ein Kontext, Problem und entsprechende Instantiierung des Frameworks. Level 3 bietet letztendlich Informationen über das **DETAILED DESIGN**. Hier wird das Framework durch die verwendeten Design-Patterns am Rand dokumentiert, also die Framework-Hook-Patterns. Dazu kommt ein Glossar und eine Suchmöglichkeit.

Um gute Framework-Dokumentation zu realisieren bietet sich das **ELUCIDATIVE PROGRAMMING** an. Es wird hier eine Implementierungsdokumentation erstellt, die mit dem Quellcode verlinkt ist. Durch Hypertext wird die Linearisierung aufgebrochen. Am Besten wird auf zwei Bildschirmen gearbeitet, auf dem einen Code, auf dem anderen Dokumentation.

Mit einem Tutorial-Generation-Environment erfolgt während der Entwicklung die Verlinkung des Codes mit der Dokumentation. Automatisierte Updates vereinfachen den Dokumentationsprozess. Ein Beispiel-System ist das **DEVELOPMENT ENVIRONMENT FOR TUTORIALS AND HANDBOOKS (DEPTH)**. Hier kann erklärender Text erstellt werden und Code-Einbettung über Drag&Drop erfolgen. Alles wird dann zum Beispiel in HTML-Format kompiliert

3.8 Trustworthy Framework Instantiation

Oft ist die Instantiierung von Frameworks eine sehr aufwendige Aufgabe. Es existieren zahlreiche Extension- und Variation-Points mit vielfältigen Abhängigkeiten. Bei White-Box-Frameworks kommt es zur Ableitung von nicht konformen Subklassen. Bei Black-Box werden nicht passende Klassen aufgrund von Multi-Point-Abhängigkeiten aggregiert. Einige Bedingungen sind statisch von vornherein nicht überprüfbar.

Problem 1: Instantiierung von 1-T—H-Hooks bei mehrseitigen Abhängigkeiten

Statischer, domänenspezifischer Constraint

z.B.: Auto-Konfigurator

Problem 2: Isomorphe Hierarchien (Katalog, Vorrat)

Dynamischer, domänenspezifischer Constraint

z.B.: Salespoint

Problem 3: Parallele Hierarchien

Statischer, technischer Constraint

z.B.: Parallele Variation von Fenster-Typen

Problem 4: Dynamische Annahmen

Dynamischer, technischer Constraint

z.B.: NULL-Prüfungen, Bereichsüberprüfungen, Sortierungen, Verträge

Cause	Stage	Static	Dynamic
Domain-specific		Multi-Point	Isomorphic Hierarchies
Technical (Design)		Parallel Hierarchies	Dynamic Assumptions

Abhilfe 1: Refactoring von mehrseitigen Abhängigkeiten

Constraint wird ins Framework bewegt, da sich Framework sonst nicht selbst kontrollieren kann

Abhilfe 2: Statische Verifikation statischer Constraints

Mit UML-Kollaborationen sind technische und domänenspezifische Instantiierungs-Constraints beschreibbar. OCL spezifiziert Invariante, Vor- und Nachbedingungen, verarbeitet Typen und kann Erweiterungen und Instantiierung analysieren

Abhilfe 3: Negatives Testen

Durchführen von Tests für das Verhalten bei Fehlinstantiierung neben den positiven Funktionstests aufstellen. Misuse-Diagramme, die für System-Abuse entwickelt wurden, kann auch Framework-Misuse, also fehlerhafte Instantiierungsbedingungen dargestellt werden

Abhilfe 4: Framework Instantiation Language

Wie in Eclipse werden Variations- und Erweiterungspunkte typisiert. Dies ist möglich für Code, GUI-Elemente und Geschäftsobjekte, die Sprache von Eclipse ist dabei XML-basiert und kann daher nur XML-Basistypen verwenden und eine baumartige Spezifizierung ermöglichen. Auf Logik basierte Sprachen können kontextsensitive Constraints und damit mehrseitige Bedingungen ausdrücken. Entsprechende Logik-Sprachen müssen dafür domänenspezifisch erweitert sein, also muss typisierte Logik zum Einsatz kommen, sowie OWL, SWRL (Ontologien), oder Frame-Logic. Konzepte wie Hooks, Variations- und Erweiterungspunkte müssen ausgedrückt werden.

Abhilfe 5: Dynamic Contract Checking

Dynamische, mehrseitige Constraints müssen zur Laufzeit ausgewertet werden, dies kann mit einem Framework-Contract-Layer oder Contract-Aspects passieren.

Abhilfe 6: Contract Aspects

Zuerst erfolgt die Kapselung der Vertragsprüfung in einer Schicht, diese kann dann in einen Vertrags-Aspekt bewegt werden. Werkzeuge für das Weaving wie Aspect/J integrieren dann den Vertrag. Einfacherer Austausch mit weniger Aufwand ist dabei möglich, wenn der Aspekt viele Schichten durchschneidet.

3.9 Binäre Kompatibilität von Framework-Plugins

Bei Änderungen am Framework auf Grund neuer Anforderungen, Code-Verbesserung und Bugfixing kommt es oft zu Inkompatibilität mit bereits existierenden Plugins und Anwendungen. Es kommt zu Umbenennungen, Hinzufügen, Löschen, Typänderung, Aufteilen und Verschmelzen. Das daraus resultierende Problem ist die Zerstörung der Kompatibilität der Binärversion eines Plugins. Die Lösung liegt in der Adaptierung des Plugins, auch iteriert, also mehrere Male bei wiederholter Anpassung des Frameworks. Der Vorteil ist erheblich für Third-Party-Firmen, die keine manuellen Anpassungen mehr vornehmen müssen. Framework-Entwickler erhalten verbesserte Unterstützung für die Entwicklung und wartung, Clients bekommen eine stille Plugin-Adaptierung.

Software-Evolution wird dadurch billiger, schneller und nutzerfreundlicher. Im B2-PDE-Projekt mit Comarch wurde eine automatisierte Plugin-Adaptierung umgesetzt.

4 Refactoring and Beyond

Refactoring bezeichnet die semantikerhaltende, aber strukturändernde Transformation eines Programms. Oft ist das Ziel, dass ein Design-Pattern entsteht. Es handelt sich um harmlose Operationen in der Evolution, also um keine Erweiterungen. Im Jahr 1992 wurde der Begriff von William Opdyke eingeführt. 1998 wird das Re-Factoring-Tool der Universität Karlsruhe in Together implementiert. Es kann Transformationen zwischen Klassengraphen leisten.

Die Klassen des Refactorings sind Umbenennung einer Entität (Alle Referenzen des Definition-Use-Graphs müssen aktualisiert werden), Bewegen einer Entität (Bewegung einer Klasseneigenschaft, also Attribut, Methode, oder Exception, muss durch Eigenschaften-Shadowing in einem bestimmten Bereich kompensiert werden), Trennen oder Vereinigen einer Entität (Methoden, Klassen, oder Pakete werden behandelt und alle Referenzen müssen aktualisiert werden). Das Outline Entity (Split Off) und Inline Entity (Merge) arbeiten auf einer Methode oder generischen Klasse und erlauben das Einfügen von Parametern. Für alle diese Klassen gilt, dass die Analyse des Definition-Use-Graph eine sehr schwierige Aufgabe ist.

Um vom Refactoring zur Software-Komposition zu kommen, müssen Operatoren in der modernen Software-Technologie betrachtet werden. **CONNECTORS** sind zwischen den Ports von Komponenten, **INHERITANCE** beschäftigt sich mit Wiederverwendung und mixin-basierter Vererbung, **PARAMETRIZATION** kann mit GenVoca erreicht werden. **ROLE MODEL MERGE** ist ein weiteres Element.

Wenn Software-Entwicklung als Algebra formuliert wird, dann müssen alle Aktivitäten als Operatoren verstanden werden. Elemente der Algebra sind dann die Connectors (Komponenten mit Ports), Inheritance (Klassen mit Eigenschaftslisten), Refacotings (Abstrakte Syntaxbäume), Parametrisierung (Templates für abstrakte Syntaxbäume). Es wird für alle ein Komponentenmodell mit Grey-Box-Komponenten angenommen.

Die **INVASIVE SOFTWARE COMPOSITION** aus dem CBSE-Kurs im Sommersemester erlaubt Adaption und Erweiterung von Komponenten an Hooks durch Transformationen. Eine Komponente wird als Fragment-Box mit einer Menge von Elementen (Klassen, Pakete) verstanden. Hooks können beliebige Fragmente oder Stellen in dieser Box sein. Beispiele sind der Beginn einer Methode oder das Ende, sowie deklarierte Hooks. Die Invasive Composition als Hook-Transformation dient mit ihren Composers als generalisiertes Konzept für die gerade besprochenen Operatoren. Connectors arbeiten dabei an deklarierten Hooks, genauso wie Inheritance. Role Model Merge hingegen verwendet implizite Hooks. Das Binden, Umbenennen, Entfernen und Erweitern funktioniert durch die Boxen, Composer und Hooks. Refactorings werden als Kompositions-Operatoren auf dem abstrakten Syntax-Graphen begriffen.

Als Konsequenz erhält man nun eine operatorenbasierte Sicht auf die Software-Entwicklung. Zu einem Zeitpunkt werden auf dem Entwurf Kompositionen angewendet, die zu einem Product-Build führen. Dabei werden sowohl semantikerhaltende, harmlose Operatoren, als auch gefährliche, nicht-semantikerhaltende, genutzt.

Die Algebra-Eigenschaften stellen nützliche Regeln wie Idempotenz, Kommutativität, Assoziativität und Monotonie (nur beim Hinzufügen durch Glueing und Extensions) zur Verfügung. Dies gilt aber nur für die semantikerhaltenden und syntax-erhaltenden Operatoren beim Refactoring, also Identitätsoperationen. Diese symmetrischen Operationen sind harmlos und können überall im Entwurf und Evolution verwendet werden. Die gefährlichen Entwicklungsschritte, Verbesserungen und Modifikationen müssen davon getrennt werden.

ABMANNS TRAUM besteht darin, dass alle Werkzeuge durch Refactoring-Operatoren und Kompositions-Sprachen ersetzt werden. Das führt zum automatisierten Design, Build und Evolution. Hierzu ist auch das RECODER-System zu beachten.

Zusammenfassend ist festzuhalten, dass Design-Patterns nur ein Weg für variable, erweiterbare Software für Frameworks sind. Auch die komponentenbasierte Software-Entwicklung (CBSE) kann mit den Komponenten Software teilen, und zwar nicht unbedingt nur in Klassen. Das ermöglicht ebenfalls Austausch. Komponen-

tenmodelle sind statische Abstraktionen, in denen Klassen nur ein Modell sind. Andere Modelle sind sehr wohl möglich.

Auch die Model-Driven-Architecture (MDA) arbeitet transformativ. Die Transformationen sind eine Variabilitäts-Methode. Die Varianten entstehen nicht durch Patterns, sondern Generatoren. Parnas hat bereits bei seinem Konzept der Modularisierung zwischen hervorgesehener und unvorhergesehener Evolution unterschieden. Dies setzen Produktlinien und MDA weiterhin um. Weitere Variabilitätskonzepte sind Templates für geplante Variabilität, sowie Aspects für ungeplante, sowie Views.