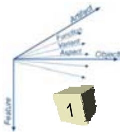
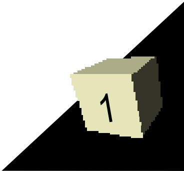
Multidimensional separation of concerns (MDSOC)

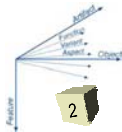
Von Michael Köhler
und Thomas Kühnau





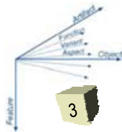
Übersicht

- Ursprung
- Separierung von Belangen
- Querschneidende Belange
- Hyperspace
- Studie: Finden von Belangen
- Fazit / Bewertung



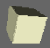
Ursprung

- Erste Erwähnung von „Separation of Concerns“ (SOC) bei Dijkstra und Parnas
- Schon damals als wichtiges Problem der Informatik erkannnt
- Lösung des Problems würde “ideale” Software erzeugen
  - 100% Wiederverwendbarkeit



Parnas 1972 in „On the criteria to be used in decomposing systems into modules“, Communication of the ACM
Dijkstra 1976 in „A discipline of programming“

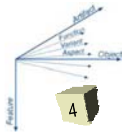
Baukastensoftware ermöglicht auch
eine einfachere Wartung, da die Lokalisation
und Aktualisierung der Software erleichtert ist



Separierung von Belangen

Bisherige Lösungen

- Aufteilung in verschiedene Dateien
- Modularisierung
- Objektorientierung
- Komponententechnologie

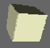


Die physikalische Dekomposition des Sourcecodes in Dateien impliziert keine hierachische Einteilung.
Sichtbarkeiten sind global.Wiederverwendbarkeit ist gering. Belange sind parallel angeordnet.

Die Modularisierung zeigt erste hierachische Einteilungen des Systems. Module erhöhen die Wiederverwendbarkeit.Sichtbarkeiten sind global.

Einführung von begrenzten Sichtbarkeiten und Vererbung. Das System wird streng hierachisch entwickelt. Wiederverwendbarkeit ist hoch.

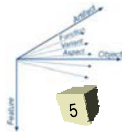
Die Komponententechnologie führt weitere hierachische Sichten ein. Die Wiederverwendbarkeit wird weiter erhöht.



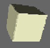
Separierung von Belangen

Bisheriges Vorgehen

- Das System wird beim Entwurf durch Komponenten, Subsysteme und Modulen in immer kleinere Teile zerlegt
- Bei der Implementierung werden zunächst die kleinsten Teile implementiert und diese dann zu immer größeren Teile zusammen gefügt



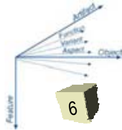
Die „Güte“ eines Systems ist zum Teil von dem hierachischen Aufbau abhängig.

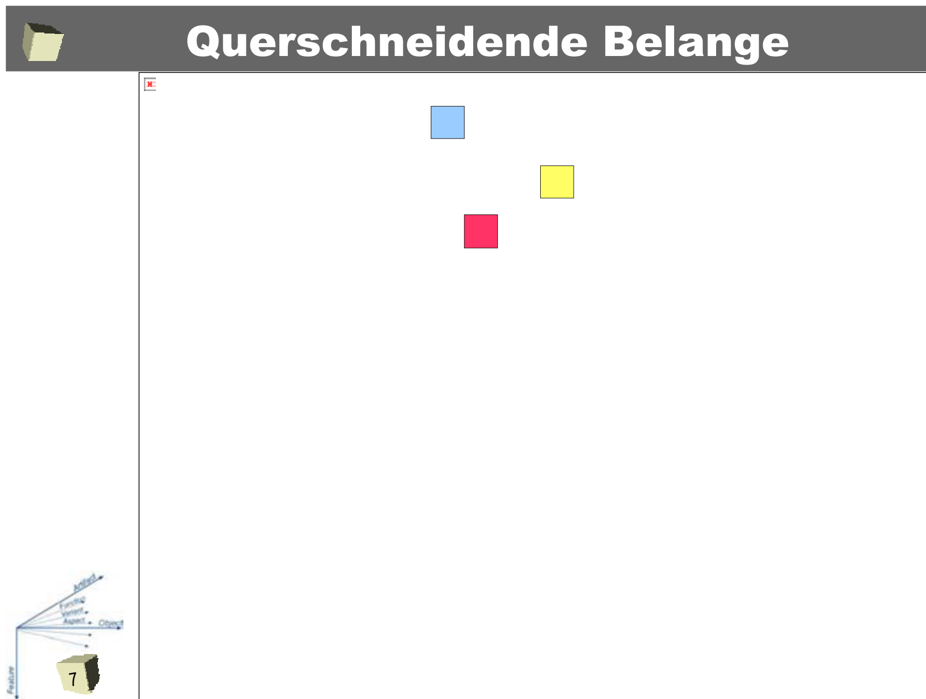


Separierung von Belangen

- Alle bisherigen Lösungen teilen Belange in hierachische Schichten ein

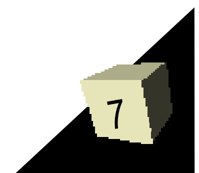
- Sie bringen keine Lösung für querschneidende Belange



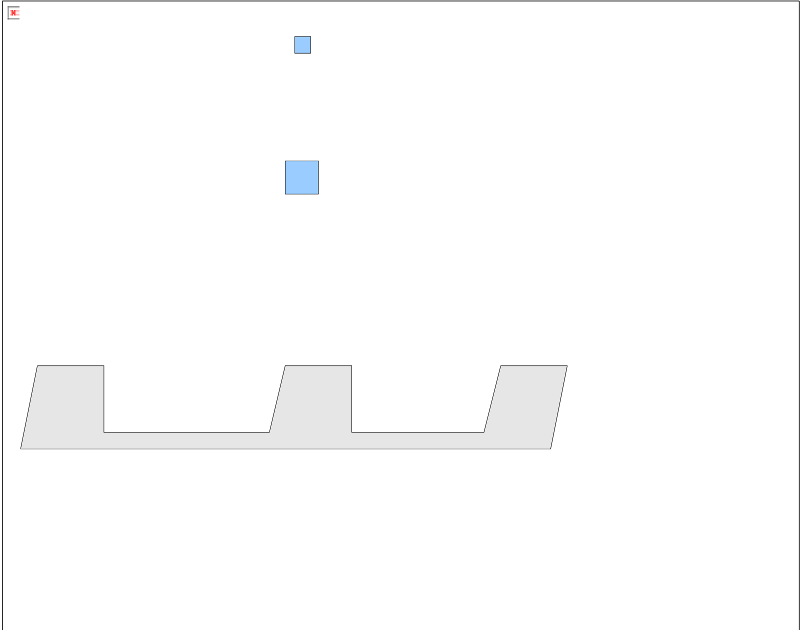


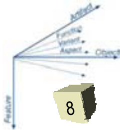
Abstrankter Raum der Belange wie er z.B. auf Dateiebene existiert.

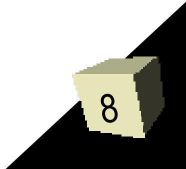
Die Aufteilung der Belange geschieht u.U. Willkührlich

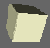


Querschneidende Belange



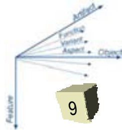


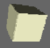




Querschneidende Belange

- Die hierachische Einteilung der Belange führt unter Umständen zu Problemen bei der Modifizierung eines querschneidenden Belangs
- Die Wartbarkeit wird erschwert
- Spätere Erweiterungen müssen in die Hierachie eingepasst werden



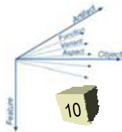


MDSOC - Einleitung

- **SOC** ist der Kern der Softwareentwicklung
- Klassifikation / Einteilung der Belange (concerns) erfolgt in der Regel nach Objekten/Klassen
- Es wird nur eine Dimension zur Separation herangezogen

  =>

  Diktatur der dominanten Dekomposition



Wie eben dargestellt ist ein Hauptproblem von Softwareentwicklung das Finden und Klassifizieren von Belangen. Durch möglichst reine Trennung sollen ein hohes Maß an Wiederverwendbarkeit und weniger komplexe Programme entstehen.

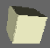
Dies hat bisher nicht funktioniert und scheint allem Anschein auch neuerdings nicht wirklich. Und nicht nur, weil bisher nur Code&Fix als Prozess bekannt ist. Anforderungen an die Software können sich im Laufe des Lebenszyklus bisweilen dramatisch ändern, dies führt dann in aller Regel zu den bekannten Problemen mit massiv-invasiven Änderungen im Code.

siehe letzter Vortrag mit den Wartungskosten.

Wo stehen wir jetzt?

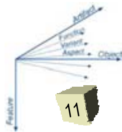
In der jetzigen Entwicklungsstufe des SE herrschen OO-Praktiken vor. Dadurch wird SOC auf Ebene der Objekte bzw. Klassen betrieben. Damit werden die Concerns aber nur in dieser Dimension (Klassen und Objekte) separiert. Objekte sind damit gut gekapselt und können leicht verändert werden. Andere Eigenschaften sind jedoch auf verschiedene Klassen verteilt und Änderungen an diesen Eigenschaften führen dann zu den beschrieben Problemen.

Diese Ausrichtung entlang einer Dimension wird von Ossher/Tarr “Tyranny of the dominat decomposition” genannt, von mir mit” Diktatur der vorrangigen oder dominanten Dekomposition” übersetzt.



Diktatur der Dekomposition

- Diktatur heißt, Trennung in nur einer Dimension. Das können z.B.
  - Klassen in OO Umgebungen
  - Funktionen in funktionalen Sprachen
  - Regeln in regelbasierten Systemen
- Die Entscheidung der Dimension fällt am Anfang des SE Prozesses
- spätere Änderungen/Erweiterungen der



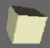
Diese Diktatur, die Trennung in nur einer Dimension, entsteht nicht nur in OO Umgebungen oder ist damit verbunden. Dieses Phänomen ist vorhanden, seit es Software gibt. Es ist bisher systemimmanent bei Softwareentwicklung vorhanden.

In OO Umgebungen sind es die Klassen/Objekt Struktur, in LISP die Funktionen und in Regelbasierten Systemen die Regeln.

Jede der obigen Separationen kann je nach Problemstellung besser oder schlechter sein.

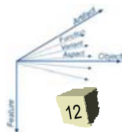
Dadurch ist es aber nicht möglich, die Vorteile der verschiedenen Dekompositionen während des Entwicklungszykluses zu nutzen.

Die Entscheidung der Entwicklungsumgebung und des Paradigmas (soweit diese Entscheidung überhaupt möglich ist) und die damit verbundenen SOC stehen ganz am Anfang des Entwicklungsprozess. Spätere Änderungen können dadurch deutlich erschwert werden. Letztendlich kommt eine Änderung wohl einer Neuentwicklung gleich.



Beispiel - Anforderungen

- Ein Programm zum Auswerten und Anzeigen mathematischer Ausdrücke
- Anforderungen an das System
  - Darstellung
  - Überprüfung und
  - Evaluation mathematischer Ausdrücke
  - Gemeinsame Repräsentation der Ausdrücke
  - Beschreibung in UML
  - Implementation in Java



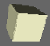
Zur Verdeutlichung ein Beispiel

Programmbeschreibung

requirements analyse bringt folgende Anforderungen ...

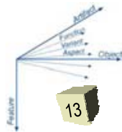
1 - 3 funktional

4 - 6 nicht funktional



Beispiel - Aufbau

- die funktionalen Anforderungen führen zu diesen Paketen
  - Darstellung
  - Überprüfung
  - Bewertung (Evaluation)
- Die gemeinsame Datenrepräsentation wird durch AST's (abstrakte Syntaxbäume) als shared data (s. SWA) erreicht.



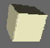
Diese nicht formalen Anforderungen führen zu einer Aufteilung in drei Pakete, die die einzelnen funktionalen Anforderungen erfüllen.

Darstellung
Überprüfung
Evaluation

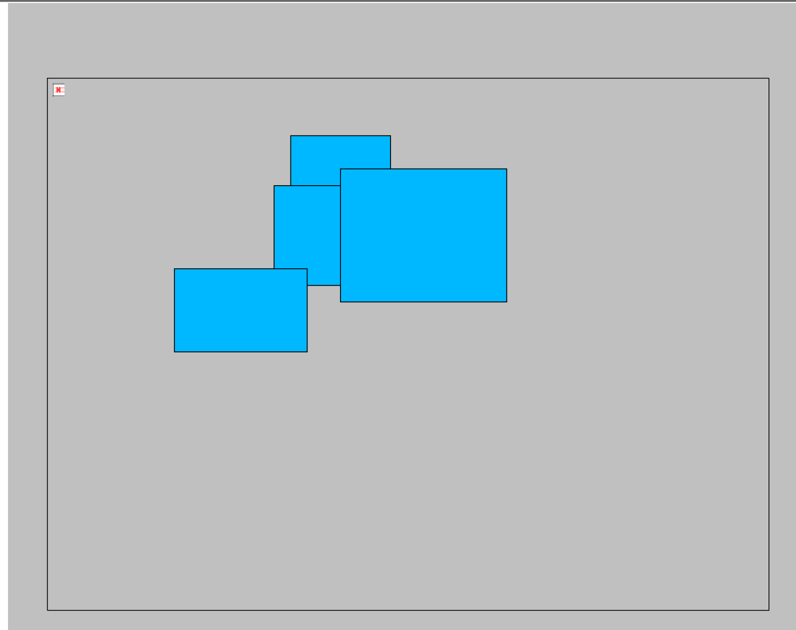
Bleiben nicht Funkionalen

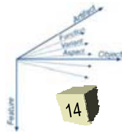
Die gemeinsame Datenrepräsentation wird durch abstrakte Syntaxbäume als shared data erreicht.

gemeinsame Datenrepräsentation --> AST .



Beispiel - Design



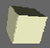


Überblick über das Design des Softwarepaketes

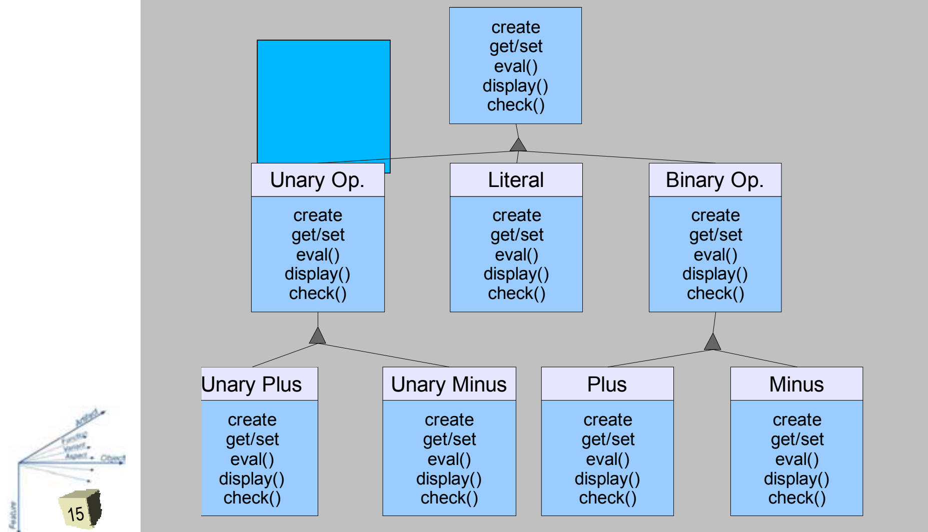
Im unteren Teil ist der AST (ASB?) zu sehen. Wahrscheinlich bekannt aus COM. Auf diesen Daten operieren die eigenständigen Module

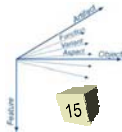
DISPLAY
CHECK
EVAL

Die Aufgaben sind offensichtlich und auch soweit gekapselt, dass diese Module zur wiederverwendet werden können.



Beispiel – Klassendiagramm



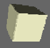


Ausschnitt aus dem UML – Klassendiagramm

Jede Klasse repräsentiert einen AST-Knoten. Es gibt Ausdrücke(Terme). Davon abgeleitet sind Literale, Unäre und Binäre Operatoren mit ihren entsprechenden abgeleiteten Klassen.

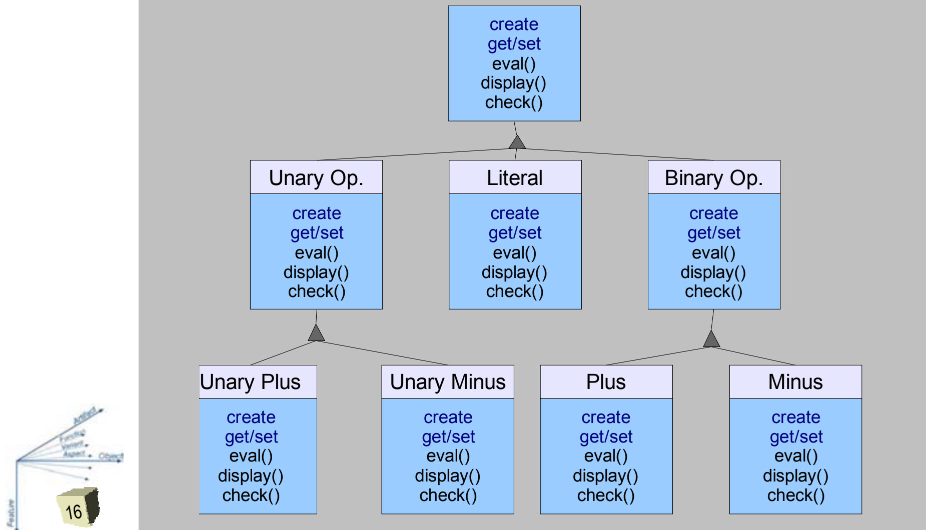
Die üblichen get/set oder Konstruktoren- Methoden sind in jeder Klasse enthalten . Außerdem enthält jede Klasse noch display(), eval(), und check() Methoden, um die entsprechenden Funktionalitäten zu implementieren.

Soweit ist das noch ganz normales, 'handelsübliches' Design.



Konstruktoren

- Get/Set und Konstruktoren

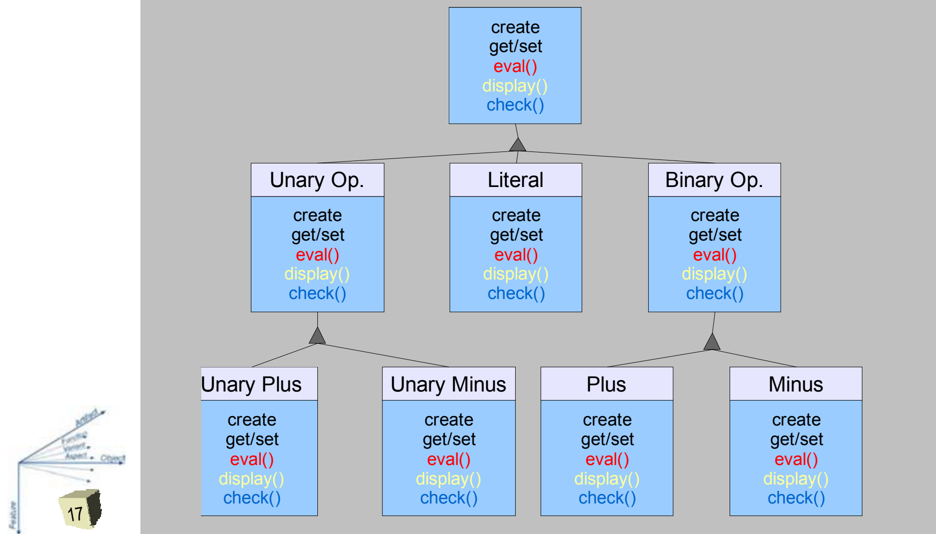


Hier sind die get/set -Methoden und die Konstruktoren markiert. Der Grund ist, das all diese Methoden mit der Kernfunktionalität des AST's umgehen (müssen).

Diese Methoden sind dadurch im Grunde genommen nicht wirklch unabhängig sondern stellen den gemeinsamen Zugriff auf die AST's dar. Da in der gewählten Dekompsition diese Funktionalität in dieser Form nicht darstellbar ist, haben wir hier den ersten querschneidenden Belang .

Änderungen an den AST's müssen also in **jeder** Klasse vorgenommen werden.

Das ist wohl nicht wirklich das, was man sich durch SOP erhofft hat.



Ein weiteres, schon implementiertes Crosscutting sind die funktionalen Anforderungen.

Erkenntlich an der Farbe(und am Namen) sind die einzelnen Anforderungen auf die verschiedenen Klassen verteilt.

Das ist ebenfalls in der jetzigen Form des SE nicht anders machbar.

Durch den Entschluss oder die Vorgabe, die Software in Java zu Implementieren kommt man nun einmal zu solchen Zerlegungen.

Das Ganze ist bis jetzt auch noch nicht dramatisch, es dient nur zur Darstellung des Problems

Evolution – neue Anforderungen

- Evolution: für Softwaresysteme eine gefährliche Bedrohung ?
- neue Anforderungen an das System
  - Bessere Performance durch caching der evaluierten Teile (Dimension Funktion)
  - Überprüfung der Ausdrücke nach verschiedenen Stilen (konfigurierbar) (Dimension Funktion)
  - selektives logging der Auswertung (Dimension ?)

Problematisch wird die Situation erst, wenn im Laufe des Softwarelebens neue Anforderungen hinzukommen.

Die Frage, die sich wohl immer wieder stellt ist: Wie kann man Softwaresysteme so erstellen, dass man nachträglich noch einfach und sicher Modifikationen an der Funktionalität vornehmen kann.
Ist Evolution für Software generell ein Problem? Ist diese These so gerechtfertigt?

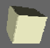
Bsp.: WIN 3.11 – WIN ME

zurück tu unserem Bsp.

Neue Anforderungen (nach einiger Zeit des Einsatzes):
Die Anwender sind durch zunehmend komplexere Ausdrücke nicht mehr mit dem Laufzeitverhalten des Systems zufrieden. Durch Profiling wurde festgestellt, dass mehrfaches Auswerten des AST zu Wartezeiten führt. Durch Caching soll dieses Problem behoben werden.

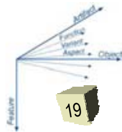
Die Benutzter würden die mathematischen Ausdrücke gern an verschiedenen ausgewertet bekommen. Diese Stile sollen zur Laufzeit gewählt werden können.

Außerdem sollte selektives Logging der Auswertung erfolgen können. Durch die Selektion zur Laufzeit ist auch diese Anforderung kein wirklicher Kandidat für AOP.



Evolution - Konsequenzen

- durch die vorrangige Dekomposition in Klassen/Objekte sind Anforderungen in der Dimension Funktion massiv-invasiv
  - ein neues Property cacheRslt ist für jede Klasse, die eine eval()-Methode besitzt, anzulegen
  - jede check()-Methode ist noch zu erweitern, um den verschiedenen Stilen zu entsprechen
  - da das logging selektiv sein soll, ist es nicht mit den üblichen AOP-Praktiken zu erschlagen, sondern muss z.B. mit einem Observerpattern behandelt werden

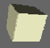


Die Forderung nach besserer Performance, und das daraus resultierende chaching bedeuten, dass ein neues Property cacheResult eingeführt wird. Diese Instanzvariable und die entsprechenden Zugriffsmethoden betreffen alle get/set und alle eval -Methoden.

Ähnlich Problematik ist auch bei den anderen Anforderungen bzw. ihrer entsprechenden Lösung zu sehen.

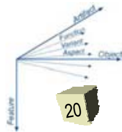
Alle Änderungen betreffen einen großen Teil bzw. alle Klassen des bestehenden Softwaresystems.

Vor allen Dingen, muss man die entsprechenden Stellen erst einmal finden.



Evolution - Komplexität

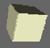
- Diese neuen Anforderungen sind noch nicht einmal unabhängig, wie logging und caching zeigen.
- Die Anforderungs-Artefakte und die Software-Implementierung sind ebenfalls in verschiedenen Dimensionen angelegt (Funktion versus Objekt).



logging und chaching betreffen beide auch noch dazu, zumindest teilweise die gleichen Methoden.

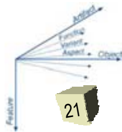
Und so ganz nebenbei fällt vielleicht auf, dass die ganze Zeit und eigentlich immer schon eine Übersetzung zwischen zwei Dekompositionen stattfindet.

Die Anforderungen sind meist funktional während die vorrangigen Artefakte bei der Programmierung / Implementierung auf Daten/Objektebene geschieht.



Idee des MDSOC

- Lösung des Problems
  Multi **D**imensional **S**eparation **O**f **C**oncerns
  - viele unabhängige Dimensionen von Belangen
  - Trennung entlang dieser Dimensionen
  - die Möglichkeit, neue Belange und neue Dimensionen dynamisch zu bearbeiten, immer dann, wenn sie auftauchen
  - überlappende und interagierende Belange



- MDSOC ist nicht abhängig von der Sprache oder einem Paradigma

Dabei ist

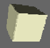
- **Dimensionen**: z.B. **Objekt/Klassen** bzw. Daten sind eine wichtige Schlüsseldimension. Eine weitere wichtige Dimension ist, wie schon gezeigt **Funktion**. Es können noch beliebig viele weitere Dimensionen existieren.

- Man betreibt SOC nun entlang der Dimensionen, die man identifiziert hat.

- Sobald allerdings neue Dimensionen oder Belange auftauchen, sollen diese „just in time“ benutzt werden können. Damit wäre einer Erweiterung eines Softwaresystems eine großer Teil seiner Probleme genommen.

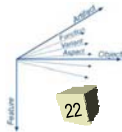
- Diese Belange sollen sich auch durchaus überlappen können oder voneinander abhängig sein.

- Diese ganze Idee soll nicht an ein Paradigma oder eine Programmiersprache gebunden sein.



Ziele des MDSOC

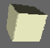
- gleichzeitige Identifikation und Kapselung jedweden Belangs zu jeder Zeit, ohne sich um andere Belange zu kümmern
- automatische Integration der separierten Belange
- nice to have:
  automatisches Modularisieren bestehender Systeme



Um dieses Ziel zu erreichen, muss es möglich jeden Belang gleichberechtigt für sich zu betrachten. Es darf **keine** vorherrschende, **diktatorische Dimension** geben. Dazu muss jeder Belang als Einheit gekapselt sein. Auf diese Weise kann man das System inkrementell erweitern.

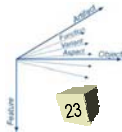
Damit die ganze Trennerei einen wirklichen Nutzen hat, sollten auch noch eine Möglichkeit bestehen, die aufgetrennten Teile wieder zu einen funktionierenden Ganzen zu verbinden.

Was das ganze natürlich noch einfacher bzw. für die Praxis nützlicher machen würde. wäre die Möglichkeit bestehende „Alt“Software automatisch zu „remodularisieren“.

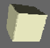


Hyperspace

- *hyperspaces* sollen die explizite Identifikation von wichtigen Belangen und Dimensionen und deren Kapselung ermöglichen.
- Das Ziel sollen möglichst geringe Auswirkungen der Änderungen und damit eine Einfachheit bei der Entwicklung sein.

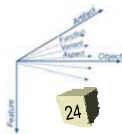


viele Dimensionen -> hyperspace



Modell - Units

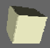
- ***units* sind Artefakte, wie z.B.**
  - Deklaration
  - Statement
  - Zustandsdiagramm
  - Anforderungsspezifikation
  - usw.
- *units* können primitiv (Methode, Variable) oder zusammengesetzt sein (Klasse, Paket, ...). Zusammengesetzte *units* werden wie



Nun zu einer Beschreibung der Bestandteile, die nötig sind, um die vorgenannten Forderungen bzw. Wünsche zu erfüllen.

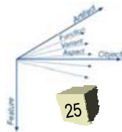
Software besteht aus verschiedenen Artefakten, die zusammen diese Software beschreiben bzw. bilden. Diese Artefakte müssen nicht alle auf **derselben Basis(sprache)** definiert sein. *unit* wird definiert als ein syntaktischer Konstrukt in einer solchen Sprache. Unit kann eine Deklaration, Statement, Zustandsdiagramm, eine Spezifikation oder eine andere Einheit, die eine Software in einer bestimmten Weise beschreibt.

Units können primitiv oder zusammengesetzt sein. Diese zusammengesetzten units werden wie Module behandelt. Ob und wann eine unit primitiv oder zusammengesetzt ist, liegt im Auge des Betrachters bzw. in der Art und Weise, wie man das gesamte Modell instanziiert. Ein verallgemeinerndes Vorgehen ist nicht vorgesehen und für die weitere Betrachtung auch nicht wichtig.



Modell – concernspace

- vereinigt die Softwarekomponenten des Systems
- organisiert die *units*, so dass die nötigen (wichtigen) *concerns* identifiziert und separiert werden können
- jedes Artefakt kann in einen *concernspace* abgebildet werden
- ein *hyperspace* ist ein mehrdimensionaler

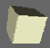


Der Concernspace umfasst nun alle units, die eine bestimmte Software ausmachen. Zum Beispiel enthält ein Concernspace für unsere mathematische Termauswertung alle Softwareartefakte die wir beschrieben haben, d.h. die 'alten', ursprünglichen und auch die 'neuen' Artefakte.

Der Sinn oder das Ziel ist, dass der CS die units so organisiert, dass alle wichtigen Belange identifiziert und separiert werden können. Dabei müssen auch die überschneidenden Belange Berücksichtigung finden. Sie müssen in einer adäquaten Art und Weise dargestellt und bearbeitet werden können. Außerdem muss noch die Vereinigung der einzelnen units zum Ganzen möglich sein.

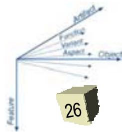
Dazu muss jedes Artefakt der Software in einen Concernspace abgebildet sein. Hier wird bewusst ein Concernspace, also Singular, benutzt, da es parallel mehrere Räume gibt bzw geben kann.

Alle Räume zusammen bilden dann einen Mehrdimensionalen Hyperraum oder hyperspace.



Modell – hyperspace Definition

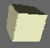
- *hyperspace* ist ein Tupel (U, M, H), wobei
  - U die Menge der *units*
  - M die *concernmatrix*, die gleichzeitig die *units* in U entsprechend ihrer Zugehörigkeit organisiert
  - H die Menge der *hypermodules*, die beschreibt, wie man Module aus den Komponenten aus U bildet.
    H bildet die Trennung der Belange ab.



Formal ist ein HS ein Tupel (U,M,H)

units wurden schon geklärt,

auf die Eigenschaften der Concernmatrix und der Hypermodule gehe ich im folgenden ein



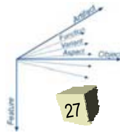
Modell - concernmatrix

- Alle Belange zusammen bilden einen mehr-dimensionalen Raum, die *concernmatrix.*

- Die Menge aller *units* aus U, die einen Belang adressieren, wird durch

$$U(c) = \{ u \in U \mid c(u) \}$$

  beschrieben.



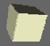
- *concerns* überschneiden sich, wenn die Mengen ihrer *units* nicht disjunkt sind.

zuerst zur CM

Mehrere Belange in einer Software sind meist **gleichzeitig** von Bedeutung, u.U. sind und diese noch **überlappend**, wie z.B.die Methode **Number.display()** einerseits im Kontext der Klasse Number und andererseits in Feature display vorhanden sind.
Diese Tatsache führt letztendlich zu einer mehrdimensionalen Matrix von Belangen. Diese Matrix ist die auf der vorherigen Folie angesprochene Concernmatrix.

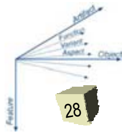
Alle units, die einen bestimmten Belang adressieren, kann man zu einer Menge zusammenfassen. Diese Menge Menge charakterisiert dann diesen Belang.

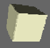
Damit kann man auch einfach die Abhängigkeiten der Belange ableiten.



Modell - concernmatrix

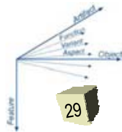
- *Eine Dimension von concerns besteht aus einer Menge von Belangen aus U, die U partitioniert.*
- *Ein Punkt (unit) im Hyperraum wird durch genau eine Angabe pro Dimension bestimmt, d.h. Belange können innerhalb einer Dimension nicht überlappen.*
- *Es müssen alle units in U erreichbar sein.*





Modell - conernmatrix

- Jede Dimension *d* besitzt einen speziellen Belang, den *NoneConcern* N_d.
- Alle *units*, die keinen anderen *concern* in *d* adressieren, adressieren N_d.
- Damit sind in N_d alle unbeeinflussten *units* gesammelt.

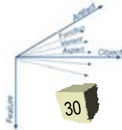


Dies bedeutet aber, dass jede Dimension einen besonderen Belang benötigt, den NoneConern. Dieser Concern nimmet alle die units in einer Dimension auf, die von keinen anderen Concern in dieser Dimension adressiert werden. Zum Beispiel werden funktionale units keine Belange innerhalb der Objektdimension adressieren. Für diese Dimension werden sie also auf den NoneConcern zeigen.

Diese Menge der unbeeinflussten units kann natürlich auch leer sein.

Modell - conernmatrix

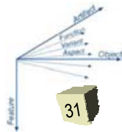
- Die *concernmatrix* wird über der Menge der *units in U* als Tupel (C,D) definiert, mit C als Menge von Belangen und D als Menge von Belang-Dimensionen, so dass
  - jeder *concern* aus C in genau einer Dimension aus D enthalten ist
  - jede Dimension *D* die Menge *U* partitioniert.
- Es soll jeder *concern* oder jede Dimension als primärer Fokus dienen können.



.

Modell - Hypermodule

- Ein *hyperslice* ist eine Menge, bestehend aus *units*. Ähnlich den Modulen kapseln *hyperslices* Belange, ohne sich an die Grenzen der üblichen Artefakte zu halten.
- Jede *unit* muss mindestens einem *hyperslice* angehören => *hyperslices* entsprechen Softwaremodulen.
- *hyperslices* können zu *hypermodules* zusammengefasst werden.



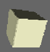
Die eben beschriebene Matrix ist nicht dazu geeignet, Belange zu kapseln. Sie identifiziert Belange. Man kann damit feststellen, welche units zu welchen Belangen gehören. Um jedoch komfortabel mit einem Belang umgehen zu können, ist es notwendig diesen auch zu kapseln.

Bislang war Kapselung nur möglich, wenn die eingesetzte Programmiersprache solch ein Konstrukt kannte, wie zum Beispiel Module, Klassen oder Funktionen. Damit ist man aber wieder bei dem ursprünglichen Problem der Diktatur.

In diesem Modell soll eine Kapselung nicht nur an Hand der Formalismen der Artefakterzeugenden Entwicklungswerkzeugen möglich sein.

Eine Menge von units wird zu einem Hyperslice zusammengefasst.Ähnlich Modulen oder Klassen kapseln Hyperslices Belange. Diese Hyperslices sollen sich aber eben nicht an die Grenzen der primären Dekomposition halten, sondern übergreifend kapseln. Die Hypeslices bilden also Softwareblöcke ähnlich wie Klassen oder Module.

Es ist klar, dass jede unit mindestens einem Hyperslice angehören muss. Neue units werden immer in einen Hyperslice angelegt. Wenn Hyperslices zusammengefasst werden, kann man aicu von Hypermodulen sprechen.

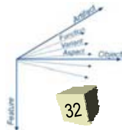


Deklarative Abgeschlossenheit

- Um eine Entkoppelung zu gewährleisten, wird eine deklarative Abgeschlossenheit der *hyperslices* gefordert.
- Jede Referenz muss deklariert, aber nicht notwendigerweise implementiert sein.
- Definition der Implementierungsmenge eines *hyperslices*

  $I(hs) = \{ u \in hs \mid \neg decl(u) \}$

  (wobei decl ein Prädikat ist, das Deklarationen identifiziert)

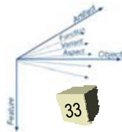


Beispiel: Angenommen, ein Display hyperslice enthält eine unit u1, die eine unit u2 benutzt , die im Kern-AST definiert wurde(z.B. getOperand()). Um die deklarative Abgeschlossenheit zu des Display Hyperslice zu gewährleisten, muss dieser slice noch eine unit u2decl enthalten, die getOperand() deklariert (nicht implementiert). Wenn dann in u1 u2decl statt u2 benutzt wird, sind die beiden slices entkoppelt. Dafür muss dann u2decl noch irgendwie an eine implemtierte unit gebunden werden (spätestens zur Laufzeit). Dies ist keine alzu neue Forderung bzw. Technik für Programmiersprachen. Durch die lockere Def. von units weitet sich dieser Mechanismus aber auf andere Brereiche aus.

Damit kann man die Implementierungsmenge eines Hyperslices definieren

Korrespondenz

- Die Assoziation zwischen der Deklaration einer *unit* u_{2decl} und einer Implementierung u_2 nennt man Korrespondenz.
- Korrespondenz ist eine ziemlich lockere Form von Bindung.
- Dadurch kann man Implementierungen leicht ersetzen.

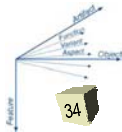


Da totale Isolation von hypeslices eher unwahrscheinlich ist, muss man eine Beziehung zwischen diesen hyperslices herstellen können. Um z.B. die Deklaration einer unit an eine passende Implementierung zu binden muss eine solche Beziehung, die Korrespondenz, hergestellt werden.

Da units per definition nicht nur Code darstellen, kann hierbei auch eine Beziehung zwischen einer requirement-unit und einem design unit (oder mehreren) hergestellt werden.

Korrespondenz

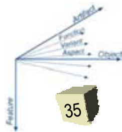
- Die Frage, ob korrespondierende *units* kompatibel sind, bedingt syntaktische **und** semantische Prüfungen (weiterer Forschungsbedarf !!).



Hyperslice

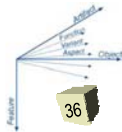
- Formal ist ein *hyperslice* ein deklarativ abgeschlossener *concern* in einem *hyperspace (U, M, H),* also hs C

- *hyperslices* sind dem Grunde nach Belange mit der Zusatzforderung der deklarativen Abgeschlossenheit.



Hypermodules

- Ein *hypermodule* wird benutzt, um eine Menge von *hyperslices* zu integrieren.
- Dazu müssen die Korrespondenzen zwischen den *hyperslices* beachtet werden.
- Def.: eine *hypermodul* ist ein Tupel (HS,CR) mit HS als Menge von *hyperslices* und CR als Menge von Kompositionsbeziehungen.



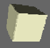
da hyperslices sozusagen den divide teil des divide-and-conquer Prinzip darstellen, muss es noch ein Teil geben der wieder zusammenführt, was einmal getrennt wurde.

dazu sind die hypermodule da.

Um hyperslices in ein hypermodule zu integrieren ist es notwendig die Korrespondenzbeziehungen die zwischen den hypeslices bestehen zu identifizieren.

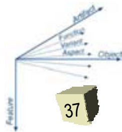
man muss dann entscheiden, wie man die hypeslices kombiniert, um den gewünschten Effekt zu erreichen.

dazu folgende def.



Hypermodule

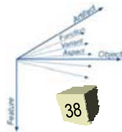
- Kompositionsbeziehungen sind Tupel *(I,r,f,o)* mit
  - *I* ist ein Tupel von Eingabe *units*
  - *r* ist die Korrespondenzbeziehung
  - *f* ist die Kompositionsfunktion *f:(I   r)* ➊ *U*
  - *o ist eine Ausgabeunit*
- *hypermodules sind absichtlich abstrakt gehalten.*



Anwendung des Modells

- Gefundene Dimensionen
  - Artefakte (Design, Requirements, Code)
  - Eigenschaften (Display, Check, Kern)
  - Objekte (Plus, BinaryOperator, …)

Hypermodule	Hyperslices	Composition Rules
FeatureHD	KernelHS DisplayHS EvalHS	merge
ArtefaktHD	RequirementsHS DesignHS CodeHS	merge



zurück zum anfänglichen Beispiel

.. (dimensionen vorstellen).

zwei hyperslice Dimensionen sind im Bild zu sehen.

- Inhalt hyperslice erklären

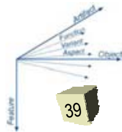
verschiedene Sichten auf gleiche Software

Anwendung des Modells

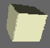
- Ein weiteres Feature des Modells sind Schnittmengen von *hyperslices*
  - z.B. ein Programm, das nur Anzeige- und Kernfunktionalität beinhaltet

    U(kernel) ➂ U(code) und U(display) ∩ U(code)

Hypermodule	Hyperslices	Composition Rules
KernelCodeHS	KernelHS CodeHS	declaratively complete intersection
DisplayCodeHS	KernelHS CodeHS	declaratively complete intersection
KerneDisplayCodeHS	KernelCodeHS DisplayCodeHS	merge

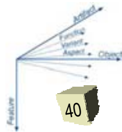


Wenn man nun eine neue Software herstellen möchte, (Produktlinien) kann man folgendermassen vorgehen:



Anwendung des Modells

- Erweiterung der Anwendung um Caching
  - Caching ist querschneidend zu vielen Belangen
    - alle Klassen der Objektdimension
    - Evaluation und Kernmodule der Funktionsdimension
    - alle Artefakte in der Artefaktdimension
- => Caching gehört in eine neue Dimension



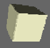
Zum nächsten Problem, der Evolution,

caching ....

Dieses Hinzufügen einer neuen Dimension ist einfach zu bewältigen. Da es unwahrscheinlich ist,dass die neue Dimension bestehenden units berührt, kann man einfach für die bestehenden units dem Noneconern für chaching einzutragen.

Neue units müssen aber in dem Kontext eines Hyperslices eingetragen werden -> CachingHS. -> CachingHD. Für jede neue unit muss der korrekte Bezug bzw. location in den anderen Dimensionen beachtet werden. z.b. cacheRslt gehöhrt zu Expression, kern-ast usw.

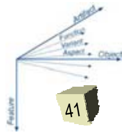
Außerdem muss natürlich auf deklarative Abgeschlossenheit geachtet werden.



Anwendung des Modells

- Entfernen von *concerns* durch Entfernen der *units* aus einem *hyperspace (U,M,H)* ist ein massiv invasiver Vorgang.

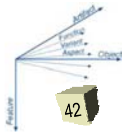
- Um eine Implementierung $u \in hs$ zu entfernen, muss u *in* u_{decl} transformiert werden. Danach muss die Validität der Komposition kontrolliert werden.

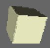


Zur letzten und vielleicht problematischsten Änderung.

Zusammenfassung

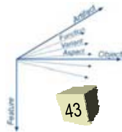
- Der Fokus kann bei der Entwicklung auf jeden Belang gelegt werden.
- Das Hinzufügen von *units*, *concerns* oder Dimensionen gerät vergleichsweise einfach.
- Das Entfernen von *concerns* ist ebenfalls vergleichsweise günstig.





Studie: Finden von Belangen

- Welche Belange gibt es in einem Programm?
- Untersuchung des “Computer Science and Engineering” in San Diego
- Es wurde untersucht was Hyperspace und AOP in realer, produktiver Software bewirken und verbessern kann

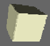


- Am Ende soll die Software mit den

Studie: Finden von Belangen

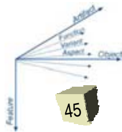
- Unix GNU “sort.c” als Untersuchungsobjekt
- Umfang von 2145 Textzeilen in einer Datei
- Das Ziel ist, jedes Zeichen des Sourcecodes mindestens einem Belang zuzuordnen

sort.c wurde gewählt, da das Programm recht bekannt, bewährt und produktiv benutzt wird.

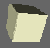


Studie: Untersuchung

- Untersuchung des Programmaufrufs
  sort -mcubdfinrtx [+pos [-pos]] ... [-o output] [-T dir] [file]
- Vom Programmaufruf unterstützte Belange
  - Sortierreihenfolge
  - Verschiedene Check-Methoden
  - Umleitung der Ausgabe und der Temp-Files
  - Angabe verschiedener Sortier-Keys
  - Numerische Sortierung und Sortierung nach Monaten

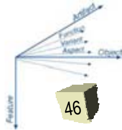


m	merge only
c	check only
u	unique records
pos	mehrfache Keys
dfinbr	Änderung des Key Vergleiches
T	Ordner der temp-Files



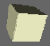
Studie: Untersuchung

- Die Implementierung unterstützt große Dateien
  - Datei wird in 32k große Sektionen unterteilt
  - Sortiern und Mergen der Sektionen
- Sicherheit
  - z.B.: Abfangen von falschen Aufrufen
    - Input File = OutFile u.a



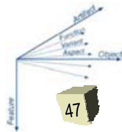
- Daher kann das Programm als robust und

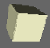
Die Sektionen werden einzeln sortiert und jeweils 16 werden gemerged



Studie: Untersuchung

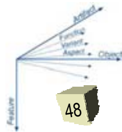
- Der Kern umfasst ein 51 Zeilen großes Modul eines Merge-Sort
- Keines der Basis-Belange berührt diesen Teil. Der Kern könnte daher ohne Probleme in anderen Programmen verwendet werden
- Die meißten andere Belange erweitern den Funktionsumfang des Merge-Sort





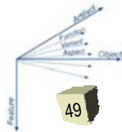
Studie: Ergebnis

- Gefunden wurden 60 zum großen Teil funktionale Belange
- Nur 18 sind nicht für Sortier-Belange zuständig
- Die Belange Textausgabe, Komandozeilenparsing, Konfigurationsoptionen und Sortierfeatures können nicht hierachisch angeordnet werden
- Sie bilden im Hyperspace vier separate Dimensionen



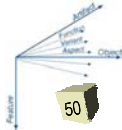
Studie: Ergebnis

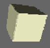
- Die Interaktion zwischen 27 der 60 Belange kann als einfaches Hyperspace beschrieben werden
- Sie bestehen aus einer Dimension mit 20 Konfigurations-Belange die 3 Dimensionen mit 7 Feature-Belange querschneiden



Studie: Ergebnis

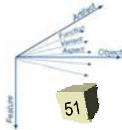
- Bestehende Software zu überarbeiten ist mit einem großen Arbeitsaufwand verbunden
- Die Untersuchung wurde ausschließlich manuell durchgeführt, da noch keine Tools existieren
- Es besteht die Gefahr wichtige Belange zu übersehen





Studie: Ausblick

- sort.c muss weiter durchsucht werden
- Die Reimplementierung wurde erst begonnen
- Das neue Programm soll durch schrittweise hinzufügen von weiteren Features erweitert werden
- Das Experiment soll bei anderen Programmen wiederholt werden
- Für weitere Untersuchungen sollen Tools entwickelt



Der Sourcecode wurde noch nicht vollständig durchsucht. Einige der wichtigsten Belange bilden im Hyperspace noch einen dicken Knoten. Das hinzufügen des „Programmnamen-Belang“ als Option im Programmaufruf bewirkt tiefgreifende Änderungen des Programmcodes.

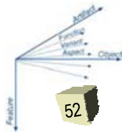
Dadurch soll die Möglichkeit gegeben werden, verschiedene Varianten der Komposition zu testen. Man erhofft dadurch neue Erkenntnisse bei der Implementierung produktiver Software zu erhalten.

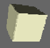
Als nächster Schritt sollen Programmgrößen um 10000 und 50000 LoC untersucht werden.

Ein Tool könnte z.B. Bereiche des Sourcecodes, der bereits einem Belang zugeordnet wurde, kennzeichnen, oder selektierte Bereiche einem Belang zuordnen. Wünschenswert, wäre ein Tool, das Belange selbstständig erkennt und zuordnet.

Fazit & Bewertung

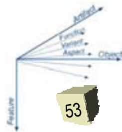
- Informationen sind nur sehr wenige zu finden
- Seit 2003 sind neue Publikationen fast nicht zu finden
- Überwiegend nur Überlegungen zur Herangehensweise ans Thema
- Webseiten verweisen immer auf die selbe Hyper/J Webseite





Fazit & Bewertung

- Bisher nur zwei Tagungen zum Thema.
- Harold Ossher und Peri Tarr scheinen die treibende Kraft zu sein
- Die Idee des Hyperspace ist in vielen Teilen noch nicht fertig durchdacht
- vollständige automatisierte Erstellung des Hyperspaces ist wohl sehr schwierig



Quellen

- “Bessere Software durch Querschneidende Module”, Klaus Ostermann, TU Darmstadt
- “Sorting out Concerns”, Lee Carver & William G. Griswold, Computer Science and Engineering, San Diego
- “Multi-Dimensional Separation of Concerns in Hyperspace”, Harold Osher & Peri Tarr, IBM T.J. Watson Research Center
- “Dimension Templates: Multi-dimensional separation of Concerns in UML”, Marcus Fontoura, Computer Science Department, Princeton University
- “Using Multidimensional Separation Of Concerns To (RE)Shape Evolving Software”, Harold Ossher & Peri Tarr, Communication of the ACM, Oktober 2001 Vol. 44 No. 10