6. Objekte - eine Einführung

Wir möchten hier aus Sicht des Software Engineerings den Begriff Objekt, Abstrakte Datentypen und vor allem Kohäsion und Kohäsion genauer zu erklären.

6.1. Was ist ein Modul?

Ein Programm, welches kaum strukturiert ist, kann auch von guten Programmierern kaum gewartet werden. Selbst der Programmierer, der das Programm erstellt hat, wird nach kurzer Zeit nicht mehr in der Lage sein, sein eigenes Programm zu verstehen.

Die Lösung:
Aufbrechen des Programmes in mehrere kleinere Teile, Module, die klare Funktionen haben und isoliert entwickelt, gewartet und weiter entwickelt werden können.

Aber wie erkennt man, wann eine Zerlegung gut oder schlecht ist? Wie soll man konkret vorgehen?
In einer wichtigen Arbeit haben 1974 die damaligen IBM Mitarbeiter Stevens, Myers und Constantine ("Structured Design", IBM Systems Journal 13 (No. 2, 1974), pp. 115-139) den Begriff *Modul* wie folgt definiert:

Randbemerkung
Objekt orientierte Konzepte wurden schon 1966 in die Informatik eingeführt, in der Programmiersprache SIMULA, entwickelt von mehreren Norwegern (Dahl, Nygaard) . Zu diesem Zeitpunkt war allerdings diese Technologie zu radikal, um praktisch eingesetzt zu werden. 1980 wurden die Konzepte im Wesentlichen wieder erfunden., im Zusammenhang mit dem Begriff der Modularität.
Ein anderes Beispiel ist das Information Hiding, welches von Parna 1971 vorgeschlagen wurde. 10 Jahre später wurden seine Konzepte im Zusammenhang mit den Abstrakten Datentypen neu aktiviert.

- Ein Modul besteht aus einem oder mehreren Programm- Anweisungen, welche über einen gemeinsamen Namen von andern Teilen des Systems angesprochen werden können und wann immer möglich eigene Variablen besitzen

Ein Modul besteht also aus Programmcode, der analog zu Funktionen oder Prozeduren angesprochen werden kann, und einen eigenen Zustand hat.

Kommt Ihnen doch irgendwie bekannt vor?

Die Definition scheint sehr breit und allgemein zu sein. Sie umfasst auch Prozeduren, Funktionen (und.... Objekte...). Die Definition umfasst auch COBOL paragraphs und sections, ohne dass diese eigene Variablen haben. Aber die Definition besagt ja: "... wann immer möglich ...". Das muss also nicht sein.

Die Definition ist aber noch nicht perfekt:
- Assembler Macros werden nicht aufgerufen, sondern ausgeführt, nachdem sie geladen wurden
- In C++ werden Definitionen mit Hilfe von Header Files "**#include**d" und nicht ausgeführt
- In Ada werden Packages (abstract data types) oder Ada generic (Macros) nicht ausgeführt

Aber diese Konstrukte sollten in einer Definition eines Moduls berücksichtigt werden.

Constantine und Yourdon haben später eine allgemeinere Definition gegeben:

- Ein Modul ist eine lexikalisch zusammenhängende Sequenz von Programm Anweisungen, welche von klammernden Elementen zusammen gefasst werden und mit Hilfe eines Namens ansprechbar sind.

Beispiele für klammernde Elemente sind:

- Begin ... End
- { ... }

Diese Definition umfasst auch all die Ausnahmen, die wir eben aufgelistet haben. Sie dürfte also allgemein genug und aussagekräftig genug sein, um in der Praxis sinnvoll einsetzbar zu sein.

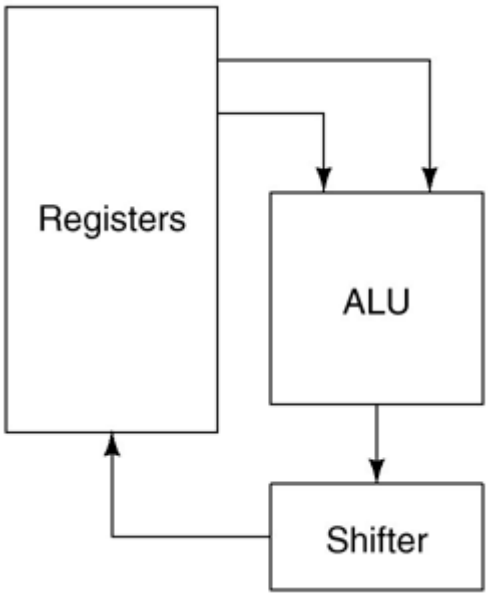
In den klassischen Programmiersprachen sind Module:

- Funktionen
- Blöcke
- Procedures

In den Objekt Orientierten Programmiersprachen sind Module:

- Objekte
- Methoden in den Objekten

Um die Wichtigkeit der Modularisierung zu verstehen, machen wir ein Hardware Beispiel:



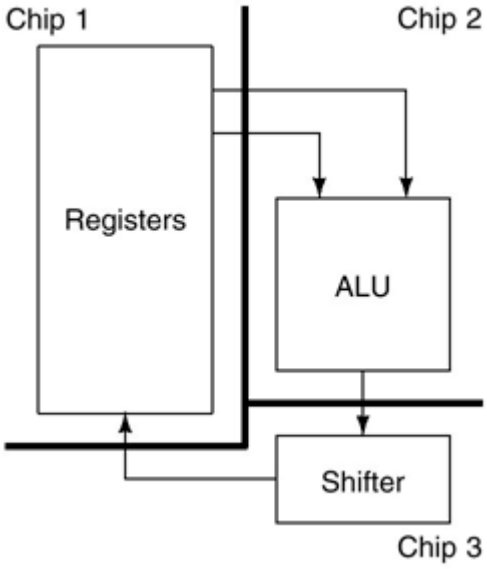
Unser Computer besteht aus einer ALU, einer Arithmetisch logischen Einheit, einigen Registern und Schieberegister.

Wir haben das System bereits in mehrere Module zerlegt, weil diese in der Regel in der Praxis auch so zusammen gefasst werden.

Stellen wir uns das gleiche System mit Hilfe von NAND und NOR Gattern realisiert vor. Unser Entwickler hat auch noch nicht von vollständigen Boole'schen Systemen gehört und weiss deswegen nicht, dass jede Boole'sche Funktion mit Hilfe von NAND und NOR darstellbar ist. Der Entwickler setzt deswegen auch noch AND und OR Gatter ein.

Die Firma beschliesst die drei Blöcke jeweils in einem Chip zusammen zu fassen und lässt diese Chips in Serie herstellen.

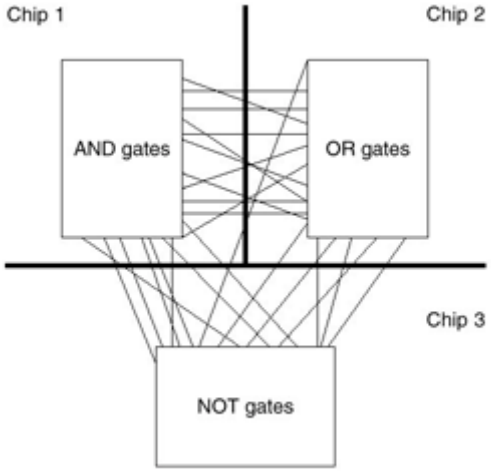
Unser Rechner sieht nun wie folgt aus, mit Hilfe der drei Chips implementiert:



Am Design hat sich nicht viel geändert. Aber jetzt haben wir ein Chipset und somit unsere Modularisierung in Silikon eingebrannt.

Unser Entwickler ist mit den Chips nicht zufrieden. Er schlägt eine neue Aufteilung auf drei Chips vor, allerdings sollen alle AND Gatter in ein Chip, alle OR Gatter ins zweite Chip und alle NOT Gatter ins dritte Chip.

Wie toll sieht unser Design jetzt aus?



DAS ist doch ein tolles Design!

Oder etwa nicht?

Was gefällt Ihnen nicht daran?
Die Gatter sind nun doch sauber geordnet!

Alles ist sehr logisch angeordnet!

Leider hat sich heraus gestellt, dass das Design noch erweitert werden muss.

Es soll zusätzlich eine Floating Point Unit eingebaut werden. Zudem sind die Register zu langsam und sollen durch schnellere ersetzt werden.

Welches Design ist änderungsfreundlicher ausgelegt?

Kommen wir zurück zur Modularisierung der Software:
- Die Gesamtfunktionalität ist festgelegt
- Die Zerlegung in einzelne Module ist so vor zu nehmen, dass eine einfache Wartung möglich ist
- Das System soll auch erweiterbar sein
- Die Wartung soll möglichst einfach sein

Unsere Software Module haben viel zu tun mit den Chips, die wir eben gesehen haben.

Stevens, Myers und Constantine haben in ihrer Arbeit den Begriff des Composite/Structured Design (C/SD) eingeführt. Wesentlich für die Zerlegung ist dabei, dass die Wartungskosten minimiert werden sollen. Die Wartungskosten sind sicher immer dann minimal, wenn wir klare, einfache Strukturen wie beim ersten Chip- Design hätten.

Myers führte die Begriffe *module cohesion* als Grad der Wechselwirkung innerhalb der Module, und *module coupling* als Grad der Wechselwirkung zwischen den Modulen ein.

Betrachten wir die Struktur unserer Module etwas genauer, dann erkennen wir, dass ein Module drei *Gesichter* hat:
- Die *action* Seite, die beschreibt, was ein Modul tun kann, also das Modulverhalten
- Die *logic* Seite, die beschreibt, wie ein Modul seine Aktionen ausführt
- Die *context* Seite, die beschreibt, wie ein Modul typischerweise eingesetzt wird

Im C/SD ist der Name des Moduls eigentlich mit der Aktionsseite gekoppelt, ist also wichtiger als seine Logik.

6.2. Cohesion / Kohäsion

Nach Myers Veröffentlichungen in Buchform [Composite/Structured Design, 1978] wurden seine Thesen intensiv untersucht und mit viel Theorie gezeigt, dass etwas Wahres an seinen Aussagen ist.

Myers hatte ursprünglich sieben Stufen der Kohäsion definiert:
1. Coincidental cohesion (schlecht)
2. Logical cohesion
3. Temporal cohesion
4. Procedural cohesion
5. Communicational cohesion
6. Informational cohesion
7. Functional cohesion (gut)

} diese gehören zusammen

Die Theoretiker zeigten, dass die Stufen 6 und 7 zusammen gefasst werden können.
Wir sollten solche "Metriken" nicht zu ernst nehmen. Wichtig ist weniger die absolute Positionierung; wesentlich wichtiger ist, zu wissen, wie zwei Modularisierungen relativ zueinander einzustufen sind.

6.2.1. Coincidental (Zufällige) Cohesion

Definition:
Ein Modul zeigt dann "coincidential cohesion", wenn er mehrere vollständig voneinander unabhängige Aktivitäten ausübt.

Beispiel:
```
{
        System.out.println("Parameter Eingabe:"); int iVar = 21; float fParam=2; if(bVar)
{bArgument}
}
```
Fragen:
Was haben die Anweisungen auf der obigen Zeile miteinander zu tun?
Antwort:
Nichts!

Warum wurde der Modul den so unsinnig geschrieben?
Ein Grund mag sein, dass die Firma eine Vorschrift erlassen hat, dass Module nicht mehr als 35 Zeilen haben dürfen, eine unsinnige Forderung!

Warum ist Coincidential Cohesion so schlecht?
Module mit coincidential cohesion verursachen viele Probleme:
- In der Wartung
- In der Weiterentwicklung
- Bei einer allfällig geplanten Wiederverwendung

[1]

Einen Modul mit nur coincidental cohesion zu verstehen ist äusserst anstrengend. Bei einem Modul würde man eine bestimmte Grundausrichtung auf eine bestimmte Aufgabe erwarten. Dies fehlt hier vollständig.

Diese Module sind schlicht nicht wieder verwendbar. Die obigen Programmzeilen werden kaum je wieder in einem Programm auftauchen! Hoffentlich auch nicht...

6.2.2. Logical Cohesion

Definition:
Ein Modul zeigt logische Kohäsion, falls er mehrere (logisch) zusammenhängende Aktion ausführt.

Beispiel:
public double dArithmetik(double arg1; double arg2, opCode) {
switch(opCode)

Weitere Beispiele:
Beispiel 2
Ein Modul wurde so entwickelt, dass ALLE I/Os mit Hilfe dieses Moduls erledigt werden können. Wenn neue Geräte entwickelt werden, dann wird dieser Modul angepasst und man ist fertig!
Klingt doch gut....

Beispiel 3
Ein Modul umfasst ALLE Operationen, die mit einer Datei zusammen hängen:
- Lesen eines Datensatzes
- Schreiben eines Datensatzes
- Mutieren eines Datensatzes

Klingt doch gut ....

Beispiel 4
Ein Modul im IBM Betriebssystem mit logical cohesion, umfasste 13 unterschiedliche Operationen und 21 Datenelemente.
Na ja, das waren noch Zeiten...

Ein Module mit logical cohesion verursacht in der Regel zwei Probleme:
- Die Schnittstelle des Moduls ist schwer verständlich
- Im schlimmsten Falle wird Programmcode für mehrere Aktivitäten eingesetzt, geteilt. Änderungen für eine Funktion führen dadurch zu Problemen bei andern Funktionen.

[1] Sie finden Java Programme am Ende des Skriptes

Bemerkungen zu den obigen Beispielen:

1. Die Beschreibung des opCodes ist intern im Modul und extern bekannt. Eine saubere Kapselung ist schwierig. Wenn lediglich ein switch() { case i : ...break; ... default: ...break} Block existiert, wie in unserem Beispiel, dann ist die Wartung noch einfach und auch übersichtlich. In echten Beispielen sehen die Module in der Regel viel komplexer aus.
2. Im schlimmsten Fall besteht der Modul aus einem switch Statement mit klar definierten cases, die durch Erweiterungen umgestellt werden müssen:
   der Programmierer möchte alle Disk I/O Routinen beieinander haben und numeriert deswegen im Modul die switches um. Die alten Programme stürzen dadurch nicht ab, aber das Ergebnis stimmt nicht mehr! Die Fehlersuche wird nicht einfach:
   der Modul funktioniert ja korrekt!

Beispiel:

```
class arith {

    static public void main(String args[]) {
        double x1, x2, x3;
        x1=1.718;
        x2=2.914;
        int oper=0;
        x3=arithmeticExpr(x1, x2, oper);
        System.out.println("x1="+x1+"; x2="+x2+"; opcode="+oper);
        System.out.println("x3="+x3);
    }

    public static double arithmeticExpr (double a1, double a2, int opcode) {
        switch(opcode) {
            case 0: return (a1*a2);
            case 1: return (a1-a2);
            case 2: return (a1-a2);
            default: return (0);
        }
    }
}
```

Zu beachten: Dieses Beispiel ist de facto auf einem höheren Level (auf welchem?), da die Operationen auf den jeweils gleichen Datenelementen geschehen.
Ein Beispiel, welches echt auf diesem Level wäre, würde neben unterschiedlichen Operationen auch noch unterschiedliche Daten verwenden.

6.2.3. Temporal Cohesion

Definition:
Ein Modul zeigt temporäre Kohäsion, falls mehrere Aktivitäten zeitlich zusammen ausgeführt werden.

Beispiel:
Viele Leute gehen am Samstag einkaufen und bringen den Müll weg, bringen Leergut zurück und Spezialabfälle zur Spezialdeponie.

Es macht aber wenig Sinn den Abfall und den Bäcker im selben Modul zu beschreiben und zu behandeln.

Beispiel:
"Initialisierung " in klassischen Sinnen:

- Alle in den Programmen verwendeten Dateien werden mit Hilfe *einer* Routine geöffnet
- Die Dateien werden nicht alle gleichzeitig benötigt, aber in einem bestimmten Job

Schritt	Aktivität
1	open(LOG,">$logfile);
2	open(IN,"<$Updates");
3	open(OUT,">$Master_New");
4	open((TEMP,"tempfile");
5	open(MASTER,"<$Master_old");
6	open(ERROR,">$errorFile");

In diesem Falle sind die Aktivitäten, die in einem einzigen Modul zusammen gefasst wurden, eher ablaufmässig korreliert, aber eigentlich eben eher nicht sinnvoll.

Das obige Beispiel verwendet die Perl Syntax.

Selbsttestaufgabe:
Schreiben Sie ein Java-Programm, bei dem alle Klassenvariablen in einem eigenen Block initialisiert werden.
Hinweis: da wir I/O in Java noch nicht besprochen haben, kann das obige Beispiel NICHT in Java übernommen werden.

Diese Art Kohäsion ist ungenügend, wieder einmal aus dem gleichen Grund:

- Wenn wir Änderungen an unseren Programmen durch führen, muss dieser Modul sicher überholt werden
- Die Software Entwicklung kann nicht unabhängig von diesem Modul weitergehen.

Wenn zum Beispiel eine weitere Datei hinzu kommt, dann müssten alle Module auf die Anpassung des Dateiverwaltungs-Modules warten.

6.2.4. Procedural Cohesion

Ein Modul zeigt prozedurale Kohäsion, falls der Modul eine Serie von Aktionen ausführt, die zeitlich zusammen hängen.

Beispiel:
Lies Artikelstamm-Datensatz
Schreib Lieferschein-Datensatz

In beiden Datensätzen werden Artikeldaten bewegt. Aber die Kohäsion ist noch nicht optimal!

6.2.5. Communicational Cohesion

Ein Modul zeigt kommunikative Kohäsion, falls der Modul eine Serie von Aktivitäten ausführt, die zeitlich zusammen hängen und mit denselben Daten ausgeführt werden.

Beispiel:
Lies Artikelstamm-Datensatz
Schreib Artikelstamm-Datensatz in Archivierungs-Datei

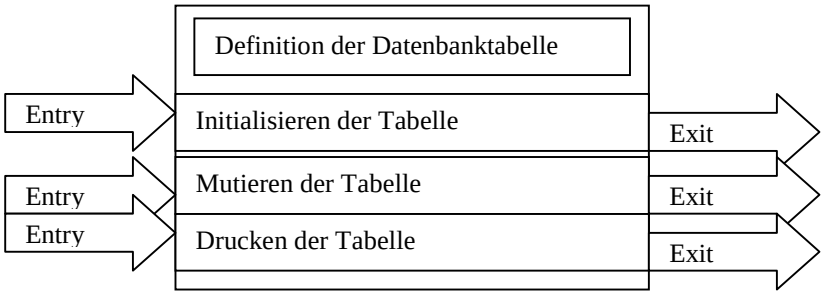
Im Vergleich zur prozeduralen Kohäsion haben wir eine deutlich bessere Situation, da alle prozeduralen Schritte auf ein und die selben Daten ausgerichtet sind.

6.2.6. Informational Cohesion

Ein Modul zeigt Informations-Kohäsion, falls der Modul mehrere Aktionen ausübt, die alle unabhängig voneinander sind, über einen eigenen Entry Point angesteuert werden können und die gleichen Daten betreffen.

Beispiel:
Eine Datenstruktur beschreibt den Aufbau einer Datei.
In einem Modul werden die Felder definiert (Header),
dann folgt die Definition verschiedener Operationen:

- Initialisieren der Datei, bezw der Datenbank-Tabelle und EXIT des Moduls nach der Initialisierung
- Mutieren der Datei bezw. der Datenbank-Tabelle und EXIT
- Drucken der Datei bezw bestimmter Datensätze und EXIT



In diesem Beispiel werden keine Prinzipien der strukturierten Programmierung verletzt:

- Jede Operation hat ihren eigenen Entry und Exit Punkt
- Jede der Operationen agiert auf den selben Daten

Ein Modul mit informational cohesion implementiert im wesentlichen einen abstrakten Datentyp (ADT : abstract data type).

Objekte sind im Wesentlichen Instanzen abstrakter Datentypen, oder eben auch ein Modul mit informational cohesion, somit ist informational cohesion optimal für Objektorientierung.

6.2.7. Functional Cohesion

Ein Modul zeigt funktionale Kohäsion, falls der Modul genau eine Aktion ausübt.

Beispiele:

- Lies_den_Temperatursensor
- Write(LOGFILE);
- CalculateSurfaceTension
- Sin(x)

Ein Modul mit funktionaler Kohäsion kann vielfach wieder verwendet werden, falls die Funktion von andern Modulen benötigt wird (Beispiel : Sinus Funktion).

Ein Modul mit funktionaler Kohäsion ist insbesondere auch wirtschaftlich sinnvoll, sofern er gut dokumentiert und ausgiebig getestet wurde.

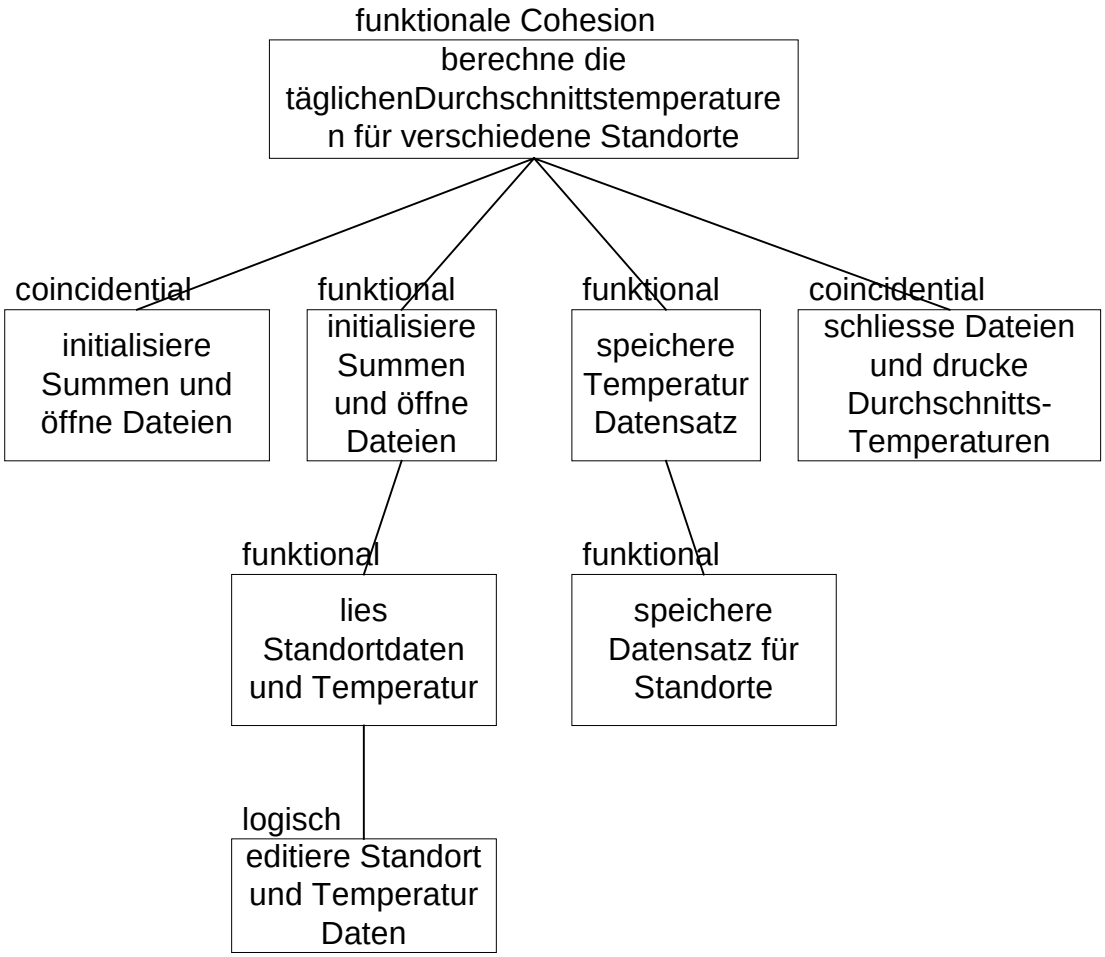
Die Wartung eines Moduls mit funktionaler Kohäsion kann auch leichter gewartet werden, und vor allem zentral. Ein Fehler in einem solchen Modul kann in der Regel besser lokalisiert werden, als ein Fehler, in einem Modul, bei dem zuerst heraus gefunden werden muss, in welchem Funktionsblock er sich befindet.

Erweiterungen solcher Module sind in der Regel einfacher als bei andern Modulen.
Warum?
In vielen Fällen kann einfach der Modul durch einen leistungsfähigeren Modul ersetzt werden.

Ein Hardware Beispiel für funktionales Design (Module mit funktionaler Kohäsion) haben wir bereits ganz am Anfang kennen gelernt.

6.2.8. Ein Kohäsions Beispiel

Im ersten Modul werden die Zähler zurück gesetzt und alle Files geöffnet. Die Frage ist, ob dieser Modul zeitliche oder zufällige Kohäsion zeigt?



Wenn immer zwei Kohäsions-Level möglich sind, wie oben, dann wird in der Regel der möglichst tiefe Level angegeben. Allerdings ist zu beachten, dass immer dann, wenn der Modul Funktionen enthält, die alle auf die gleichen Daten zugreifen, ein höherer Level resultiert als im Falle auch noch heterogener Daten.

6.3. Kopplung

Im Falle der Kohäsion haben wir die INNERE Struktur der Module untersucht und versucht Kriterien zu finden, welche uns erlauben etwas über die Qualität der Modularisierung auszusagen.

Bei der *Kopplung* definiert man analoge Level. Diese betreffen jedoch jetzt den Austausch von Informationen zwischen Modulen.

Bei der Kopplung werden 5 Kopplungsebenen unterschieden:

5	Data Coupling	(gut)
4	Stamp Coupling	
3	Control Coupling	
2	Common Coupling	
1	Content Coupling	(schlecht)

6.3.1. Content Coupling

Zwei Module zeigen inhaltliche Kopplung, falls ein Modul direkt Bezug nimmt auf den Inhalt eines anderen Modules.

Beispiel 1:
Bei der Ausführung des Modules P wird zu einem Label in Modul Q verzweigt und nach Ausführung eines Blockes in Modul Q wieder zurück gesprungen.

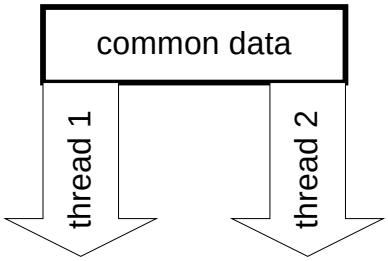
Beispiel 2:
Bei der Ausführung des Modules P werden Daten aus dem Modul Q modifiziert.

Konsequenzen:
Die beiden Module sind offensichtlich eng gekoppelt. Jede Änderung in Modul Q kann zum Absturz von Modul P Führen, sofern kritische Teile verändert werden.

Selbsttestaufgabe:
Versuchen Sie ein Java Beispiel für inhaltliche Kopplung zu finden.
Sie werden hoffentlich einige Mühe haben. Java kennt kein GOTO, aber mit dem BREAK Statement kann man ähnlichen Unfug machen.

6.3.2. Common Coupling

Zwei Module sind common gekoppelt, falls beide Module Zugriff zu den selben globalen Daten haben.



Diese Art der Kopplung geschieht schneller als einem lieb ist: wenn Sie in Java Variablen als public deklarieren, dann impliziert dies automatisch, dass ein Zugriff auf diese Variablen aus verschiedenen Modulen erfolgen kann.

Common Coupling hat unangenehme Seiteneffekte:
Prozedurale Abläufe sind eigentlich nicht mehr nachvollziehbar.

Beispiel:
Falls Modul_1 und Modul_2 Zugriff auf die boolean Variable piV haben, dann ist das folgende, eigentlich sonst absolut banale Programmstück nicht mehr deterministisch:

```
public boolean piV=true;
While (piV) {
        if (bV1) threadModule(piV);
        if (bV2) threadModule(piV);
}
```

Was in Thread 1 und Thread 2 passiert sei dahin gestellt. Wenn wir zwei faire Thread haben, dann können wir nicht einmal voraussagen (deterministisch), wann welcher Thread CPU Zeit erhält und ausgeführt wird.

Da beide Threads die public Variable piV verändern können, ist unklar, wieoft die WHILE Schlaufe ausgeführt wird. Es können also Seiteneffekte auftreten.

Eine weitere Konsequenz kann sich in der Wartungsphase ergeben: Änderungen in Modul_1 oder Module_2 speziell den Zugriff auf die globalen Variablen betreffend, können globale Auswirkungen haben und zu Änderungen in anderen Modulen führen, also ein sehr unerwünschtes Verhalten zeigen.

Ein Wiederverwenden der Module ist eher unwahrscheinlich, da die Module ja nicht unabhängig voneinander existieren.

Im weiteren werden bei solchen Modulen mehr Daten öffentlich sichtbar als unbedingt nötig, im Widerspruch zu Parnas' Prinzip.

6.3.3. Control Coupling

Zwei Module sind kontrollmässig gekoppelt, falls ein Modul ein Kontrollsteuerelement an den zweiten Modul übermittelt; das heisst, dass der erste Modul die Logik des zweiten Modules explizit kontrolliert.

Dies kann zum Beispiel so geschehen, wie bei der logischen Kohäsion.

Falls ein Modul P einen anderen Modul Q aufruft, und Q an P eine Fehlermeldung zurück gibt, dann besteht zwischen den zwei Modulen eine Kontroll-Kopplung, falls der erste Modul auf Grund dieser Meldung eine Fehlerbehandlung durch führt.

Die Konsequenzen einer Kontroll-Kopplung sind in der Regel:
- Schlechte Wartbarkeit
- Kaum Wiederverwendbarkeit

In der Regel zeigen Module mit Kontroll- Kopplung auch logische Kohäsion und folglich treten auch die selben Probleme auf.

6.3.4. Stamp Coupling

In einigen Programmiersprachen kann man lediglich einfache Variablen als Parameter verwenden. In anderen Programmiersprachen kann man auch ganze Datenstrukturen übergeben.

Stamp Coupling besteht genau dann, wenn Datenstrukturen als Parameter übergeben werden, aber lediglich Teile davon echt benötigt werden.

Ein typisches Beispiel ist die Übermittlung einer Adresse, wobei gleichzeitig alle weiteren Personaldaten übermittelt werden, sicher ein völliger Unsinn!

Seiteneffekte bei Stamp Coupling sind unter Umständen, dass Teile der Datenstruktur in einem Modul verändert werden, wobei dieser Modul eventuell dazu nicht berechtigt ist.

Es gibt auch viele Fälle wo es Sinn macht, ganze Datenstrukturen als Parameter zu verwenden. Stamp Coupling tritt häufig in C++ bei der Verwendung von Pointern auf.

6.3.5. Data Coupling

Zwei Module sind Daten gekopplet, falls alle Parameter bei der Kommunikation homogen sind dh. entweder einfache Parameter oder Datenstrukturen, bei denen jedes Element benutzt wird.

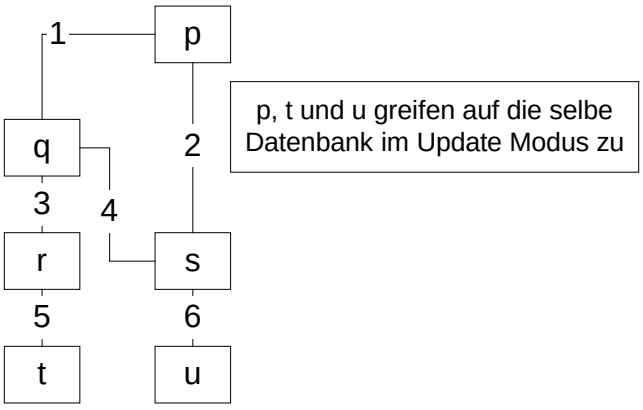
Beispiel:
1. Multiplikation zweier komplexer Zahlen

Z1 und Z2, z3 seien Komplexe Zahlen. Z3=Multiplikation(z1,z2) führt zu einer Datenkopplung

2. Bestimme_job_mit_der_hoechsten_prioritaet(queue)

Bewertung dieser Art der Kopplung:
Datenkopplung ist in der Regel eine sehr brauchbare Kopplungsart, da die Einflussnahme klar abschätzbar ist und die unabhängige Wartung und Weiterentwicklung der Module weitestgehend gewährleistet ist.

6.3.6. Kopplungsbeispiel



Interfacebeschreibung:

Nummer	Eingabe	Ausgabe
1	FlugzeugTyp	StatusFlag
2	-	Stückliste
3	FunktionsCode	-
4	-	Stückliste
5	SerieNummer	Teilehersteller
6	SerieNummer	Teilebezeichnung

Beispiel:
Modul P ruft Modul Q auf mit Hilfe des Interfaces 1.
Module P sendet dazu dem Modul Q eine Anfrage betreffend eines "FlugzeugTyp"'s (Interface 1).
Modul Q sendet an Modul einen Status zurück.

Daraus lässt sich für jede der Verbindungen die Kopplungsart bestimmen. Als Ergebnis erhalten wir folgende Matrix:

	q	r	s	t	u
p	Data		Stamp / Data	Common	Common
q		Control	Stamp / Data		
r				Data	
s					Data
t					Common

Betrachten wir einige Beispiele, damit die Tabelle verständlicher wird:
Datenkopplung zwischen den Modulen P und Q:
Zwischen P und Q wird gemäss Interface 1 lediglich ein Datenelement ausgetauscht.

Das Gleiche gilt für die Kopplung zwischen R und T (Interface 5).

Bei der Kopplung zwischen P und S mit Hilfe von Interface 2 besteht Datenkopplung, falls alle Elemente der Stückliste benötigt werden, sonst haben wir Stamp Kopplung.

Die Kopplung zwischen Q und R mit Hilfe des Interfaces 3 ist Control Coupling, weil eine Funktion übergeben wird.

Wie kommt man zur Common Coupling Aussage?
Diese folgt aus der Bemerkung in der obigen Skizze (gleiche Datenbank).

Nachdem wir nun Kriterien für die Güte eines Designs haben, stellt sich die Frage: wie gelangt man zu einem guten Design?

Darauf kommen wir noch zurück!

Randbemerkung
Die Übergabe einzelner Datenfelder an einen Modul ist eventuell langsamer als die Übergabe eines kompletten Datensatzes. Der Nachteil des Datensatzes ist : wir geben zu viele Daten mit, erhöhen also die Abhängigkeiten der Module.
Ratschlag: wann immer möglich sollte man auf Optimierungen verzichten, falls diese die Kopplung erhöhen.
Falls die Optimierung unbedingt benötigt wird, dann sollten Alternativen geprüft werden.
Dies bezeichnet man als das erst und das zweite Gesetz der Optimierung nach Don Knuth:
First Law of Optimization : Don't!
Second Law of Optimization : Not yet!

6.4. Daten Kapselung

Überlegen Sie sich, wie die Jobverwaltung eines Grossrechner-Betriebssystems aussehen könnte.

Sie möchten Jobs verwalten, die an das Betriebssystem zur Bearbeitung abgegeben wurden:

- Druckjobs
- Batchjobs
- ...

Jeder Job wird , neben anderem, gekennzeichnet durch eine Jobnummer und eine Priorität. Als Prioritätslevel seien definiert:

- tief
- mittel
- hoch

Der Scheduler teilt den Jobs jeweils ein Zeitintervall zu und sorgt dafür, dass die verschiedenen Warteschlangen "fair" behandelt werden.

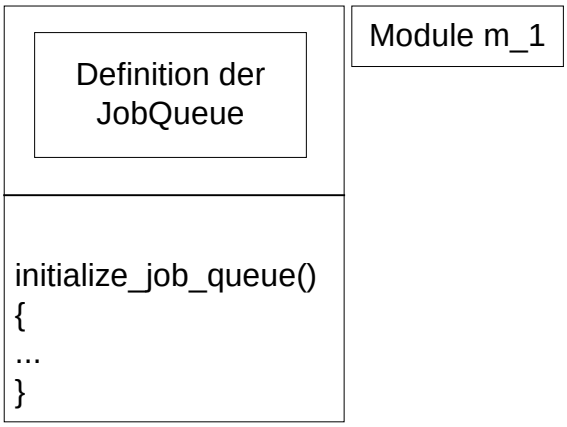
Damit die verschiedenen Jobs jeweils pro Queue sauber getrennt werden können, bietet es sich an, je eine Queue zu definieren.

Die Job Queue muss initialisiert werden, Jobs müssen hinzu gefügt werden können, Jobs müssen auch wieder entfernt werden können und die Queue auf den neusten Stand gebracht werden.

Also: JobQueue Methoden:

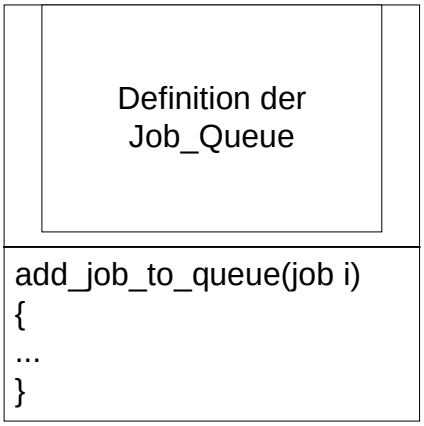
- initialisieren der Job Queue
- Job der Queue hinzufügen
- Job aus der Queue entfernen

Job Scheduler : ist dafür verantwortlich, dass all diese Methoden ausgeführt werden, in einer sinnvollen Reihenfolge.Der Einfachheit halber betrachten wir den Spezialfall von Batch Queues, nicht generellen Queues.Schematisch können wir unser System in folgende Module zerlegen:

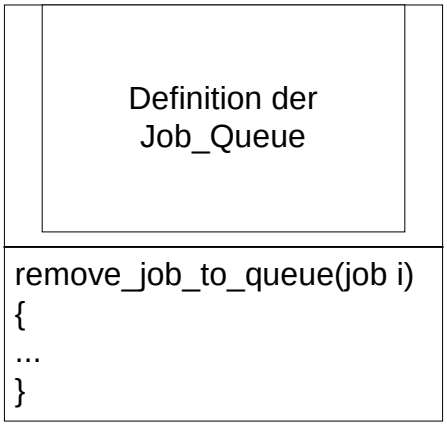


Und hier schematisch die restlichen Methoden und der Scheduler:

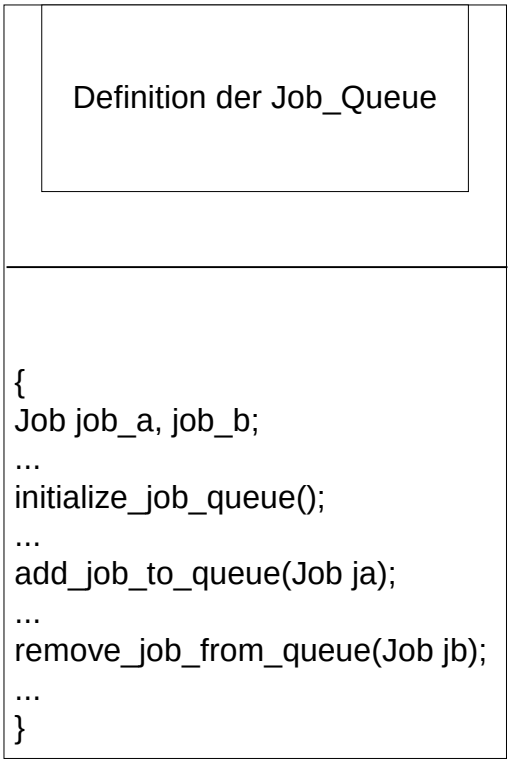
m_2



m_3



m_123

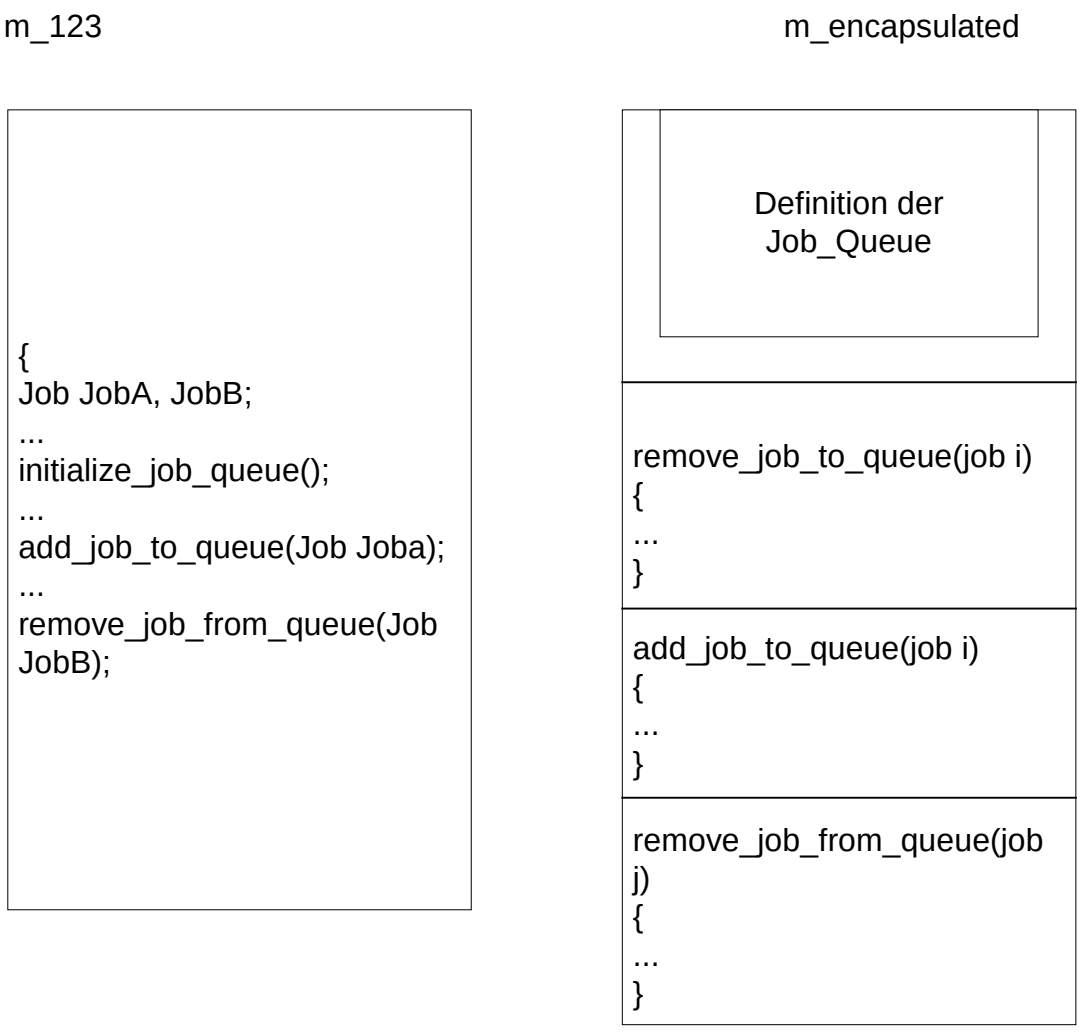


Alle Module sind nur schematisch dargestellt. Eine vollständige Java Implementation ist relativ leicht möglich, wie wir noch sehen werden.

Bewerten wir nun die Module in Bezug auf Kohäsion und Kopplung:
Die Module zeigen wenig Kohäsion, weil die Aktivitäten über viele Module verteilt sind.

Falls wir uns entschliessen sollten, die Implementation der Job Queue zu ändern, dann müssten wir jeden Modul abändern.

Betrachten wir nun eine alternative Modularisierung:



Der Modul auf der rechten Seite zeigt jetzt offensichtlich informational cohesion:

- alle Aktivitäten werden auf den selben Daten ausgeführt.
- Alle Aktivitäten haben einen eindeutigen Entry und einen eindeutigen Exit Punkt

Man bezeichnet eine solche Implementation auch als *Data Encapsulation*, oder auf deutsch "Daten-Kapselung".

Welche Vorteile ergeben sich aus der Datenkapselung?

6.4.1. Datenkapselung und Produkt Entwicklung

Data Encapsulation ist ein Beispiel für die *Abstraktion,* bei der die Datenstrukturen und die darauf ausgeführten Operationen zusammen gefasst werden. In unserem Beispiel wurde die Definition der Job Queue und die Operationen initialize, add_job, remove_job als ein Modul zusammen gefasst.

Der Designer und der Entwickler bewegen sich dadurch auf einem höheren Level, lösen also de facto allgemeinere Probleme, nämlich das Problem der Job-Verwaltung, im Gegensatz zum Record-Level, auf dem die einzelnen Datensätze verwaltet werden müssen.

Auch hier kann man eine stufenweise Verfeinerung erkennen:
Module werden zuerst auf einer allgemeinen Stufe, der Stufe der Queues definiert,
dann müssen die Module implementiert werden, mit Hilfe von Records oder anderen Datenstrukturen.

Beim Design setzt der Entwickler voraus, dass bestimmte Konzepte auf einem tieferen Level existieren. Später kümmert er sich um deren Realisierung.

Auf den höheren Ebenen betrachten wir vor allem das Verhalten, weniger die Implementierung der Datenstruktur.

Die Strukturierung von Informationssystemen mit Hilfe von Ebenen geht zurück auf Dijkstra, der an einem Betriebssystem gezeigt hat, wie dieses Design Pattern systematisch benutzt werden kann, um ein ganzes Betriebssystem zu entwerfen.

Dijkstra definierte dabei folgende Level:
- Level 5 : Computer Operator
- Level 4 : Benutzer Programme
- Level 3 : ...
- Level 1 : Memory Segment Controller
- Level 0 : Clock-Interrupt Handling und Prozessor Allocation

Analog wurden von Tanenbaum mehrere Ebenen für die Rechner-Organisation definiert:
- Level 5 : Problem-Orientierte Programmiersprache
- Level 4 : Assembler
- Level 3 : Betriebssystem / Operating System
- Level 2 : Maschinensprache
- Level 1 : Mikroprogramme
- Level 0 : Digitale Logik

Auch hier wird das Design Pattern "Layer" angewendet.

Bei der Rechner Organisation hat man die selbe Hoffnung wie bei der Data Abstraction : Änderungen auf tiefere Leveln sollten sich nicht auf die oberen Level, sondern möglichst nur auf die benachbarten, falls überhaupt, auswirken.

In unserem Warteschlangen Beispiel haben wir zwei Daten-Abstraktionen kombiniert: *Data Abstraction* und *Procedural Abstraction.*

Abstraktion kann also definiert werden als eine Methode, der Stufenweisen Verfeinerung.

Kapselung dagegen, kann man als Zusammenfassung aller Aspekte einer realen Grösse verstehen.

Das Ziel von beiden ist es, Begriffe wie Warteschlangen analog zu sqrt() oder sin(...) zu verwenden.

6.4.2. Data Encapsulation und Produkt Wartung

Daten Kapselung ist aus der Sicht der Software-Wartung immer dann sinnvoll, wenn jene Module, die vermutlich öfters geändert werden müssen, klar isoliert werden, um dadurch die Propagation von Änderungen zu minimieren. Änderungen sollten also so wenig Module wie möglich betreffen.

Betrachten wir nun eine mögliche Implementation der Job Queue in Java mit Hilfe eines Arrays. Üblicherweise verwendet man starre Arrays bei der Entwicklung eines Software Packages, bevor man zu dynamischen Datenstrukturen übergeht (Listen). Dadurch vermeidet man, dass man die Komplexität der Algorithmen und der dynamischen Datenstrukturen gleichzeitig im Griff haben muss.

```
class JobQueue {
        // Instanz variablen : Problem wegen public data members werden später beseitigt
        public int queueLength;                                 // Länge der job Queue
        public int queue[ ] = new int[25];
        // maximal 25 Jobs (besser wäre eine dynamische Liste
        // aber der Übergang zur dynamischen Liste kann später erfolgen
        // statt 25 würde man sicher eine Klassenvariable public static ... definieren
        // public int queue ist, weil public COMMON (von überall her zugreifbar) also
        // schlecht!!

        // Methoden
        public void initializeJobQueue() {
        // leere Job Queue hat Länge 0
                queueLength = 0;
        }

        public void addJobToQueue(int jobNumber) {
        // Job der Warteschlange anhängen
                queue[queueLength] = jobNumber;
                queueLength = queueLength + 1;
        }

        void removeJobFromQueue(int jobNumber) {
        // setze jobNumber gleich dem Job am Anfang der Warteschlange
        // entferne den Job am Kopf der Warteschlange und schiebe die andern Jobs nach
        // Frage : stimmt diese Implementation?
                jobNumber = queue[0];
                queueLength = queueLength - 1;
                for (int k=0; k < queueLength; k++)
                        queue[k] = queue[k+1];
        }
} // class JobQueue
```

Da wir die Queue als **public** deklariert haben, wird diese Datenstruktur von überall her zugreifbar. Dadurch entstehen unnötige Komplikationen. Denn eigentlich wird diese Information nicht benötigt. Alles was man braucht, sind die Methoden.

Betrachten wir nun eine andere, universellere Implementation der Job Queue:
An Stelle des Arrays verwenden wir nun eine zweifach verhängte dynamische Liste.
Der einzelne Job wird dann durch einen Record wie unter skizziert dargestellt:

```
class JobRecord {
        // lineare Liste
        public int                      jobNo;          //      Anzahl Jobs (Integer)
        public JobRecord        inFront;        //      Job vor diesem
        public JobRecord        inRear;         //      Job nach diesem
}       // class JobRecord
```

Inwiefern ändert sich dadurch die Verwendung der Job Queue Datenstruktur?

Offensichtlich bleiben alle Methoden unverändert. Insbesondere wird immer noch wie vorher ein Integer als Job Identifier verwendet. Der Parameter der Methoden ändert sich also nicht.

```
class JobQueue {
        // zweifach gelinkte Liste
        public JobRecord        frontOfQueue;// Verbindung nach vorne
        public JobRecord        rearOfQueue;  // Verbindung nach hinten
        // Methoden
        public void initializeJobQueue() {
        //      leere Job Queue hat Länge 0
        //      initialisieren der Queue geschieht, indem der Vorgänger und der Nachfolger
        //      NULL sind
        }
        public void addJobToQueue(int jobNumber) {
        // Job der Warteschlange anhängen
        // Ablauf Skizze:
        /*      kreiere einen neuen JobRecord
                trage jobNumber in das entsprechende Feld ein (jobNo)
                Anfügen geschieht, indem
                rearOfQueue ins Front Feld eingetragen wird
                und das rearOfQueue null (nicht 0, sondern null als "leer") gesetzt wird
                einsetzen des neuen Records in die rearOfQueue
                */
        }

        void removeJobFromQueue(int jobNumber) {
        /*      setze jobNumber gleich dem Job (also jobNo) am Anfang der Warteschlange
                setze jetzt frontOfQueue gleich dem nächsten Record
                und setze das inFront Feld des ersten Record der Warteschlange auf null
         */
        }
} // class JobQueue
```

6.5. Abstrakte Datentypen

Die Implementation der Job Queue, die dynamische Liste plus alle Methoden, in einer Klasse, bezeichnet man als *Abstract Data Type* oder abstrakten Datentyp (ADT).

Abstrakte Datentypen können in der Regel sehr universell eingesetzt werden, deswegen wurden sie auf diese Art und Weise definiert und implementiert. Wir können dadurch die verschiedenen Job Queues als Instanzen des ADTs realisieren.

Die Instanzierung der Job Queue ändert sich bei der Verwendung der obigen Queue Implementation nicht:

```
class JobQueue
{
        // instance variables
        public int      queueLength;
        public int      queue = new int[25]; // same remarks as before

        // methods
        public void initializeJobQueue()
        {
                // no need to change ...
        }
        public void addJobToQueue(int jobNumber)
        {
                // body of method unchanged ...
        }
        void removeJobFromQueue(int jobNumber)
        {
                // body of method unchanged ...
        }
} // Job Queue
// Remark : not perfect (queue and queueLength are visible)
```

Auch *der Scheduler* braucht dadurch **NICHT** geändert zu werden:

```
class Scheduler{
        // ... hier steht sehr viel!
        public void queueHandler() {
                int                     jobA, jobB;
                JobQueue        jobQueueJ = new JobQueue();
                //      weitere Anweisungen
                jobQueueJ.initializeJobQueue();
                //      weitere Anweisungen
                jobQueueJ.addJobToQueue(jobA);
                //      weitere Anweisungen
                jobQueueJ.removeJobFromQueue(jobB);
                // weitere Anweisungen
        }
        //      ...
```

```
}       //      class Scheduler
```

Abstrakte Datentypen sind ein wichtiges Hilfsmittel, um wieder verwendbare Komponenten zu realisieren:

Betrachten wir ein weiteres Beispiel: es soll ein abstrakter Datentyp "Rationale Zahlen" definiert werden.
Hier eine erste Skizze:

```
class Rational
{
        public int      numerator;
        public int      denominator;

        public void  sameDenominator(Rational r, Rational s)
        {
                /* Code, welcher die zwei rationalen Zahlen
                gleichnamig macht
                */
        }
        public boolean equal(Rationa r, Rational s)
        {
        Rational        u,v;
                u=r;
                v=s;
        sameDenominator(u,v);
                return(v.numerator==u.numerator);
        }
        /*
        weitere methoden für die Addition, Subtraktion,
        Division und Multiplikation zweier rationaler Zahlen
        */
}
```

Dass der Zähler und der Nenner **public** sind, ist für den Moment eher schlechte Programmierpraxis. Aber im Sinne der stufenweisen Verfeinerung, werden wir dieses Manko noch beheben.

Abstrakte Datentypen unterstützen sowohl Daten-Abstraktion als auch Prozedure-Abstraktion (data abstraction , procedural abstraction [Abschnitt 6.4.1]). In der Regel wird ein abstrakter Datentyp auch kaum völlig verändert werden, sofern er sauber definiert wurde. Es kann höchstens passieren, dass weitere Methoden hinzu gefügt werden.

Somit sind abstrakte Datentypen im Sinne einer iterativen oder evolutionären Software Entwicklung sehr attraktiv.

Die "Erfinder" der Abstrakten Datentypen (Liskow und Guttag : *Abstraction and Specifikation in Program Development* The MIT Press, Cambridge, MA, 1986) haben eine weitere Abstration, die *iterative abstraction,* definiert. Dabei geht es um Folgendes: Iterationen können auf einer höheren Ebene definiert werden, ohne dass man sich um die konkrete Form der Iteration bereits Gedanken macht. Auf einem tieferen Level muss man die

Iteration dann mit den Standard-Kontrollstrukturen der spezifischen Programmiersprache implementieren.

6.6. Information Hiding

Alle drei Abstraktions-Typen, die wir im vorigen Kapitel kennen gelernt haben, sind "Instanzen" eines abstrakteren Konzeptes, dem *information hiding* oder Parnas' Prinzip

(Parnas : "Information Distribution Aspects of Design Methology" *Proceedings of the IFIP Congress* Ljubljana, Yugolsavia, 1971, pp. 339 - 344; "A Technique for Software Module Specification with Examples", *Communications of the ACM,***15**, (may 1972) pp 330-336; "On the Criteria to be used in decomposing Systems into Modules", *Communications of the ACM,* **15,**(December 1972) pp. 1053-1058).

Seine Ideen haben einen direkten Einfluss auf die Wartung der Software und somit auch einen hohen Einfluss auf die Lebenszyklus Kosten eines Software Paketes.

Bevor man ein Modul implementiert, erstellt man eine Liste mit möglichen Änderungswünschen. Dadurch vermeidet man, dass Änderungen später auftauchen, die die gesamte Architektur unseres Systems in Frage stellen.

Berücksichtigt man diese Unsicherheiten und kapselt unsichere Teile in separate Module, dann werden sich zukünftige Änderungen lokal begrenzen lassen.

Betrachten wir dazu unser Queue Beispiel:

In der ursprünglichen Form waren queue[] und queueLength als **public** deklariert. Unter diesen Umständen ist es zum Beispiel möglich, Warteschlangeneinträge wiefolgt durch zu führen:

```
...
highPriorityQueue.queue[7] = -2145;
..
```

Wir können also den Inhalt, die Werte der JobQueue fast beliebig ändern, ohne eine der Methoden des ADTs QUEUE zu verwenden.

In Java ist eine Korrektur dieses "Fehlers" leicht möglich: wir ersetzen einfach den Zugriffsmodifier **public** durch den Zugriffsmodifier **private.**

Hier das Ergebnis (bereits bekannt):

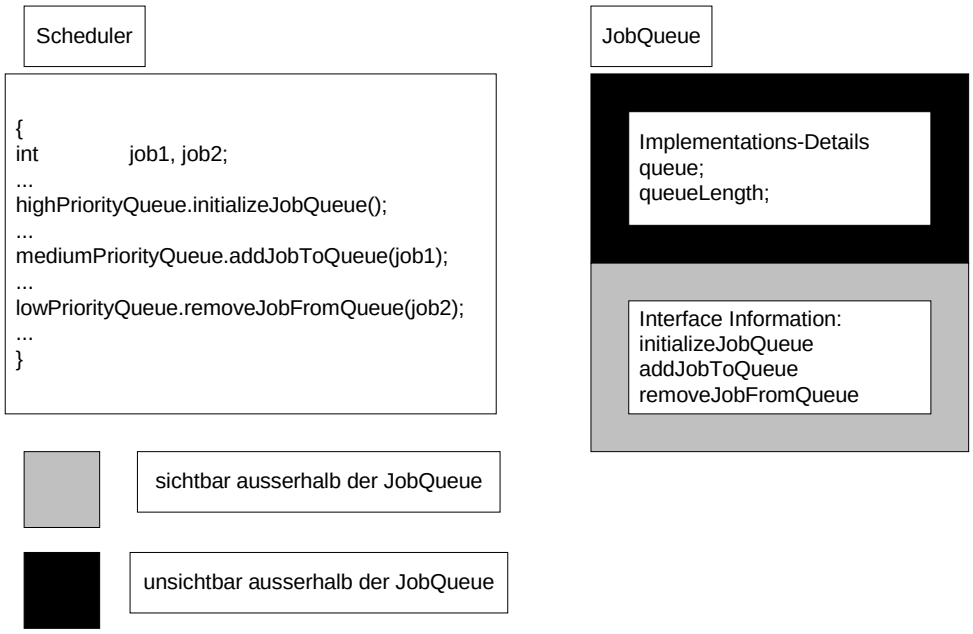
```
class JobQueue
{
        // instance variables
        private int queueLength;
        private int queue[] = new int[25]; // Laenge fix;

        // methods
        public void initializeJobQueue()
        {               //no changes
```

```
        }
        public void addJobToQueue(int jobNumber)
        {               // no changes
        }
        public void removeJobFromQueue(int jobNumber)
        {               // no changes
        }
}       // JobQueue : queueLength & queue inaccessible
```

Dadurch kann man in Java ADTs vollständig implementieren!

Zusammenfassung:



6.7. Objekte

Objekte sind nun der logische nächste Schritt. Objekte besitzen bereits alle Eigenschaften, die wir bisher kennen gelernt haben, wie Data Abstraction, Information Hiding. Allerdings kommt noch etwas hinzu.

Eine erste informelle Definition eines Objektes:
Ein Objekt ist eine Instanzierung eines Abstrakten Datentypes.

Ein Software Paket besteht also aus abstrakten Datentypen und Variablen (Objekten), die Instanzen der ADTs sind.

Nun haben wir aber bereits erwähnt, dass die obige Definition eines Objektes unvollständig ist.
Was fehlt denn noch?

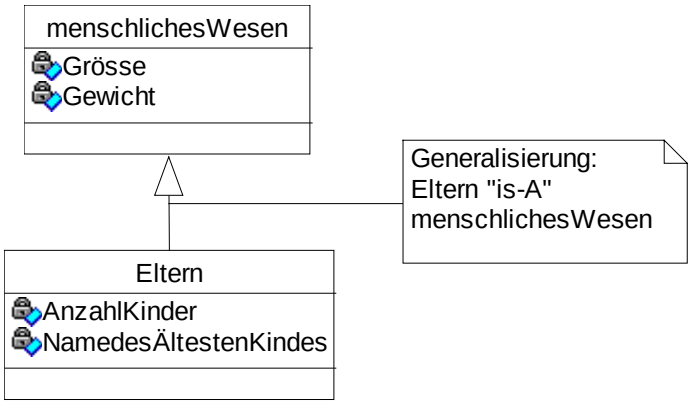
Die Vererbung!
Die Grundidee hinter der Vererbung ist, dass viele neuen Datentypen sich als "Erweiterungen" eines bereits definierten Datentypes auffassen lassen. Wir müssen bei der Definition eines Datentypes nicht immer von Grund auf neu anfangen.

In OO Programmiersprachen kann man *Klassen* definieren.

Ohne Anspruch besonders originell sein zu wollen, betrachten wir als Beispiel ein humanes Beispiel:

Wir definieren eine Klasse **menschliches Wesen** (oder humanBeing) und Hans ist eine Instanz davon. Wir haben also ein Objekt **Hans** . Jetzt definieren wir **Eltern** als *Subklasse* der Klasse menschlichesWesen. Die Klasse **Eltern** hat damit alle Eigenschaften der Klasse **menschliches Wesen** (hoffentlich!) und zusätzliche Eigenschaften (auch hoffentlich!).

Beispiele von Attributen, welche die Eltern Klasse zusätzlich hat:
- Anzahl Kinder
- Name des ältesten Kindes
- ...



Dieser Sachverhalt wird mit Hilfe der Unified Modelling Language (**UML**) wie folgt dargestellt:

Nicht sichtbar sind die vererbten Attribute, Gewicht, Grösse, ... (zum Beispiel Geschlecht).

In Java wird für die Vererbung das Schlüsselwort *implements* verwendet:

```
class HumanBeing
{
        private int age;
        private int height;
        Gender male, female;

        // declarations of Operations on HumanBeing
}       // class HumanBeing
class Parent extends HumanBeing
{
        private char nameOfOldestChild[ 20 ];
        private int  numberOfChildern;

        // declarations of Methods for Parent
}       // class Parent
```

Sofern wir ADTs und deren Verallgemeinerung, Objekte und Klassen einsetzen, dann wird offensichtlich, dass eine Äquivalenz besteht zwischen Daten und Prozeduren.

Betrachten wir ein Beispiel:
In klassischen Programmiersprachen wird der Zugriff auf ein Datenfeld einer Datenstruktur wie folgt spezifiziert:

Rec_1.data-field_3; // Zugriff auf Datenfeld 3 der Record Struktur 1

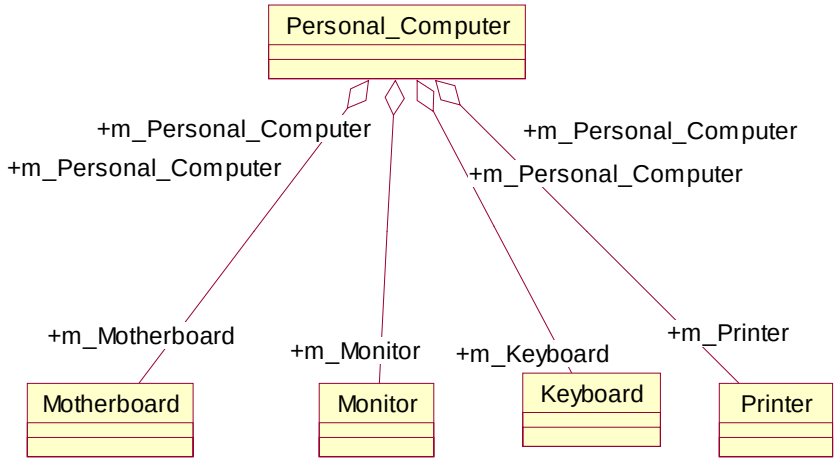
Objektorientiert geschieht der selbe Zugriff mit Hilfe einer Methode:

erstesObjekt.liesAttributA; // Zugriff auf ein Attribut mit Hilfe einer Methode

Objekte (und Klassen) verstärken somit das data hiding, führen somit zu universeller einsetzbaren Programmen.

Neben der Vererbung kennt man noch eine ganze Menge anderer grundlegender Konzepte im OO Bereich.
Eines ist die Aggregation, der starke Zusammenhalt mehrerer Klassen. Dies entspricht der realen Welt : ein PC besteht aus Mainboard, Monitor, Keyboard und eventuell einem Drucker

Und so sieht dies als UML Diagramm aus:



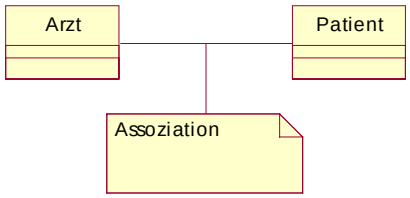
Dieses Diagramm wurde mit Hilfe von Rational Rose gezeichnet. Mit Hilfe von Rational Rose lässt sich auch Jaca Code generieren:

```
// Source file: Personal_Computer.java

public class Personal_Computer {
   public Motherboard m_Motherboard;
   public Monitor m_Monitor;
   public Keyboard m_Keyboard;
   public Printer m_Printer;

   Personal_Computer() {
   }
}
```

Eine weitere Art der Beziehung zwischen zwei Klassen ist die Assoziation, die Sie sicher aus der Theorie der Normalisierung im Fach Datenbanken her kennen.



Und so sieht der generierte Java Code aus:

```
// Source file: Arzt.java
public class Arzt {
   public Patient m_Patient;

   Arzt() {
   }
}
```

```
// Source file: Patient.java
public class Patient {
   public Arzt m_Arzt;

   Patient() {
   }
}
```

6.8. Vererbung, Polymorphismus und Dynamic Binding

Zuerst ein einfaches Beispiel:
Ein Computer hat verschiedene Peripherie- Geräte:

- Festplatte
- Floppy
- CD-ROM
- ...

Für jedes Gerät wurden bestimmte File- Operatoren definiert:

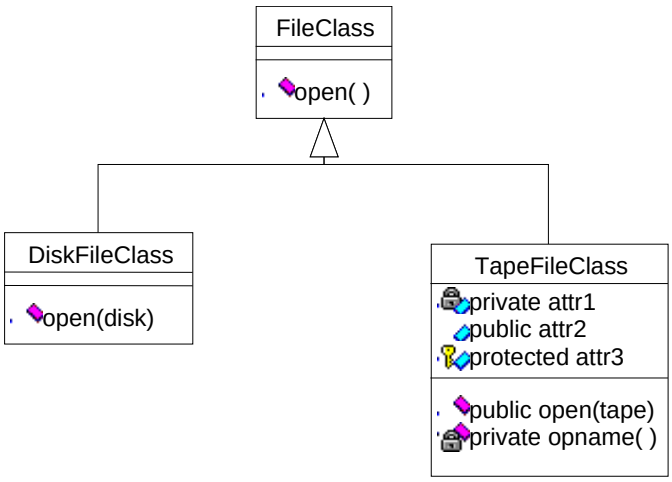
- openFile
- readFile
- updateFile
- deleteFile
- closeFile

Klassisch würde man für jedes Gerät diese Methoden definieren.

Objekt-Orientiert verwendet man *Polymorphismus:*

- Die Methoden heissen immer gleich
- Je nach Aufruf, wird zur Ausführzeit (also *dynamisch)* die korrekte Version der Routinen verwendet

In Java wurde ein zusätzlicher Klassentyp dafür geschaffen: **abstract.** Eine abstrakte Klasse definiert nur ein Gerüst der Klasse; die Klasse **muss** verfeinert werden.



Zur Laufzeit, beim Methodenaufruf, wird entschieden, welche Routine / Methode konkret verwendet wird.

Polymorphismus hat auch Nachteile:

- Es kann erst zur Laufzeit entschieden werden, welche Methode konkret ausgeführt wird
- Das Testen einzelner Methoden kann sich als recht schwierig erweisen
- Wartung wird in der Regel erschwert

6.9. Cohesion und Coupling bei Objekten

Objekte sind letztlich auch Module. Die Konzepte "Coupling" und "Cohesion" lassen sich somit auch auf Objekte anwenden.

Gibt es spezifische Objekt Orientierte Ausprägungen von Coupling und Cohesion?

Betrachten wir zuerst Cohesion:
Ein Objekt erhält seine Funktionalität auf zwei Arten:

- Durch Vererbung
- Durch "Erweiterung"

Für die Definition der Cohesion ist der Ursprung der Funktionalität unwichtig. Vererbung hat somit keinerlei Einfluss auf die Cohesion.

Betrachten wir nun die Kopplung:
Ein Objekt erhält seine Kopplung auf zwei Arten:

- Durch Vererbung
- Durch "Erweiterung"

Gefährliche Seiteneffekte treten immer dann auf, wenn Variablen und Methoden als **public** definiert wurden, da dadurch der Zugriff aus andern Routinen, Modulen und Methoden ermöglicht wird.

Coupling global betrachtet, ist aber unabhängig vom Ursprung. Ein typischer OO Coupling Level ist somit auch nicht nötig.

6.10. Aufgaben

1 Erklären Sie wie das Konzept des Modules in einer Programmiersprache Ihrer Wahl implementiert wird.

2 Bestimmen Sie die Kohäsion folgender Module:

```
a){
     ReadTape(record);
     CheckPaityBit(record);
 }
b) {
    editSalary(record):
   }
c) {
   calculateFastFourierTransform(xyz);
  }
d) measurePreasure(tank);
   soundAlarm(aBit);
```

3 Sie sind für die Software Entwicklung einer High Tech Firma zuständig. Ihr Chef bittet Sie, zu untersuchen, wie die Wiederverwendung der Module erhöht werden kann. Welche Antwort geben Sie Ihren Chef?

4 Ihr Manager gibt Ihnen den Auftrag zu untersuchen, inwiefern bestehende Module wieder verwendet werden können.
Ihr erster Vorschlag war: zerlegen aller Module die coincidential cohesion zeigen in Module mit functional cohesion.
Ihr Manager antwortet ihnen, dass in diesem Falle alle neuen Module noch nicht getestet wurden und auch noch dokumentiert werden müssen.
Was antworten Sie ihm darauf?

5 Welchen Einfluss hat die cohesion auf die Wartung?

6 Welchen Einfluss hat Coupling auf die Wartung?

7 Erklären Sie den Unterschied zwischen data encapsulation und data abstraction.

8 Erklären Sie den Unterschied zwischen Abstraktion und Data Hiding

9 Erklären Sie den Unterschied zwischen Polymorphismus und Dynamic Binding.

10 Modifizieren Sie die Java Programme im Skript so, dass sie fehlerfrei compilieren, mit Hilfe eines einfachen Main Programmes.

11 Java wurde vorgeworfen, dass die Sprache ADTs implementiert, aber auf Kosten des information hidings. Nehmen Sie dazu Stellung. Suchen Sie dazu im Internet nach ähnlichen kritischen Punkten von Java.

12 Welchen Einfluss hat cohesion auf Wiederverwendung?

13 Welchen Einfluss hat coupling auf die Wiederverwendung von Komponenten?

14 Warum wurden Objekte erst etwa zwanzig Jahre nachdem sie zum ersten Male in Simula eingeführt wurden, wieder entdeckt und zu einem Mainstream Thema?

15 Analysieren Sie das Ball-Java Programm (im Anhang) auf : information hiding, Abstraktionslevel, coupling und cohesion!

16 Wie würde diese Analyse aussehen, wenn an Stelle einer OO Programmiersprache eine konventionelle Programmiersprache eingesetzt würde?

17 Lesen Sie den Artikel. Betrand Meyer: "Lessons from the Design of the Eiffel Libraries" *Communications of the ACM* **33**, (Sept. 1990) pp 68 - 88

6.11. Anhang

```
//
//      the Ball World game
//

import java.awt.*;
import java.awt.event.*;


public class BallWorld extends Frame {

        public static void main (String [ ] args)
        {
                BallWorld world = new BallWorld (Color.red);
                world.show ();
        }

        private static final int FrameWidth = 600;
        private static final int FrameHeight = 400;
        private Ball aBall;
        private int counter = 0;

        private BallWorld (Color ballColor) {
                // constructor for new ball world
                        // resize our frame
                setSize (FrameWidth, FrameHeight);
                setTitle ("Ball World");

                        // initialize object data field
                aBall = new Ball (10, 15, 5);
                aBall.setColor (ballColor);
                aBall.setMotion (3.0, 6.0);
        }

        public void paint (Graphics g) {
                        // first, draw the ball
                aBall.paint (g);
                        // then move it slightly
                aBall.move();
                if ((aBall.x() < 0) || (aBall.x() > FrameWidth))
                        aBall.setMotion (-aBall.xMotion(), aBall.yMotion());
                if ((aBall.y() < 0) || (aBall.y() > FrameHeight))
                        aBall.setMotion (aBall.xMotion(), -aBall.yMotion());
                                // finally, redraw the frame
                counter = counter + 1;
                if (counter < 2000) repaint();
                else System.exit(0);
        }
}
```

```
//
//      general purpose reusable bouncing ball abstraction

//

import java.awt.*;

public class Ball {
        protected Rectangle location;
        protected double dx;
        protected double dy;
        protected Color color;

        public Ball (int x, int y, int r)
        {
                location = new Rectangle(x-r, y-r, 2*r, 2*r);
                dx = 0;
                dy = 0;
                color = Color.blue;
        }

                // functions that set attributes
        public void setColor (Color newColor)
                                { color = newColor;    }

        public void setMotion (double ndx, double ndy)
                                { dx = ndx; dy = ndy; }

                // functions that access attributes of ball
        public int radius ()
                                { return location.width / 2; }

        public int x ()
                                { return location.x + radius(); }

        public int y ()
                                { return location.y + radius(); }

        public double xMotion ()
                                { return dx; }

        public double yMotion ()
                                { return dy; }

        public Rectangle region () { return location; }

                // functions that change attributes of ball
        public void moveTo (int x, int y)
                                { location.setLocation(x, y); }

        public void move ()
```

```
                    { location.translate ((int) dx, (int) dy); }

        public void paint (Graphics g)
        {
                g.setColor (color);
                g.fillOval (location.x, location.y, location.width, location.height);
        }
}
```

Hier eine Erweiterung der Ball-Welt:

```
//
//      Multiple Ball version of the Ball World Game
//

import java.awt.*;
import java.awt.event.*;


public class MultiBallWorld extends Frame {

        public static void main (String [ ] args)
        {
                MultiBallWorld world = new MultiBallWorld (Color.red);
                world.show ();
        }

        private static final int FrameWidth = 600;
        private static final int FrameHeight = 400;
        private Ball [ ] ballArray;
        private static final int BallArraySize = 6;
        private int counter = 0;

        private MultiBallWorld (Color ballColor) {
                // constructor for new ball world
                        // resize our frame
                setSize (FrameWidth, FrameHeight);
                setTitle ("Ball World");

                        // initialize object data field
                ballArray = new Ball [ BallArraySize ];
                for (int i = 0; i < BallArraySize; i++) {
                        ballArray[i] = new Ball(10, 15, 5);
                        ballArray[i].setColor (ballColor);
                        ballArray[i].setMotion (3.0+i, 6.0-i);
                        }
        }

        public void paint (Graphics g) {
                for (int i = 0; i < BallArraySize; i++) {
                        ballArray[i].paint (g);
```

```
                        // then move it slightly
                    ballArray[i].move();
                    if ((ballArray[i].x() < 0) || (ballArray[i].x() > FrameWidth))
                        ballArray[i].setMotion (-ballArray[i].xMotion(),
ballArray[i].yMotion());
                    if ((ballArray[i].y() < 0) || (ballArray[i].y() > FrameHeight))
                        ballArray[i].setMotion (ballArray[i].xMotion(), -
ballArray[i].yMotion());
                    }
                        // finally, redraw the frame
            counter = counter + 1;
            if (counter < 2000) repaint();
            else System.exit(0);
        }
}
```

6.12. Anhang : Cohesion und Coupling in Java

6.12.1. Cohesion Beispiele in Java

Zuerst noch einmal die Definition
"Cohesion is the degree to which the tasks performed by a single module are functionally related."

IEEE, 1983

und einige Zitate:

"Cohesion is the "glue" that holds a module together. It can be thought of as the type of association among the component elements of a module. Generally, one wants the highest level of cohesion possible."

Bergland, 1981

"A software component is said to exhibit a high degree of cohesion if the elements in that unit exhibit a high degree of functional
relatedness. This means that each element in the program unit should be essential for that unit to achieve its purpose."

Sommerville, 1989

6.12.1.1. Module Cohesion

Die verschiedenen Stufen der Cohesion:

Coincidental (schlimmster Fall)
Logical
Temporal
Procedural
Communication
Sequential
Functional (bester Fall)

6.12.1.2. Coincidental Cohesion

Merkmal: keine oder wenig konstruktive Beziehungen zwischen den einzelnen Elementen eines Modules.

Objektstruktur:

Das Objekt stellt kein Objekt- orientiertes Konzept klar dar.
In der Regel handelt es sich um eine Sammlung von Quellcode eventuell mit Vererbungen.

Beispiel:

```
class Rous // ohne Gemeinsamkeiten
  {
  public static int findPattern( String text, String pattern)
    { // blah}
  public static int average( Vector numbers )
    { // blah}
  public static OutputStream openFile( String fileName )
    { // blah}
  }
```

6.12.1.3. Logical Cohesion

Merkmal: ein Module stellt mehrere Funktionen zur Verfügung, die irgend wie miteinander zu tun haben und die mit Hilfe von Funktionsparametern angewählt werden.

Die Struktur ähnelt jener des Control Coupling (siehe weiter unten)

Verbesserungsvorschlag:

Isolieren der einzelnen Funktionen in separate Operationen

Beispiel :

```
public void sample( int flag ) {
  switch ( flag ) {
    case ON:
      // bunch of on stuff
      break;
    case OFF:
      // bunch of off stuff
      break;
    case CLOSE:
      // bunch of close stuff
      break;
    case COLOR:
      // bunch of color stuff
      break;
  }
}
```

6.12.1.4. Temporal Cohesion

Merkmal: Die Elemente wurden in einem Modul zusammen gefasst, weil alle Elemente zeitlich innerhalb einer bestimmten Zeitlimite bearbeitet werden.

Allgemeines Beispiel:

"Initialization" Module stellen die Startwerte für ein oder mehrere Objekt(e) zur Verfügung.
"End of Job" Module räumen auf.

Beispiel :

```
procedure initializeData(){
  font = "times";
  windowSize = "200,400";
  foo.name = "Not Set";
  foo.size = 12;
  foo.location = "/usr/local/lib/java";
}
```

Verbesserungsvorschlag: jedes Objekt bekommt seinen eigenen Konstruktor und Destruktor

```
class foo {
  public foo() {
    foo.name = "Not Set";
    foo.size = 12;
    foo.location = "/usr/local/lib/java";
  }
}
```

6.12.1.5. Procedural Cohesion

Merkmal: Elemente werden auf Grund ähnlicher oder gleicher Algorithmen zusammen gezogen.

Es ist nicht ganz einfach Beispiele dafür zu finden. Aber die allgemeine Aussage oben zeigt in welcher Richtung die prozedurale Kohäsion geht

Beispiel:

```
class Stack {
  public void pop() {
    int next = elements[ top-- ];
    System.out.println( next );
  }
  public void push() {
    int addMe = Console.readInt( "Geben Sie eine ganze Zahl ein:" );
    elements[++top] = addMe;
  }
}
```

Verbesserungsvorschlag: versuchen Sie in den Elementen, hier den Methoden, nur soviel "Verarbeitung" zu integrieren, wie unbedingt nötig ist und lagern Sie alles andere in separate Module aus (Einlesen und push() sind im Beispiel unnötigerweise in ein und der selben Methode.

6.12.1.6. Communication Cohesion

Merkmale: alle Methoden eines Objektes operieren auf den selben Eingabedaten oder / und produzieren die selben Ausgabedaten.

Verbesserungsvorschlag: zerlegen Sie das Objekt in einzelne kleinere Objekte, jeweils mit einer oder wenigen Methoden.

Bemerkung : diese Art der Kohäsion tritt in Objekt- orientierten System kaum auf, auf Grund des möglichen Polymorphismus.

6.12.1.7. Sequential Cohesion

Merkmale: mehrere Operationen werden sequentiell also nacheinander abgearbeitet, in einem Modul. Das Ergebnis der Verarbeitung wird jeweils weiter gereicht, wie bei einer Pipeline.

Verbesserungsvorschlag: zerlegen der Verarbeitung in mehrere Module bezw. Objekte.

6.12.1.8. Functional Cohesion

Merkmale: ein Modul zeigt genau dann funktionale Kohäsion, wenn der Modul genau eine Funktion implementiert.

In einem Objekt-orientierten System:

Jede Operation eines public Interface eines Objektes sollte funktional kohäsiv sein
Jedes Objekt sollte ein einzelnes kohäsives Konzept enthalten, darstellen.

6.12.1.9. Informational Strength Cohesion

Myers sagt:

"The purpose of an informational-strength module is to hide some concept, data structure, or resource within a single module."

Merkmale : ein informational-strength Modul ist wie folgt definiert:
der Module enthält mehrere Entry Points, aber jeder Entry Point betrifft genau eine Form
Alle Funktionen hängen über Datenstrukturen oder Resourcen zusammen, die im Modul zusammengefasst sind.

6.12.1.10. Object Cohesion

Die Objekt Kohäsion ist ein Mass für den Zusammenhalt einer Klasse.

Damit man den Kohäsionsgrad einer Klasse oder eines Objektes abschätzen kann, muss man technisches Know How über die Applikation besitzen. Zudem sind Erfahrungen im Bauen, Modifizieren , Warten, Testen und Verwalten solcher Systeme nötig.

6.12.1.11. Fragen, die man stellen muss, um den Kohäsionsgrad zu bestimmen

Repräsentiert das Objekt ein vollständiges Konzept zusammenhängend oder stellt es eher eine Sammlung von Informationskonzepten dar?

Entspricht das Objekt einem real vorkommenden Objekt?

Wird das Objekt eher charakterisiert durch eine nicht zusammenhängende Menge von Daten, Statistiken oder ähnlichen Grössen

Ist jeder Methode, die public ist, eine eindeutige Verantwortung zugeordnet?

Stellt das Objekt auch losgelöst vom Kontext eine sinnvolle Definition eines Objektes dar.

6.12.2. Coupling in Java

Man unterscheidet folgende Kopplungsarten:

- Coupling
  - Data Coupling
  - Control Coupling
  - Global Data Coupling
  - Internal Data Coupling
  - Lexical Content Coupling
- Object Coupling
  - Interface Coupling
    - Object Abstraction Decoupling
    - Selector Decoupling
    - Primitive Methods
    - Selectors
    - Constructors
  - Inside Internal Object Coupling
  - Outside Internal Coupling from Underneath
  - Outside Internal Coupling from the Side

Dies ist eine verfeinerte Skala im Vergleich zu jener weiter vorne!

Referenz:
Object Coupling and Object Cohesion, chapter 7 of Essays on Object-Oriented Software Engineering , Vol. 1, Berard,
Prentice-Hall, 1993

6.12.2.1. Qualität von Objekten: Nicht alle Objekte sind gleich viel wert!

Es wurde versucht, eine Metrik für die Qualität von Objekten zu definieren.

6.12.2.1.1. Coupling als Metrik

gibt an, wie viel Interaktion *innerhalb* eines Objektes in einem System statt findet!

6.12.2.1.2. Cohesion als Metrik

gibt an, welche Aktivitäten von einem einzelnen Modul funktional zusammen hängen!

6.12.2.2. Zerlegbare Systeme / Decomposable Systems

Ein System ist immer dann zerlegbar, falls eine oder mehrere Komponenten des Systems keine Wechselwirkung oder anderweitige Beziehung mit andern Komponenten des Systems (auf der selben Abstraktionsebene) haben.

6.12.2.3. Ein fast zerlegbares System

Jede Komponente des Systems steht in Wechselwirkung mit jeder andern Komponente des Systems (auf der gleichen Abstraktionsebene).

6.12.2.4. Entwurfsziele

Die Wechselwirkung zwischen den einzelnen Komponenten eines Systems sollten möglichst gering sein.

"Unnecessary object coupling needlessly decreases the reusability of the coupled objects "

"Unnecessary object coupling also increases the chances of system corruption when changes are made to one or more
of the coupled objects"

6.12.2.5. Data Coupling

Die Ausgabe eines Modules ist die Eingabe des andern Modules

Abstraktes Beispiel:

Objekt A übetgibt Objekt X an Objekt B
Objekt X und B sind gekoppelt.
Änderungen an der Signatur von X haben zur Folge, dass B überholt werden muss.

Beispiel:

```
class Receiver
  {
  public void message( MyType X )
    {
    // code goes here
    X.doSomethingForMe( Object data );
    // more code
    }
  }
```

Probleme:

Objekt A übergibt Objekt X an Objekt B
X ist ein zusammen gesetztes Objekt.
Objekt B benötigt eine Komponente Y von X
B, X, die interne Darstellung von X, und Y sind gekoppelt

Beispiel : Sortieren von Studenten Datensätzen

```
class StudentRecord {
  Name lastName;
  Name firstName;
  long ID;

  public Name getLastName() { return lastName; }
  // etc.
}
SortedList swe = new SortedList();
StudentRecord newStudent;
//etc.
swe.add ( newStudent );
```

Warum ist dies eine schlechte Lösung?

```
class SortedList
  {
  Object[] sortedElements = new Object[ properSize ];
  public void add( StudentRecord X )
    {
    // coded not shown
    Name a = X.getLastName();
    Name b = sortedElements[ K ].getLastName();
    if ( a.lessThan( b ) )
      // do something
    else
      // do something else
    }
  }
```

Lösung 2

Warum ist diese Lösung besser?

```
class SortedList{
  Object[] sortedElements = new Object[ properSize ];
  public void add( StudentRecord X ) {
    // coded not shown
    if ( X.lessthan( sortedElements[ K ] ) )
      // do something
    else
      // do something else
  }
}
class StudentRecord{
  private Name lastName;
  private long ID;
  public boolean lessThan( Object compareMe ) {
    return lastName.lessThan( compareMe.lastName );
  }
  etc.
}
```

In dieser Lösung sind SortedList und StudentRecord nicht mehr gekoppelt.

```
interface Comparable {
  public boolean lessThan( Object compareMe );
  public boolean greaterThan( Object compareMe );
  public boolean equal( Object compareMe );
}
class StudentRecord implements Comparable {
  private Name lastName;
  private long ID;
  public boolean lessThan( Object compareMe ) {
    return lastName.lessThan( ((Name)compareMe).lastName );
  }
}
class SortedList {
  Object[] sortedElements = new Object[ properSize ];
  public void add( Comparable X ) {
    // coded not shown
    if ( X.lessthan( sortedElements[ K ] )
      // do something
    else
      // do something else
    }
  }
```

6.12.2.6. Control Coupling

Kontrollinformation wird zwischen Modulen ausgetauscht. Dadurch kontrolliert ein Modul die Kontrollsequenzen in einemanderen Modul.

Typisches Muster:

A sendet eine Message an B
B verwendet einen Parameter der Meldung um zu entscheiden, was zu tun ist

```
class Lamp {
  public static final ON = 0;
  public void setLamp( int setting ) {
    if ( setting == ON )
      //turn light on
    else if ( setting == 1 )
      // turn light off
    else if ( setting == 2 )
      // blink
  }
}
Lamp reading = new Lamp();
reading.setLamp( Lamp.ON );
reading.setLamp)( 2 );
```

Abhilfe:
Die Operation kann inmehrere Operationen zerlegt werden

```
class Lamp {
  public void on() {//turn light on }
  public void off() {//turn light off }
  public void blink() {//blink }
}
Lamp reading = new Lamp();
reading.on();
reading.blink();
```

Weiteres Standardmuster:

A sendet eine Message an B
B liefert Kontrollinformationen an A zurück

Beispiel: Fehlercodes

```
class Test {
  public int printFile( File toPrint ) {
    if ( toPrint is corrupted )
      return CORRUPTFLAG;
    blah blah blah
  }
}
Test when = new Test();
int result =  when.printFile( popQuiz );
if ( result == CORRUPTFLAG )
  blah
else if ( result == -243 )
```

Abhilfe ; mit eignen Exceptions arbeiten

```
class Test  {
  public int printFile( File toPrint ) throws PrintExeception {
    if ( toPrint is corrupted )
      throws new PrintExeception();
    blah blah blah
  }
}
try {
  Test when = new Test();
  when.printFile( popQuiz );
}
catch ( PrintException printError ) {
  do something
}
```

6.12.2.7. Global Data Coupling

Zwei oder mehrere Module greifen auf die selbe Datenstruktur zu

Typisches Muster:
Eine Methode in einem Objekt referenziert ein bestimmtes exzternes Objekt