Ausarbeitung zum Thema:

Auswirkungen des Geheimnisprinzips und der Modularisierung nach Parnas auf den Entwurf und die Entwicklung von Software und Programmiersprachen bis in die Heutige Zeit

Alexander Schatz
Hochschule für Technik Stuttgart
Bachelorstudiengang Informatik

Wintersemester 2007/08

26. Oktober 2007

Inhalt

1. Einleitung

Das in dieser Arbeit beschriebene Geheimnisprinzip ist ein Prinzip, dass wir nicht nur im Kontext der Informatik, sondern tagtäglich anwenden. Wir benutzen das Geheimnisprinzip zur Beherrschung der Komplexität unserer Umwelt; etwa indem wir Auto fahren, ohne wissen zu müssen, wie ein Verbrennungsmotor funktioniert oder ein elektrisches Gerät in Betrieb nehmen, ohne wissen zu müssen, wie der Strom in die Steckdose kommt.

Das zentrale Thema dieser Abhandlung ist die Modularisierung von Software, bzw. deren Umsetzung mittels des Geheimnisprinzips. Hierzu sind die Ursprünge und grundsätzlichen Überlegungen, und vor allem deren Umsetzung zu beleuchten.

Bei der Modularisierung kommt es wesentlich darauf an, dass die Module in sich geschlossene Teillösungen mit möglichst wenigen, wohldefinierten Interaktionen zu anderen Modulen bilden. Von einer guten Modularisierung verlangt man, dass sie folgendes leistet:

- Jedes Modul ist eine in sich geschlossene Einheit
- die Verwendung des Moduls erfordert keine Kenntnisse über seinen inneren Aufbau
- die Kommunikation eines Moduls mit seiner Umgebung erfolgt ausschließlich über eine klar definierte Schnittstelle
- Änderungen im Inneren eines Moduls, welche seine Schnittstelle unverändert lassen, haben keine Rückwirkungen auf das übrige System
- die Korrektheit des Moduls ist ohne Kenntnis seiner Einbettung ins Gesamtsystem nachprüfbar. [4]

Das von Parnas (1972) entdeckte Geheimnisprinzip (information hiding) ist das Gliederungskriterium, welches gute Modularisierungen mit den oben genannten Eigenschaften liefert. Das Geheimnisprinzip (information hiding) ist also folgendermaßen zu charakterisieren: Das Informaton Hiding ist ein Kriterium zur Gliederung eines Gebildes in Komponenten, so dass jede Komponente eine Leistung (oder eine Gruppe logisch eng zusammenhängender Leistungen) vollständig erbringt und zwar so, dass außerhalb der Komponente nur bekannt ist, *was* die Komponente leistet. *Wie* sie ihre Leistungen erbringt, wird nach außen *verborgen*. Parnas hat das Geheimnisprinzip ursprünglich definiert als das Einkapseln von Entwurfsentscheidungen beim Modularisieren von Software mit dem Ziel, die *Realisierung* der Entwurfsentscheidungen vor den Modulverwendern zu *verbergen* und die Entwürfe dadurch leichter *änderbar* zu machen.[10]

In Kapitel 2 werden wir näher darauf eingehen, wie das Geheimnisprinzip im Laufe der Zeit umgesetzt wurde, sowie Betrachtungen darüber anstellen, was geschieht, wenn Information Hiding nicht konsequent eingehalten wird.

Kapitel 3 betrachtet die Umsetzung des Prinzips bei heutigen Programmiersprachen. Auf die Umsetzung des Prinzips in weiterreichenden Techniken wie Testverfahren, Interoperabilität und Kompatibilität im Softwareentwurf, sowie Information Hiding im Kontext Verteilter Systeme, wird in Kapitel 4 eingegangen. Kapitel 5 beleuchtet die Zukünftige Entwicklung bezüglich des Geheimnisprinzips.

2. Umsetzung des Geheimnisprinzips im Laufe der Zeit

2.1. Stand der Modularisierung vor dem Geheimnisprinzip

Vor der Modularisierung war so genannter „Goto- Spagetticode“ das vorherrschende Softwaredesign. Dabei war weder der Kontrollfluss noch der Datenfluss strukturiert. Durch die Goto-Statements entstand ein vollkommen unüberschaubarer Kontrollfluss im Code. Die von Dijkstra begründete “Strukturierte Programmierung” verbesserte diesen Zustand. [3] Allerdings war die Datenhaltung innerhalb der Systems dabei noch genauso komplex und verworren wie zuvor. Dies stellte nach wie vor eine enorme Fehlerquelle dar, und verhinderte den erfolgreichen Entwurf komplexerer Systeme.

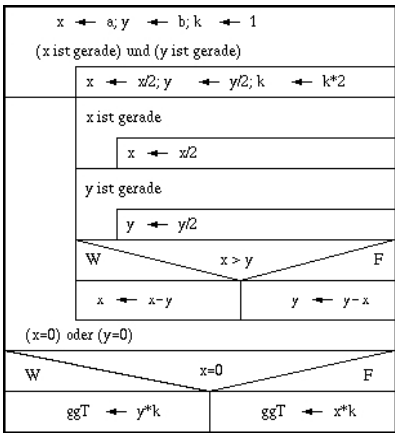

Abbildung 1 Strukturierter Code (jedoch nur Kontrollfluss ist strukturiert) [12]

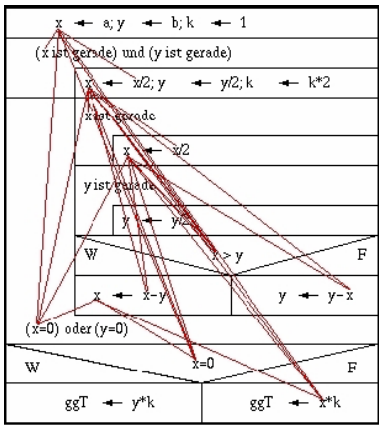

Abbildung 2 Daraus resultierender Datenfluss nur einiger der verwendeten Variablen [12]

Der Datenaustausch sollte dabei über gemeinsame, globale Variablen erfolgen. Wie sehr dies die Anpassbarkeit und Austauschbarkeit der Module einschränkt wurde damals noch nicht erkannt. [4]

Man stellte sich den Datenaustausch so stark vereinfacht wie in der folgenden Abbildung vor.

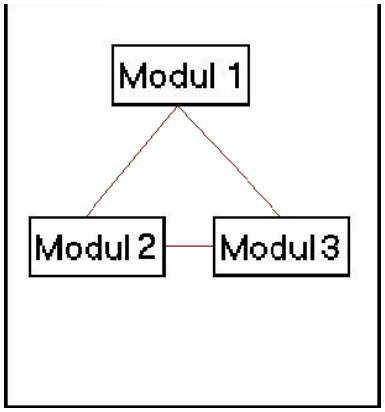

Abbildung 3 Theoretischer Datenaustausch bei der Modularisierung [12]

Tatsächlich läuft der Datenaustausch aber viel komplexer ab.

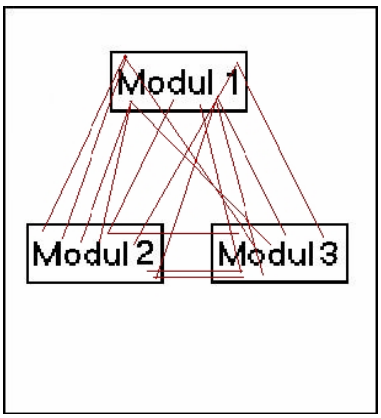

Abbildung 4 Tatsächlicher Datenaustausch [12]

Beispiel (in C-Code):

```
int lagerbestand; //globale Variable
void setzen (int bestand){
lagerbestand = bestand;
}
int abfrage() {
return lagerbestand;
}
void erhoehen(int bestand) {
lagerbestand = lagerbestand + bestand;
}
void verringern(int bestand) {
lagerbestand = lagerbestand – bestand;
}
```

2.2. Der Weg hin zum Geheimnisprinzip

Als David Parnas bei Philips Computer Industry arbeitete, war die Aufteilung von Softwaresystemen in Module noch völlig unbedeutend. Er war wie seine Kollegen davon überzeugt, dass Dijkstras Arbeiten das Problem der Komplexität gelöst hätten. Bei der industriellen Softwareentwicklung musste er jedoch schnell erkennen, dass dies absolut nicht der Fall war. Gegenüber seinem damaligen Chef bei Philips nannte er die dort erstellte Software ziemlich hässlich. Er merkte an, dass diese Software nicht den Dijkstra-Kriterien von „sauberen“ und „schönen“ Modulen entsprach. Der Philips Manager hielt dies aber für bedeutungslos, und forderte im Gegenzug Parnas dazu auf ihm zu sagen, welche Programmierprinzipien denn verletzt seien und wie die exakte Definition von „sauberen“ und „schönen“ Modulen sei. Parnas musste damals die Antwort schuldig bleiben.

Eine Beobachtung beim Mittagessen machte ihm das Problem bewusster. Er beobachtete einen Software- und einen Datenbankspezialist, wie sie auf einer Serviette den Zugriff auf einen Kontrollblock entwarfen. Diese Serviette wurde später kopiert, gelocht und abgeheftet. Es wurde versucht, die beiden Komponenten miteinander zu verbinden, was jedoch letztendlich scheiterte. Parnas war klar, dass irgendetwas bei der Vorgehensweise verändert werden sollte, hatte aber noch keine konkrete Vorstellung davon was. Daraufhin kamen ihm die ersten Gedanken in Richtung „Information Hiding“ (Geheimnisprinzip). Welche er in seiner ersten Abhandlung zu diesem Thema festhielt und 1971 auf der IFIP Konferenz vorstellte.

In seiner Arbeit machte er klar, dass es die Informationsverteilung ist, welche Softwaresysteme „dirty“ (schmutzig) macht. Nach der bis dahin angewandten Methode werden fast unsichtbare Verbindungen zwischen den Modulen aufgebaut, welche aber eigentlich unabhängig sein sollten. Laut David Parnas sollte der Zugriff auf Teile einer Programmeinheit, die für die „reguläre Benutzung“ nicht nötig sind, verboten sein. Das bedeutet also , dass diese in einem Modul versteckt werden müssen. Diese Art des Vorgehens, nennt man Datenkapselung und ist die Anwendung des Geheimnisprinzips auf Daten. Die Vorteile des Prinzips erkannte Parnas in der verbesserten Übersichtlichkeit und damit verbunden einer geringeren Fehleranfälligkeit. Des Weiteren werden auf solche Art programmierte Module austauschbar und wartbar. Dies war die Antwort auf die Fragen seines Managers. „Clean“ konnte er jetzt damit definieren, dass ein „sauberes Design“ den Prinzipien des Geheimnisprinzips folgen müsse. Einen weiteren Erkenntnisfortschritt erlangte David Parnas, als er einen Software Engineering Kurs an der Universität abhalten musste. Am Anfang des Kurses war noch keinem klar, wie dieser Kurs aufgebaut werden sollte, und welche Inhalte er haben muss. Ihm kam die

Idee, ein Projekt durchzuführen, welches ein Beispiel für das „Clean Design“ bilden sollte. Klar war, dass so ein Projekt nur sehr wenig Informationsverteilung beinhalten sollte. Er definierte 5 Module mit unterschiedlichen internen Implementierungen, welche zu einem festgesetzten Termin fertig gestellt sein, und dann durch verbinden dieser Module zum Laufen gebracht werden sollten. Er teilte den Kurs in 5 Gruppen von je 4 Studenten. Jeder einzelne der Gruppe hatte ein Modul mit derselben Spezifikation zu entwickeln. Diese Aufteilung ergab 1024 mögliche Kombinationen (4^5). Davon wurden 25 getestet und zum Laufen gebracht.

Weil die Module im Projekt eine unterschiedliche interne Implementierung hatten, konnten die Studenten keine detaillierten Informationen über sie austauschen. Den Studenten war eine abstrakte Schnittstelle gegeben worden, die für alle Varianten eines Moduls gleich war. Die Beschreibung der Schnittstelle wurde als Spezifikation behandelt. Von allen Implementierungen des Moduls wurde verlangt, dass sie diese Beschreibung erfüllen.

Den Studenten wurde nicht verraten, welche Kombinationen getestet werden würden, so dass sie keine zusätzliche Information austauschen konnten. Als Folge wurden die Implementierungsdetails als Geheimnis behalten. Die Studenten hätten eine Schnittstellenspezifikation austauschen müssen, für welche sie jedoch eine Spezifikationstechnik benötigten. Es war somit klar, dass eine Überwindung des Datenflussproblems nur über eine Benutzung abstrakter Schnittstellen möglich ist. Das Ergebnis dieses Experiments wurde in seiner Arbeit „Some Conclusions from an Experiment in Software Engineering Techniques“ beschrieben.

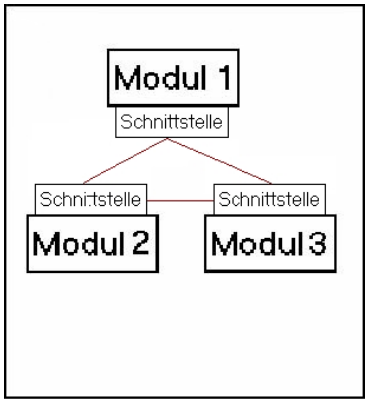

Abbildung 5 Datenfluss zwischen Modulen mit abstrakten Schnittstellen [12]

In weiteren Artikeln über das Geheimnisprinzip stellte Parnas fest, dass es fast unmöglich ist eine geeignete Sprache für die Modulspezifikation zu finden, und es somit besser wäre, die Spezifikation anhand der mathematischen Standardnotation durchzuführen. Er stellte weiterhin fest, dass Design das Wichtigste ist. Dies führt nur über eine Begrenzung der auszutauschenden Information. Schnittstellen dürfen also nur solide und dauerhafte Informationen enthalten. Beliebige oder sich wechselnde Informationen müssen im Modul gekapselt werden. Außerdem stellte er auch fest, dass der Vorteil des Geheimnisprinzips die Steigerung der internen Qualität und Änderbarkeit der mit diesem Prinzip entworfenen Software ist. Der größte Nutzen hoher interner Qualität ergibt sich natürlich bei Folgeversionen. Laut David Parnas zieht ein Softwareprodukt immer

Folgeversionen bzw. Varianten mit sich. Früher oder später sind neue Merkmale zu implementieren oder eine Portierung auf eine andere Plattform notwendig. Spätestens dann macht sich eine hohe interne Qualität bezahlt, da Folgeversionen wesentlich schneller auf den Markt gebracht werden können. Nun war es möglich Softwarepakete sauber zwischen Anwendungslogik und technischen Anteilen des Systems zu trennen. Vom Information Hiding Prinzip aus folgt, dass Software, die sich unterschiedlich schnell ändern wird, wie schon vorher erwähnt, in unterschiedliche Module gehört. Die

technische Infrastruktur ändert sich in schnelleren Zyklen als die Anwendung selbst. Durch die Trennung von Anwendung und Technik wird es grundsätzlich möglich, z. B. das Datenbanksystem auszutauschen, indem nur wenige technische Module angepasst werden. In der Praxis kommt es heute immer noch häufig vor, dass Anwendungslogik und Implementierungsdetails eng verwoben sind. [3],[4],[5],[6],[7]

2.3. Einschränkung der Umsetzung durch Missachtung des Geheimnisprinzips

Das Geheimnisprinzip beschreibt ein Vorgehen, bei dem die "öffentlichen" Informationen über ein Modul minimiert werden.

Gute Wissenschaftler und Ingenieure neigen dazu alle ihnen zugänglichen Informationen zu benutzen. Obwohl dies bei Einzelentwicklungen oder in sehr kleinen Gruppen zu hervorragenden Ergebnissen führt, ist es bei größeren oder gar verteilten Entwicklungsteams jedoch zum Scheitern verurteilt. Der Grund dafür ist, dass sich Informationen zu häufig ändern und die Kommunikation dieser Änderungen schwierig ist, da

- Man nicht weiss, wer welche Informationen
- überhaupt nutzt
- Nicht beurteilt werden kann, wie und ob sich eine Änderung auf andere Module auswirkt
- Die Kommunikation unter Umständen nur mit großer zeitlicher Verzögerung möglich ist
- Die Kommunikation vor allem bei internationalen Teams häufig verlustbehaftet ist
- Dadurch viel Produktivität verloren geht

Es geht deshalb darum, so wenige Details wie möglich und so viele wie unbedingt nötig auszutauschen.

Um die Folgen der Missachtung des Geheimnisprinzips zu illustrieren hier das Beispiel eines global agierenden Softwarehauses.

Das Softwarehaus hat zwei Entwicklungsstandorte. Einen in den USA, einen in Indien. In den USA arbeitet Entwickler A, in Indien die Entwickler B und C. A hat eine komplexe Java-Klasse geschrieben, dabei aber alle Attribute und Methoden als "public" deklariert, sprich sich nicht um das Verbergen von Implementierungsdetails gekümmert. B und C nutzen diese Klasse nun unabhängig voneinander in unterschiedlichen Softwareprodukten.

B beschränkt sich auf die in Dokumentation und Spezifikation definierten Klassen. Den Quellcode der Klasse hat er sich nie angeschaut.

C benutzt eine Vielzahl von Methoden der Klasse, und greift direkt auf Attribute zu. Unter anderem benutzt er auch Methoden, die von A lediglich als Hilfsfunktionen betrachtet werden, und daher nicht in der Beschreibung der Klasse auftauchen.

Es gibt eine Vielzahl von Szenarien, in denen sich das Vorgehen von C sowie die Entwurfsfehler von A sehr negativ auswirken.

1) A verändert den Code der Klasse.
   B ist davon kaum betroffen. Seine Programme werden nach wie vor tadellos funktionieren. Er kann sich darauf verlassen, dass die Klasse nach wie vor ihr dokumentiertes Verhalten aufweist. Er muss lediglich zur Kenntnis nehmen, dass eine neue Version des Moduls existiert, und diese von nun an verwenden.
   C muss mit massiven Problemen rechnen. Sein Code könnte plötzlich nicht mehr kompilieren, oder noch schlimmer, plötzlich subtiles Fehlverhalten aufweisen

Zudem ist C gezwungen, sich die Beschreibungen der Änderungen, oder sogar den neuen Code selbst, genau anzusehen. Bei mangelnden Englischkenntnissen ein fast unmögliches Unterfangen.

Wurden von C bereits Tests durchgeführt, sind diese nun wertlos. Die gesamte Software muss neu getestet werden, da nicht überschaubar ist wie sich die Änderungen auswirken.

2) Ein Fehler im Modul von A wird entdeckt, und dieser A per E-Mail mitgeteilt. Wegen der Zeitverschiebung beginnt A erst in 12 Stunden mit der Arbeit.

   B kann ohne Verzögerung weiterarbeiten. Sobald das korrigierte Modul verfügbar ist, wird er einfach den alten Code ersetzen. Auch nach der Korrektur wird das Modul noch der Schnittstellendokumentation genügen.

   C steht nun vor einem Dilemma. Er weiß nicht, auf welche Weise A den Fehler beheben wird und wie viel Zeit dieser dafür benötigen wird. Setzt er die Arbeit an seinem Code fort, läuft er Gefahr, seine gesamte Arbeit verwerfen zu müssen sobald die Änderungen abgeschlossen sind. Wartet er hingegen auf A, verliert er sehr viel Zeit.

   Nach Abschluss der Korrekturen durch A tritt zudem wieder Szenario 1) in Kraft

3) A schreibt sein Modul komplett neu, evtl. achtet er diesmal auch auf die Erzwingung des Geheimnisprinzips.

   B ersetzt das Modul sobald es fertig gestellt und getestet ist. Er wird automatisch von allen zukünftigen Verbesserungen profitieren.

   C wird nun mit hoher Wahrscheinlichkeit dazu gezwungen sein den Code des alten Moduls selbstständig weiter zu pflegen. Wird ein Fehler im neuen Modul entdeckt und behoben ist er gezwungen auch den alten Code auf diesen Fehler zu untersuchen und gegebenenfalls den Code zu korrigieren. Der Wartungsaufwand seiner Software hat sich massiv erhöht, und alle Vorteile durch die Modularisierung sind verloren.

Dies zeigt deutlich, dass Modularisierung für verteilte Softwareentwicklung unabdingbar ist, aber Modularisierung ohne durchdachte Auswahl der öffentlichen Komponenten und der Erzwingung des Geheimnisprinzips für die übrigen Komponenten wirkungslos ist.

3. Umsetzung des Geheimnisprinzips in heutigen Programmiersprachen

3.1. Objektorientierte Programmierung

Die heute so verbreitete Objektorientierte Programmierung kann – sofern korrekt angewendet - als eine direkte Umsetzung von Parnas Überlegungen betrachtet werden. Die Module finden in Form von Klassen und den daraus instanzierten Objekten eine natürliche Umsetzung. Einheitliche Schnittstellen werden durch Vererbung oder die Implementierung explizit definierter Schnittstellen umgesetzt. Die Gruppierung von Datenstrukturen und all ihren Zugriffsmethoden innerhalb einer Klasse entspricht genau der von Parnas empfohlenen Methode der Modularisierung. Dasselbe gilt für das Verbergen von Implementierungsdetails durch Abstraktion und die Bereitstellung exklusiver, sicherer Zugriffsmethoden auf Daten.

Wie aber gerade Parnas selbst betont, garantiert die Verwendung objektorientierter Programmiersprachen noch lange keinen sinnvoll modularisierten und somit wart- und austauschbaren Code. Falscher Gebrauch der Möglichkeiten dieser Sprachen – insbesondere der Vererbung - kann schnell wieder zu den verborgenen Abhängigkeiten führen, die man durch Modularisierung eigentlich vermeiden wollte.

Zu viele öffentlich zugängliche Methoden oder gar Attribute nehmen Änderungsfreiheiten und zerstören die Kapselung. Schlecht geplante Zugriffsmethoden machen die gemeinsame Nutzung eines Objekts schwierig, und erfordern wieder tiefe Kenntnisse einer Klasse im zu benutzenden Code.

Die zusätzlichen Eigenschaften, die gut modularisierter Code haben muss werden nach Parnas also erst durch die Anwendung von geeigneten Entwurfsmustern und einer gründlichen, ingenieurmäßigen Ausbildung der Entwickler erreicht. Objektorientierte Programmierung alleine löst dieses Problem also nicht, kann dabei aber ein mächtiges Hilfsmittel sein.

3.2. Das Beispiel Java

Moderne Objektorientierte Programmiersprachen bieten Konstrukte an um Modularisierung zu vereinfachen und das Geheimnisprinzip zu erzwingen. In Java werden Module in Form einer Klasse oder einer Sammlung von Klassen – einem Paket – verwirklicht.

Es gibt dabei Schlüsselwörter, die die Sichtbarkeit von Daten, Methoden und Klassen genau regeln. Es ist damit möglich, Daten nach außen vollkommen zu verbergen, nur bestimmten “verwandten” Klassen zugänglich zu machen, oder globalen Zugriff darauf zu gewähren.

Ebenfalls gibt es Mechanismen, die sicherstellen, dass eine Klasse eine zuvor definierte Schnittstelle auch tatsächlich anbietet.

Beispiele:

Der folgende Java Code definiert eine Schnittstelle.

```
interface Konto {
public int getStand();
public void zahleEin(int betrag);
}
```

Klassen, die diese Schnittstelle implementieren wollen, müssen die Methoden getStand() und zahleEin() implementieren. Die Methoden müssen zudem öffentlich zugänglich sein, die passenden Parameter akzeptieren und die passenden Rückgabewerte liefern.

Sämtliche weiter Eigenschaften einer daraus abgeleiteten Klasse sind implementierungsspezifisch, und sollten von Nutzern dieser Klasse nicht benutzt werden, bzw. sollten vom Entwickler der Klasse gar nicht erst benutzbar gemacht werden.

Die folgende Klasse implementiert diese Schnittstelle:

```
class GiroKonto implements Konto {
private int stand;
public int getStand() {
        return stand;
}
public void zahleEin(int betrag) {
        if (isValid(betrag)) {
                stand = stand + betrag;
        }
}
private boolean isValid(int betrag) {
        ...
}
}
```

Die geforderten Methoden wurden implementiert. Zudem wurde eine Variable “stand” eingeführt, auf die von außen nicht direkt zugegriffen werden kann.

Die Methode isValid() kann ebenfalls nicht von außen aufgerufen werden, sondern wird lediglich von der aktuellen Implementierung der Klasse intern verwendet. Dadurch kann ihr Verhalten in zukünftigen Versionen der Klasse beliebig angepasst werden.

Wäre das Attribut “stand“ ebenfalls als “public“ deklariert, würde das Geheimnisprinzip nicht mehr erzwungen. Bei Umbenennungen des Attributs oder bei Änderungen seiner Semantik könnten dann Probleme auftreten. Durch die Deklaration als “private“ hat der Entwickler der Klasse “GiroKonto“ aber alle Freiheiten. Das Gleiche gilt für die Methode isValid().

3.3. Modularisierung in der Praxis

Eine Grundidee von Parnas ist es, Details über Datenstrukturen vollkommen zu verbergen. Eine Umsetzung dieser Idee ist das Collection-Framework in Java. [9]

Über einheitliche Methoden wie z.B. add(), remove(), contains() oder Iteratoren kann man Datenstrukturen manipulieren und durchsuchen ohne etwas über die Strukturen selbst wissen zu müssen. Der Umgang mit verketteten Listen, Bäumen oder Hashtabellen ist vollkommen identisch, obwohl sich diese Strukturen intern stark unterscheiden.

Bemerkt man beispielsweise, dass das Laufzeitverhalten der gewählten Struktur nicht zu den Anforderungen passt, genügt es eine Zeile im Programmcode zu ändern um

eine völlig andere Datenstruktur zu verwenden. Der eigentliche fachliche Code bleibt völlig unberührt. Um dies in der Praxis zu erreichen genügen aber die Mittel der Programmiersprache nicht. Zusätzliche

Konventionen die sowohl von den Entwicklern der Klassenbibliothek als auch deren Nutzern eingehalten werden müssen sind notwendig. Ebenso ist es erforderlich diese Konventionen zu dokumentieren und als vollwertigen Bestandteil der Schnittstelle zu betrachten. Beispielsweise erfordert ein binärer Suchbaum eine Größer-Kleiner-Gleich-Relation zwischen seinen Elementen. Wird diese für die zu speichernden Objekte gar nicht oder fehlerhaft definiert, kann die Datenstruktur nicht funktionieren. Entsprechend dem Prinzip der Modularisierung muss diese Relation in der Klasse der zu speichernden Objekte definiert werden.

Das Framework kann zwar schon zur Kompilierzeit erkennen ob dies geschehen ist, indem es von den Objekten die Implementierung einer genau definierten, abstrakten Schnittstelle fordert (Ein Java-Interface), allerdings muss das semantisch korrekte Verhalten dieser Methoden vom Entwickler sichergestellt werden.

Die Entwickler des Frameworks müssen ihrerseits sicherstellen, dass alle internen Abläufe vollkommen verborgen werden. Es darf vom Benutzer beispielsweise nicht gefordert werden bei der Benutzung bestimmter Datenstrukturen Sperrmechanismen anzuwenden, die bei anderen Strukturen nicht benötigt werden. Sind Sperren notwendig müssen diese intern verwaltet werden, oder grundsätzlich für alle Datenstrukturen vom Benutzer gefordert werden, d.h. in die Dokumentation der abstrakten Schnittstelle aufgenommen werden.

Wie das Beispiel der Datenstrukturen allerdings auch zeigt, ist es nicht möglich, sämtliche

Eigenschaften der Implementierung zu verbergen. Das liegt einerseits an Beschränkungen existierender Programmiersprachen – Angaben über das Laufzeitverhalten einer Methode sind im Quellcode nicht möglich -, andererseits aber auch an der enormen Komplexität dieses Problems.

Die Funktion contains() wird beispielsweise bei einer Hashtabelle erheblich schneller arbeiten als bei einer langen, verketteten Liste. Selbst wenn nach einen Wechsel zwischen diesen beiden Datenstrukturen das Programm prinzipiell korrekt weiterarbeitet, könnte das massiv veränderte Laufzeitverhalten das Ergebnis praktisch unbrauchbar werden lassen. Dabei ist auch deutlich zu machen, welche weiteren einheitlichen Eigenschaften nicht garantiert werden.

Beispielsweise zerstört eine Hashtabelle jegliche Vorsortierung der Elemente, während eine lineare Liste diese erhält und ein Baum eine Sortierung anhand der definierten Relation erzwingt.

Solche Unterschiede sind in Java nicht formal beschreibbar. Dies widerspricht zwar dem Konzept der vollständigen Kapselung, ist aber aus Gründen der Effizienz unvermeidlich. Man kann beispielsweise die Iteratoren so gestalten, dass sie Elemente immer in der Reihenfolge ihrer Einfügung ausgeben. Dies bedeutet einerseits, dass erhebliche Zusatzinformationen innerhalb der Datenstrukturen erforderlich sind und andererseits, dass das Laufzeitverhalten verändert werden muss, so dass die Vorteile bestimmter Strukturen völlig verloren gehen.

Die Notwendigkeit solcher Kompromisse bei der Modularisierung wurde von Parnas auch erkannt. Die Lösung dafür ist sorgfältige Dokumentation von Modulen und die Beachtung dieser Dokumentation bei der Benutzung.

Die Tatsache, dass das Generics-Framework seine schlechteren Vorgänger erst nach vier Versionen bzw. zehn Jahren abgelöst hat zeigt auch, wie schwer der Entwurf sauber modularisierter und massiv wieder verwendbarer Software auch heute noch ist.

4. Einhaltung des Geheimnisprinzips in weiterreichenden Techniken

4.1. Testverfahren

Zentrale Bedeutung haben Kapselung und Modularisierung heutzutage bei Softwaretests. Moderne Testverfahren wie “Unit-Tests” [10] sind nur aussagekräftig und praktikabel, wenn es keine versteckten Interaktionen zwischen den Modulen gibt und sich der Zustand von Modulen nicht unbemerkt verändern kann.

Dazu muss es möglich sein, ein Modul wiederholbar auf einen bekannten Zustand initialisieren zu können. Nur die Informationen, die dem Modul über seine definierten Schnittstellen übergeben wurden, dürfen dessen Zustand beeinflussen. Sind Module nicht sauber getrennt, könnte ein Fehler in einem Modul oder in der Testumgebung die Daten eines anderen Moduls verfälschen. Es wäre dadurch unmöglich, den genauen Ort des Fehlers schnell festzustellen oder reproduzierbare Ergebnisse zu erhalten.

Ähnlich ist die Situation bei formalen Korrektheitsbeweisen. Diese sind nur möglich, wenn man alle Schnittstellen genau kennt.

Der Bereich der Büro- und Heim-PCs wird heute sehr stark vom Betriebssystem Microsoft Windows dominiert. Dieses System stellt in unzähligen Bibliotheken eine Vielzahl von Funktionen bereit, die weit über den Umfang klassischer Betriebssysteme hinausgehen. [11]

Millionen von Anwendungen unterschiedlichster Softwarehersteller nutzen diese Funktionen.

Verlassen sich diese Anwendungen nun auf Implementierungsdetails einer bestimmten Version von Windows, wäre es für Microsoft unmöglich sein System weiter zu entwickeln oder darin Fehler zu korrigieren. In der Praxis unterscheidet sich die Implementierung einiger Funktion in Windows von Version zu Version ganz erheblich. Beispielsweise wurde Code, der in alten Versionen Teil des Kernels war in den Benutzermodus verlagert, und umgekehrt. Anwendungen, die sich auf die öffentliche Beschreibung der API beschränken sind von diesen Änderungen völlig abgekapselt und können in der Praxis problemlos auf verschiedensten Versionen von Windows laufen.

Verlassen sich Anwendungen auf Implementierungsdetails, so wird jede noch so kleine Änderung an einer Bibliothek weltweite Softwareanpassungen und -tests erfordern, deren Kosten leicht mehrere Milliarden Euro betragen. Teilweise wäre die Durchführung dieser Maßnahmen gar nicht möglich, weil der Hersteller der Software sich schon längst vom Markt zurückgezogen hat.

4.2. Interoperabilität und Kompatibilität

Die oben beschriebene Situation gleicht der bei der verteilten Softwareentwicklung. Die auszutauschenden Informationen müssen minimiert und möglichst formalisiert werden um massive Mehrarbeit zu verhindern und die Zuverlässigkeit der Software zu gewährleisten. Entwickler müssen klare Informationen über das Verhalten der gelieferten Bibliotheken erhalten, und sich darauf verlassen können, dass diese Informationen gültig bleiben.

Der Hersteller der Bibliotheken muss im Gegenzug davon ausgehen können, dass nur die von ihm veröffentlichten Eigenschaften seiner Software genutzt werden. Zudem muss er beim Entwurf seiner Schnittstellen sehr sorgfältig vorgehen. Erweist sich ein Design nach längerer Zeit als mangelhaft ist es nicht mehr möglich die Schnittstelle zu ändern. Es bleibt dann nur die Möglichkeit eine zweite Bibliothek zu entwickeln, die die Funktionalität ihres Vorgängers

über eine andere Schnittstelle bietet.

Natürlich ist dies in der Praxis schwer zu vermeiden, da die Anforderungen an Module einem

ständigen Wandel unterliegen. Es kommt dann durchaus vor, dass ein Hersteller gezwungen ist mehrere Versionen einer Bibliothek gleichzeitig anzubieten und zu pflegen. Insbesondere bei neu eingeführten Technologien ist dies ein häufiges Problem, da Erfahrungen fehlen, das Anwendungsszenario sich ändert und ein gewisser Druck besteht das eigene Produkt schnell auf dem Markt zu platzieren. Ein Beispiel dafür ist das Framework MSXML von Microsoft, das innerhalb weniger Jahre in sechs Versionen erschienen ist. Aufgrund der sicherheitskritischen Natur dieser Bibliothek müssen auch heute noch die Versionen 3 bis 6 gepflegt werden. [13]

4.3. Verteilte Systeme

In Verteilten Systemen stellt Modularisierung nicht mehr nur eine mögliche Technik des Systementwurfs dar, sondern ist eine grundlegende Eigenschaft dieser Systeme.

Während in herkömmlichen Systemen bei schlechtem Design sehr leicht ungewollte Kommunikation zwischen Modulen auftreten kann, sind Entwickler Verteilter Systeme mit dem entgegengesetzten Problem konfrontiert. Es gibt keinen von allen Modulen gemeinsam genutzten Arbeitsspeicher mehr. Jegliche Kommunikation zwischen Modulen muss daher aktiv erfolgen und ganz bewusst angestoßen werden. Aufgrund des relativ langsamen Datenaustausches über Netzwerke ist es dabei unerlässlich, den Ablauf der Kommunikation genau zu verstehen, sprich die Schnittstellen genau zu kennen. Caching-Verfahren sind ebenfalls nur möglich, wenn die Regeln der Kommunikation genau definiert sind. Ansonsten ist es nicht möglich die Integrität der Daten zu gewährleisten und unnötige Kommunikation zu vermeiden.

Solche Systeme sind also ohne durchdachte Modularisierung und sauberen Entwurf der Schnittstellen niemals in der Lage effizient und zuverlässig zu funktionieren.

Mit der steigenden Bedeutung Verteilter Systeme wird auch die Bedeutung von Parnas Ideen noch zunehmen.

5. Ausblick in die Zukunft

Ein heutiger Entwicklungstrend ist, den Entwurf von komplexen Softwaresystemen, durch so genannte Entwurfsmuster zu vereinfachen. Dies geschieht mittels der Parnas'schen Prinzipien der Modularisierung und des Geheimnisprinzips.

Der Entwurf objektorientierter Software an sich, unter Einhaltung der Prinzipien von Parnas, ist schwer. Wesentlich schwerer ist der Entwurf wiederverwendbarer objektorientierter Software. Man muss dabei die relevanten Module herausarbeiten, und sie zu Klassen passender Granularität abstrahieren. Ein Entwurf muss sowohl den spezifischen Programmanforderungen genügen, als auch allgemein genug sein, um Widerverwendbarkeit in ähnlichen Problemkreisen zu gewährleisten. Eine weitere Anforderung ist Revisionen von Entwürfen zu vermeiden.

Dies erreicht man durch Entwicklung von so genannten Entwurfsmustern. Diese Muster lösen spezifische Entwurfsprobleme. Ein Entwurfsmuster stellt einen Baustein dar, der für bestimmte Problemklassen Objektzusammenstellungen mit festen Kommunikationsschnittstellen festlegt. Bei der Softwareerstellung mittels dieser Muster muss nur die jeweilige Klasse angepasst werden.[8]

Ein weiteres Forschungsgebiet sind Ereignisbasierte Softwarearchitekturen. Diese sind auf Grund ihrer großen Vorteile zu einem der wichtigsten Merkmale großer verteilter Systeme geworden. Die lose Kopplung der beteiligten Komponenten erlaubt es, autonome, heterogene Systemteile leicht zu integrieren und die Entwicklungsfähigkeit und Skalierbarkeit der entstehenden, komplexen Systeme zu steigern.

Bisher wurde hauptsächlich auf die Effizienz der Notifikations-Dienste Wert gelegt, wohingegen Entwurf, Programmierung und Administration solcher Systeme wenig betrachtet wurden. Im Bereich Software Engineering wurde schon frühzeitig die Bedeutung von Kapselung (information hiding) und Abstraktion erkannt, die maßgeblich die Entwicklung des strukturierten Programmierens, des Modul-, Klassen-, und Komponentenbegriffs beeinflusst haben. Obwohl diese Ideen in Request/Reply-basierten Systemen umgesetzt worden sind (z.B. Corba), fehlen vergleichbare Strukturierungsmechanismen für ereignisbasierte Systeme. In der Literatur ist kein Modulkonzept oder Komponentenbegriff für ereignisbasierte Systeme eingeführt worden und typische Implementierungstechniken wie Publish/Subscribe führen neben den primitiven API Aufrufen keine weiteren (Programmierabstraktionen ein. Ein geeignetes Modulkonzept kann die Beziehungen zwischen den Komponenten materialisieren und stellt somit für den Programmierer und den Administrator ein wertvolles Hilfsmittel dar.[9]

6. Fazit

Angefangen mit der geschlossenen Entwicklung von Software innerhalb einer Firma bis hin zur Kombination von Modulen unterschiedlicher Hersteller spielen Geheimnisprinzip und Modularisierung eine zentrale Rolle.

Diese Arbeit verdeutlichte, wie sehr die Welt der modernen Softwareentwicklung von Parnas Erkenntnissen profitiert, und wie sehr sich diese Ideen auswirken.

Ebenso wurde auf die steigende Bedeutung der Modularisierung in existierenden und zukünftigen Verteilten Systemen hingewiesen.

Es wird klar ersichtlich, dass das Geheimnisprinzip auf Ebene der Programmiersprachen und Programmierparadigmen weitestgehend umgesetzt wurde. Darüber hinaus stellen wir jedoch fest, dass der Faktor Mensch bei der Umsetzung eine gravierende Rolle spielt. Noch ist es dem Programmierer möglich die Prinzipien sauberer Modellierung mit dem Geheimnisprinzip auszuhebeln und somit ungewollte Abhängigkeiten zu schaffen.

Zwei Voraussetzungen zur vollständigen Umsetzung des Geheimnisprinzips, wären einerseits die größere Gewichtung sauber Programmierung und Modellierung in der Lehre von Softwaretechnik und Programmierung, sowie anderseits die Entwicklung von Programmiersprachen, die die strikte Umsetzung unterstützen und deren Umgehung erschweren, bzw. unmöglich machen.

Abbildungsverzeichnis

Literatur

[1] Hompage der HFT Stuttgart, Blackboard, http://www.hft-stuttgart.de

[2] Johannes Schultze, J. H. S., Das Geheimnisprinzip - Zwei Artikel von David Lorge Parnas, Seminar Grundlegende Konzepte der Softwaretechnik, http://www.informatik.uni-hamburg.de/SWT/attachments/LVTermine/02-ParnasInformationDistributionAspects.pdf zuletzt abgerufen am 25.10.07

[3] Parnas, D. L., Modularization by Information Hiding, 2001, Konferenzbeitrag (Software Pioniere), http://www.sdm.de/download/sdm-konf2001/f_3_parnas.pdf zuletzt abgerufen am 25.10.07

[4] Parnas, D., On the Criteria To Be Used in Decomposing Systems into Modules, December 1972, Communications of the ACM (Artikel), http://campus.hesge.ch/Daehne/2006-2007/Module625/Algo/01-Article%20original%20de%20Parnas.pdf zuletzt abgerufen am 25.10.07

[5] Rechenberg, Pomberger, Informatikhandbuch,1997

[6] IDuden, Informatik 1993

[7] Hans Dieter Hellige, Geschichten der Informatik – Visionen, Paradigmen, Leitmotive, 2004

[8] Haltof, H., Entwurfsmuster, Modularisierung 06, 2006, Vorlesungsunterlage http://www.hs-bremerhaven.de/Binaries/Binary5727/Entwurfsm_modularisierung_06.pdf, zuletzt abgerufen am 25.10.07

[9] Himmerlich, N., Grundkonzepte Objektorientierter Programmierung / Beispiele in Java, Oberon-2, Python und Delphi, 2006, Projektarbeit im Studiengang Lehramt Informatik, Friedrich-Schiller-Universit Jena http://www.uni-jena.de/img/unijena_/faculties/minet/casio/DidaktikDerInformatik/studentischeArbeiten/ProjektarbeitHimmerlich.pdf zuletzt abgerufen am 25.10.07

[10] Stuttgart, F. W. U., Objektorientierte Architekturen, Frameworks und Architekturmuster, Wintersemester 2002/03, Hauptseminar an der Universität Stuttgart, http://www.iste.uni-stuttgart.de/ps/Lehre/HS_OO_Entwurf/Wallwitz.pdf zuletzt abgerufen am 25.10.07

[11] Microsoft, Windows API, 10.1.2007, Online Library, http://msdn2.microsoft.com/en-us/library/Aa383750.aspx zuletzt abgerufen am 25.10.07

[12] Hartzendorf, M., Problemseminar "Wegweisende Arbeiten in der Softwaretechnik, 2004, Seminararbeit, http://ebus.informatik.uni-leipzig.de/www/media/lehre/seminar-pioniere04/hartzendorf-ausarbeitung-parnas.pdf zuletzt abgerufen am 25.10.07

[13] Microsoft, XML, 2007, MSDN Online Library,
http://msdn2.microsoft.com/de-de/library/aa286548.aspx
zuletzt abgerufen am 25.10.07