Software-Entwurf

Software-Entwurf

Software wird selten allein entwickelt. Also braucht man einen Plan, wie vorgegangen werden soll, wie *arbeitsteilig* entwickelt wird. Dieser Plan muss also eine Art *Bauplan* für die zu konstruierende Software sein. Das Erstellen dieses Bauplans wird als *Entwerfen* bezeichnet – und das Ergebnis als der *Entwurf*.

Nebenbei: man *kann* Software *nicht ohne* einen Bauplan konstruieren. D.h. jeder der Software entwickelt, hat einen Plan, was er wie tun möchte. Manchmal ist dieser Plan nur im Kopf eines Programmierers vorhanden und manchmal ist er zu Beginn des Programmierens noch so verworren, dass er sich während der Implementierung immerzu wieder ändert.

Diese Vorlesung ist ein Plädoyer dafür, sich *zuerst* einen Bauplan zu machen, diesen zu überdenken, im Team zu diskutieren, und erst dann mit der Implementierung zu beginnen. *Wieviele* Details im Bauplan bereits festgelegt werden sollten – dies freilich hängt stark vom jeweiligen Projekt, von der Erfahrung der Teammitglieder, den Anforderungen des Kunden und weiteren Faktoren ab.

Ziele des Entwurfs

Ausgehend von der Anforderungsanalyse und einer ersten Skizze des zu konstruierenden Systems in der Szenarioanalyse, muss im Entwurf festgelegt werden:

- Technische Basis/Infrastruktur
- Softwarearchitektur
- Entwurf der Komponenten und Klassen

Diese Überlegungen müssen in einer Weise notiert werden, dass sie den Entwicklern zugänglich und verständlich sind. Sie müssen so vollständig und konsistent sein, dass eine arbeitsteilige Erstellung der Software möglich wird.

Aufgaben im Entwurf

Die angesprochenen Entscheidungen betreffen:

Softwareentwicklung setzt heute auf bestimmte Gegebenheiten, eine *technische Basis und Infrastruktur* auf. Die Wahl dieser Infrastruktur hat eine große Bedeutung für die spätere Entwicklung und für den Detailentwurf.

Fragestellungen sind hier beispielsweise

- Welche Verteilung des Systems wird benötigt?

  Bei Webanwendungen wird heute in der Regel eine sogenannte 3-Tier-Systemarchitektur verwendet, bei Anwendungen im Intranet kommt aber auch eine Client/Server-Architektur in Betracht. Diese Entscheidung kann sich grundlegend auf den Bauplan, also den Entwurf auswirken.

- Welche Hardware wird eingesetzt?

  Bei einem System im technischen Bereich, denken wir wieder mal an die Steuerung eines Airbags, kann der Typ des verwendeten Microcontrollers einen wesentlichen Einfluss auf die zu konstruierende Software haben.

- Welcher Persistenzmechanismus ist erforderlich?

  Bei vielen Anwendungen werden Daten langfristig gespeichert, also ist zu überlegen, welcher Persistenzmechanismus eingesetzt werden soll: dateibasierte Speicherung? Datenbanksystem? NoSQL-Speicherung?

- Welche Betriebssysteme und Rechnernetze müssen unterstützt werden?

  Eine Frage kann auch sein, auf welchen Betriebssystemen und mit welchen Netzprotokollen die zu erstellende Software lauffähig sein soll...

Häufig werden Entscheidungen über die verwendete Infrastruktur zu einem frühen Zeitpunkt getroffen. Manchmal sind sie Bestandteil der Anforderungen, weil die Infrastruktur bereits durch frühere Entscheidungen vorgegeben ist.

Ein weiterer Schritt im Entwurf größerer Systeme besteht darin, eine *Architektur* der Software zu finden. Dabei geht es darum, die wichtigsten Komponenten oder Subsysteme zu identifizieren und die Art und Weise, wie sie geordnet zusammenwirken sollen. Im Vordergrund stehen grundlegende Fragen, wie der *Informationsfluss* im System und wie die *Ablaufsteuerung* organisiert werden sollen.

Wir haben bereits bei der Szenarioanalyse die Grundzüge *einer* Softwarearchitektur kennengelernt: Bei webbasierten Anwendungen wird oft

eine Schichtenarchitektur verwendet, bei der die grundlegenden Komponenten oder Schichten sind:

- Komponenten der Präsentationsschicht, die dem Benutzer Informationen anbieten und mit ihm interagieren – in der UML mit der Stereotype «boundary» gekennzeichet.
- Komponenten der Anwendungsschicht, die für die eigentliche Ablaufsteuerung der Anwendung zuständig sind; die die sogenannte „Geschäftslogik“ durchführen und dafür sorgen, dass die Informationen der Anwendung in der gewünschten Weise verarbeitet, transformiert oder aufbereitet werden. Diese Teile der Software, die mit «control» gekennzeichnet werden, sorgen also für Ablaufsteuerung und Informationsfluss.
- Die Persistenzschicht hat die Verantwortung für das Speichern der Informationen – ihre Komponenten werden mit der Stereotype «entity» gekennzeichnet.

In dieser Veranstaltung werden wir uns nur mit dieser (sehr wichtigen und häufigen) Softwarearchitektur befassen. (Im Laufe des Studiums, insbesondere im Master-Studiengang, werden weitere Veranstaltungen angeboten, bei denen diese Fragen vertieft werden. Im Buch von Zuser u.a. finden Sie auf S. 276 eine kurze Darstellung, welche Arten von Softwarearchitekturen es gibt.)

Der Entwurf der Softwarearchitektur ist oft eng mit den Entscheidungen über die technische Basis und Infrastruktur des Systems verzahnt. Entscheidungen in beiden Bereichen können sich gegenseitig beeinflussen, weil sie den Raum der möglichen Lösungen einschränken. Andererseits ist es aber auch die Aufgabe der Softwarearchitektur, flexible Lösungen zu finden, die auch spätere Variabilität in Bezug auf die Infrastruktur erlaubt. Ein Beispiel: eine gute Architektur bindet ein Datenbanksystem *so* ein, dass es eventuell später durch ein anderes Produkt ersetzt werden kann. Oder: eine Architektur kann dafür sorgen, dass betriebssystemabhängige Funktionen so gekapselt werden, dass eine spätere Portierung auf ein anderes Betriebssystem möglich und vom Aufwand vertretbar ist.

Schließlich müssen die Details der Komponenten in den Schichten der Anwendung festgelegt werden (bei einer 3-Schichten-Architektur):

- Details der Benutzeroberfläche oder anderer externer Schnittstellen. Bei der Benutzeroberfläche kann man den Prototypen aus der Anforderungsanalyse einsetzen und verfeinern.

- Details der Anwendungslogik: wie kommen Daten von der Oberfläche in die Datenbasis? Welche Verarbeitungsschritte sind notwendig?
- Details der Persistenzschicht: wie werden die Daten gespeichert? Wenn in einem relationalen Datenbanksystem: Datenbankschema (Thema in der Veranstaltung Datenbanksysteme)

Dieser Detailentwurf führt uns schließlich zum Entwurf von Klassen und ihren Methoden.

Die grundlegenden Fragen hierbei kann man in drei „Dimensionen“ sehen:

1. statische Sicht, auch Struktursicht:

   Welche Klassen benötigen wir, welche Attribute haben sie, wie sind sie strukturell verbunden? Wie organisieren wir die Assoziationen der Klassen, welche Container werden eingesetzt usw.?

2. dynamische Sicht:

   Welche Methoden werden benötigt? Wie werden sie verwendet? Wie beeinflussen sie den Zustand einer Komponente oder Klasse?

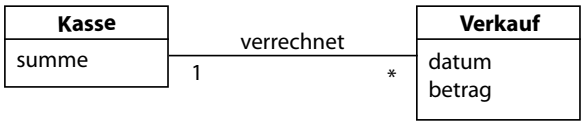
3. funktionale Sicht:

   Wie werden die Methoden implementiert? Welche Algorithmen werden verwendet? Wir wirken die Methoden zusammen, um eine bestimmte Transformation von Informationen zu erreichen?

Darstellung von Entwurfsentscheidungen in UML-Notation

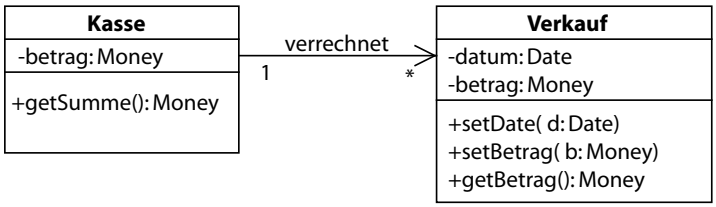
Wenn wir in der Domänenanalyse bisher die UML als Notation verwendet haben, dann können wir diese Darstellung nun weiterverwenden und so erweitern, ergänzen oder umändern, dass die Entwurfsentscheidungen visualisiert werden. Wir starten also mit Der Beschreibung der Dinge der wirklichen Welt und ihrer Beziehungen (aus dem Fachmodell) und machen daraus die Klassen des Entwurfs, die jetzt *Codestrukturen* beschreiben. Weitere Kandidaten für Klassen im Entwurf haben wir in der Szenarioanalyse gefunden.

Zwei Beispiele sollen zeigen, wie sich jetzt die Darstellung verändert:

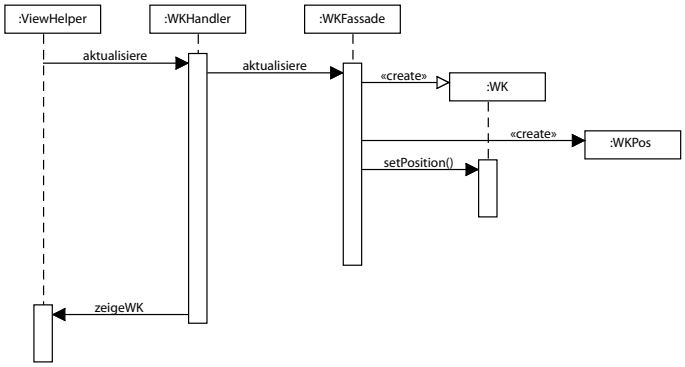


Die vorige Abbildung zeigt (reichlich vereinfacht) das Fachmodell einer Kasse, die viele Verkäufe verrechnet. Die UML wird verwendet, um „Dinge“ im Anwendungsgebiet darzustellen.

In folgender Abbildung machen wir nun daraus ein Klassenmodell für den Entwurf: nun modellieren wir Codestrukturen. Wie man sieht kommt Kapselung hinzu, Sichtbarkeitsbereiche, Methoden, Navigierbarkeit der Assoziation.



Aus unserer Szenarioanalyse „Position in Warenkorb legen“ wird das Sequenzdiagramm



Entwurfsprinzipien

Das Vorgehen im Entwurf entscheidet über die

- Verständlichkeit
- Testbarkeit
- Wiederverwendbarkeit
- Erweiterbarkeit

von Klassen und Komponenten.

Deshalb betrachten wir im Folgenden einige grundlegende Prinzipien, die beim Entwerfen von Klassen berücksichtigt werden. Die ersten fünf dieser Prinzipien nennt Craig Larman in seinem Buch „Applying UML and Patterns“ GRASP = General Responsibility Assignment Software Patterns[1]

Larmans fünf Grundprinzipien sind

1. Informationsexperte
2. Erzeuger
3. Hohe Kohäsion
4. Lose Kopplung
5. Controller

Darüberhinaus kommt noch das sogenannte „Demeter-Gesetz“ vor.

Informationsexperte

Zweck

Eine Verantwortlichkeit (Methode) wird derjenigen Klasse zugeordnet, die über alle Informationen zu ihrer Ausführung verfügt. Also: der *Experte* ist verantwortlich.

Struktur

- Methode zu der Klasse, die über die entsprechenden Attribute verfügt.
- Delegation von Teilaufgaben an assoziierte Klassen – natürlich nach demselben Prinzip.

Eigenschaften

- Kapslung von Information. Das Prinzip entspricht der grundlegenden Idee der Objektorientierung, der Idee des gekapselten Speichers.
- Verhalten wird verstreut, jeweils den Informationsexperten zugeordnet.

[1] grasp engl. packen, fassen, greifen, *fig.* verstehen, begreifen, erfassen

- Vorsicht bei „cross concerns“ wie etwa Datenbankzugriff

Beispiel

Rechnung: Wer ist verantwortlich für den Rechnungsbetrag?

- Rechnung = kennt und ist verantwortlich für Gesamtbetrag
- Position = kennt und ist verantwortlich für Teilbetrag
- Artikel = kennt und ist verantwortlich für Einzelpreis

Erzeuger

Zweck

In objektorientierten Systeme kollaborieren Objekte. Dazu müssen sie erzeugt werden. Wer ist für das *Erzeugen* von Objekten zuständig? Das kann man aus dem Verhältnis der Klassen zueinander ableiten:

Struktur

Eine Klasse E ist geeignet als Erzeuger von Objekten der Klasse K, wenn

- E enthält Objekte der Klasse K (Komposition, Aggregation)
- E verwaltet Objekte der Klasse K
- E ist (einziger) Verwender von K-Objekten
- E hat die Informationen, die zum Erzeugen von K-Objekten benötigt werden (E ist Experte)

Eigenschaften

- Durch das Prinzip des Erzeugers wird die Kopplung der Klassen nicht erhöht: sie besteht ohnehin
- Die Kombination von Aggregation mit Erzeugung entspricht dem Prinzip des gekapselten Speichers
- Bei komplexen Objekten kann man für die Erzeugung eine eigene Klasse vorsehen, man spricht dann von einer *Factory*

Beispiel

Rechnung

Die Rechnung enthält die Positionen, also ist es sinnvoll, wenn die Rechnung auch Positionen erzeugen kann. Etwa durch eine Methode `addRPosition( Artikel a, Menge qty)`

Hohe Kohäsion

Zweck

Kohäsion ist ein Maß dafür, wie eng die Beziehungen und der Fokus der Verantwortlichkeiten einer Klasse oder eines Subsystems sind. Kohäsion steht für die Abhängigkeiten *innerhalb* einer Klasse.

Struktur

- Wenig Kohäsion: Eine Klasse ist verantwortlich für viele Aufgaben in unterschiedlichen Bereichen — vermeiden
- Wenig Kohäsion: Eine Klasse ist für eine sehr komplexe Aufgabe in einem Bereich zuständig — besser zerlegen
- Hohe Kohäsion: Jede Methode einer Klasse hat einen einzigen, klaren Zweck; die Methoden bilden eine Gruppe zusammengehörender Aufgaben

Eigenschaften

- Hohe Kohäsion entspricht dem Prinzip der Modularisierung
- Klassen mit hoher Kohäsion sind gut zu verstehen
- Gute Wartbarkeit und Erweiterbarkeit
- Gute Wiederverwendbarkeit

Beispiel

siehe bei Lose Kopplung

Lose Kopplung

Zweck

Kopplung ist ein Maß dafür, wie sehr ein Element von anderen abhängt. Das Prinzip der *losen Kopplung* fordert: Klassen so definieren, dass nur die *nötige* Kopplung da ist.

Struktur

- Klasse A ist von B abhängig, wenn:
  - A hat Attribut vom Typ B
  - A verwendet eine Methode eines Objekts vom Typ B
  - A hat Methode mit Parameter/Return vom Typ B
  - A ist von B abgeleitet

  - A implementiert die Schnittstelle von B
- Methoden entwerfen, die die (statische) Kopplung im Fachmodell oder Designmodell nicht erhöhen.

Eigenschaften

- Leichte Verstehbarkeit
- Gute Wiederverwendbarkeit
- Nicht tangiert durch Veränderung anderer Klassen
- Spätere Variierbarkeit durch lose Kopplung

Beispiel

Rechnung – Buchung der Bezahlung

Variante 1: Verbucher erzeugt ein Objekt `aZahlung` und übergibt es an Rechnung

Variante 2: Verbucher meldet Zahlung an Rechnung, und die Rechnung erzeugt das Zahlungsobjekt.

Hohe Kopplung führt oft zu Gabelstruktur im Sequenzdiagramm, lose Kopplung zu Treppenstruktur.

Controller

Zweck

Die Anwendung interagiert mit Benutzern. Die Benutzer erzeugen Systemereignisse wie Mausklicks, Texteingabe — wie soll die Anwendung darauf reagieren?

Ein *Controller* ist für die Verarbeitung von Eingaben zuständig und delegiert sie an „interne“ Objekte.

Struktur

- Der Controller ist eine Klasse, die externe Ereignisse z.B. Aktionen einer graphischen Benutzeroberfläche entgegennimmt und an die Anwendung weiterleitet
- Der Controller *delegiert*, er steuert die Aufgaben
- Oft steuert ein Controller genau einen Anwendungsfall

Eigenschaften

- Wiederverwendbarkeit durch Trennung von Oberfläche und Anwendungslogik

- Austauschbarkeit von Oberflächenelementen
- Steuerung des Zustands eines Systems durch einen Controller (beschrieben durch ein Zustandsdiagramm)

Beispiel

Ein *control object*, das einen Anwendungsfall steuert

Das Gesetz von Demeter

Zweck

Verminderung der Kopplung zwischen Klassen, indem nur die minimal benötigte Kenntnis von Schnittstellen verwendet wird

„The LoD (Law of Demeter) is an application of the Low Coupling Principle by making the notion of bad coupling explicit and checkable by tools“ (Karl J. Lieberherr)

Struktur

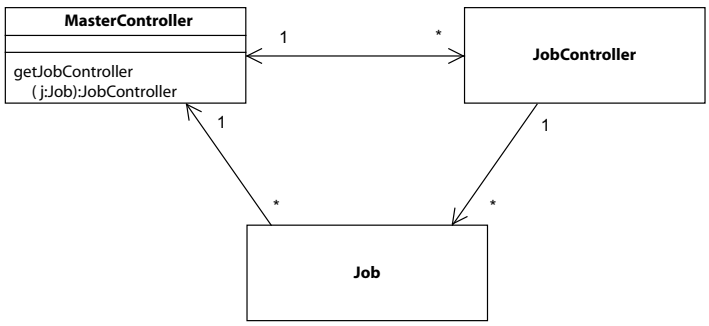
Kurzfassung: „Don't talk to strangers“
Eine Methode m der Klasse K soll nur verwenden

- K selbst, also eigene Methoden
- Argumente, also z.B kann K::m(b: B) eine Methode von B verwenden
- innerhalb von m selbsterzeugte Objekte
- Objekte, auf die ein Objekt von K *direkt* zugreifen kann (also z.B. Attribute oder via Assoziationen)

Eigenschaften

- vermindert Kopplung
- kann durch Werkzeuge überprüft werden (Codeanalyse)

Beispiel



> Wenn nun ein Job-Objekt den JobController verwendet, ist das Demeter-Gesetz nicht mehr erfüllt – deshalb ist dieses Design fragwürdig.

Das Demeter-Gesetz hat Karl J. Lieberherr seit 1987 propagiert – siehe http://www.ccs.neu.edu/home/lieber/LoD.html. Es trägt den Namen nach dem Projekt (Demeter Project), in dem es zum ersten mal konsequent angewandt wurde.

Entwurfsmuster – Ausblick

Wir konnten in der Vorlesung das Gebiet des Entwurfs nur anreissen. Gut strukturierte Entwürfe kann man nicht nur dadurch erreichen, dass man den behandelten Entwurfsprinzipien folgt – sondern insbesondere durch das Studieren, Verstehen und Verwenden von gut ausgearbeiteten und durchdachten Entwürfen. Man spricht von Entwurfsmustern, wenn erprobte Lösungen von wiederkehrenden, typischen Fragestellungen in einem bestimmten Kontext in einer verständlichen Form ausgearbeitet wurden.

Solche Entwurfsmuster kann man in Projekten einsetzen. Entwurfsmuster haben jeweils einen Namen, unter dem man folglich auch komplexere Lösungen kurz und treffend ansprechen kann. So ist es unter Entwicklern, die Entwurfsmuster kennen, üblich sich etwa so zu unterhalten: „Ich denke, hier sollten wir einen Visitor nehmen. . . “.

Das bekannteste und wichtigste Buch über Entwurfsmuster ist:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*

Dieses Buch sollte jeder Softwareentwickler kennen. Mehr darüber in der nächsten Vorlesung.

Lektüreempfehlung zu dieser Vorlesung: Kapitel 10 im Buch von Zuser u.a.

Burkhardt Renz
Technische Hochschule Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 2.1 – 16. Mai 2012