Lösungen zu den Wiederholungsfragen und Aufgaben

Im Folgenden finden Sie alle Lösungen zu den Wiederholungsfragen und Aufgaben der einzelnen Kapitel des Buches. Bei den Lösungen der Aufgaben handelt es sich um Vorschläge, das heißt es können auch abweichende Lösungen richtig sein.

1. Lösungen zu Fragen und Aufgaben von Kapitel 1

Fragen:

1.1 c)
1.2 b)
1.3 b)

Aufgabe 1.1

Die Erfolge über die Matthias berichtet sind vielleicht dadurch zu erklären, dass es sich um kleinere Projekte handelte, bei denen die Anforderungen überschaubar komplex und/oder wohldefiniert waren. Es könnte sich auch um Anpassungen aufgrund von wohldefinierten Änderungen oder um abgrenzbare Erweiterungen gehandelt haben. Handelt es sich bei den Auftraggebern um sehr erfahrene Anwender, die schon bei vielen Software-Projekten mitgewirkt haben, so wären die Erfolge sicherlich auch erklärbar. Ansonsten führt die Anwendung des Wasserfallmodells dazu, dass die am Projektanfang aufgeschriebenen Anforderungen vielfach keine stabile Grundlage für das Projekt liefern können, schwer vorhersehbare Neuerungen und Änderungen nicht berücksichtigt werden können oder die Unzulänglichkeiten oft erst sehr spät entdeckt werden.

Die erwähnten Projekte von Steffen haben offenbar eine längere Laufzeit, so dass genau die oben genannten Schwachstellen auftreten können. Der Unified Process könnte durch seine Iterationen im Sinne von kleinen, aufeinander aufbauenden Mini-Projekten mit einer Laufzeit von wenigen Wochen eine Verbesserung bringen. Insbesondere würde das Risiko, dass man am Projektende nicht das abliefert, was der Auftraggeber braucht, reduziert. Änderungen während der Laufzeit des Gesamtprojektes können eher berücksichtigt werden. Die Anwender könnten im Projektablauf lernen und Erfahrungen sammeln, so dass sie hinsichtlich der vollständigen und detaillierten Anforderungsbeschreibung am Anfang des Projektes nicht überfordert werden.

Aufgabe 1.2

Betrachtet man die Software-Entwicklungszeit vom Projektstart bis zur Auslieferung des Produkts, so werden in dieser Zeitperiode formale Maßnahmen zur Qualitätssicherung durchgeführt, z.B. Qualitäts-Reviews. Bei diesen formalen Qualitätssicherungsmaßnahmen werden Fehler bzw. Änderungsnotwendigkeiten aufgedeckt, die Anzahl sei F_1. Nach Auslieferung der Software an den Auftraggeber entdeckt der Anwender auch Fehler, die Anzahl sei F_2. Wird nun die Fehlerrate als der Quotient aus F_1 und (F_1+F_2) ermittelt, so sagt eine Fehlerrate von 1 aus, dass alle Fehler vor der Auslieferung entdeckt und sicherlich auch beseitigt wurden. Wie lange die Zeit zur Ermittlung von F_2 festgelegt wird, hängt sicherlich vom Einzelfall ab.

Die Aktivitäten, die mit Analyse und Entwurf zusammengefasst werden, dienen dazu festzulegen, wie die Anforderungen an das zu entwickelnde System umgesetzt werden sollen. Zeitlich gesehen sollte dies im Rahmen mehrerer Iterationen überwiegend in der Spezifikationsphase erfolgen. Allerdings kann es durchaus sein, dass auch in der Phase Konstruktion im Rahmen nachgelagerter Iterationen noch Analyse- und Entwurfsaktivitäten durchgeführt werden. Dies ergibt sich daraus, dass im Unified Process, die einzelnen Phasen nicht durch eine Aktivität gekennzeichnet sind, sondern, dass Aktivität und Phase im Gegensatz zum Wasserfallmodell grundsätzlich auseinanderfallen. Allerdings haben die einzelnen Tätigkeiten ihren Schwerpunkt in bestimmten Phasen.

Bei einem iterativen Vorgehen wird ja nicht vorausgesetzt, dass die Anforderungsanalyse im Sinne einer Phase zu einem bestimmten Zeitpunkt vollständig abgeschlossen ist, so dass auf dieser Basis eine Vergabe an einen externen Auftragnehmer vorgenommen werden könnte. Das bedeutet, dass es sinnvoll ist, mehrere Verträge abzuschließen. Voraussetzung dafür ist, dass nach einer Vorbereitungsphase ein Überblick darüber besteht, was durch das geplante Anwendungssystem grundsätzlich geleistet werden soll. Darüber hinaus ist es erforderlich, dass die Anforderungen in Form von Anwendungsfällen mehr oder weniger ausführlich beschrieben sind. Besondere Wichtigkeit erlangt das Festlegen von Prioritäten für die Anwendungsfälle. Entsprechend dieser Prioritäten werden die Anwendungsfälle in einzelnen Iterationen detailliert und realisiert. Dabei ist es empfehlenswert, dass diese Iterationen exakt terminiert und budgetiert werden, weiterhin sind Qualitätseigenschaften zu vereinbaren. Auf dieser Basis können Verträge abgeschlossen werden. Eine Unsicherheit besteht natürlich darin, dass der Inhalt und Umfang der angestrebten Software-Lösung u.U. nicht vollständig und detailliert beschrieben ist. Die Gefahr, dass der Auftragnehmer jedoch unangemessen wenig liefert wird dadurch verringert, dass dieser ja durchaus Interesse daran hat auch den Folgeauftrag für die nächste(n) Iteration(en) zu bekommen. U.U. kann es sein, dass am Ende der geplanten Gesamtprojektzeit vielleicht nur 80 % des ursprünglich anvisierten Umfangs erreicht werden kann. Allerdings sind es dann eher die 80 %,

die einerseits wichtig (hohe Priorität) sind und andererseits auch die aktuellen Anforderungen abdecken und nicht die Anforderungen, die man vor zwei oder drei Jahren detailliert spezifizierte und von denen man glaubte, dass sie den Bedarf in zwei oder drei Jahren abdecken könnten und heute noch geringe Relevanz haben.

2. Lösungen zu Fragen und Aufgaben von Kapitel 2

Fragen:

2.1 a); 2.2 c); 2.3 e)

Aufgabe 2.1

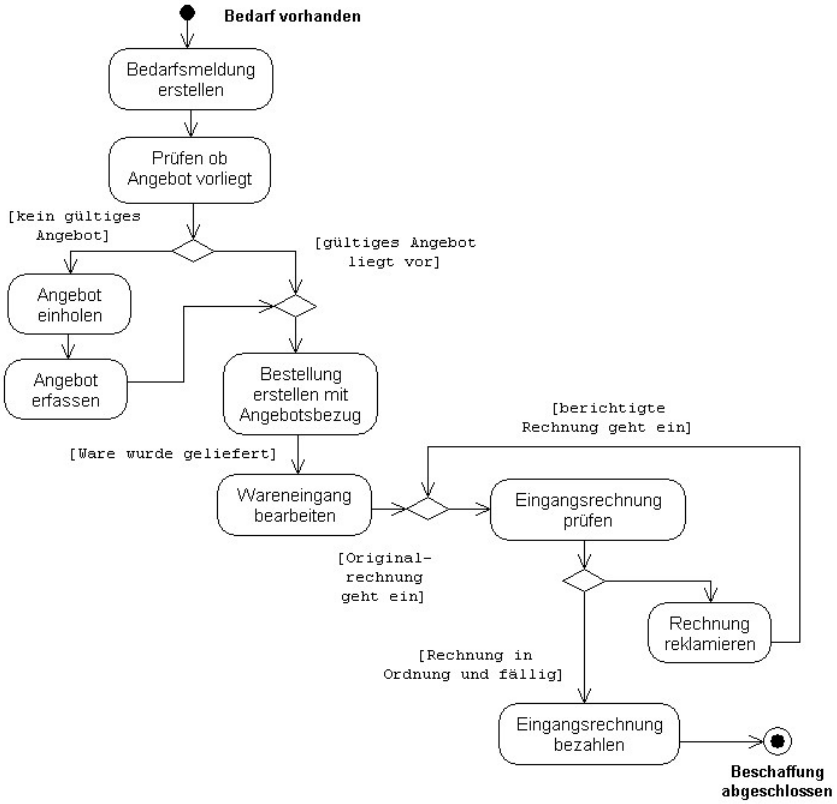

Abb. 1: Geschäftsprozess *Beschaffung durchführen* (ergänzt)

Aufgabe 2.2

Ergänzte Spezifikation des Systemanwendungsfalls a)

Tabelle 1: Anwendungsfall *Eingangsrechnung prüfen* (ergänzt)

Anwendungsfall:	Eingangsrechnung prüfen
Ziel:	Ordnungsgemäße Verbuchung der Eingangsrechnung
Vorbedingungen:	zugehörige Bestellung und evtl. Wareneingänge im System erfasst
Nachbedingung Erfolg:	Rechnung ist im System erfasst, Bestellung als fakturiert gekennzeichnet sowie Rechnung in der Kreditorenbuchhaltung erfasst und zur Zahlung freigegeben.
Nachbedingung Fehlschlag:	Differenzen zwischen Bestellung bzw. Wareneingang und Rechnung sind identifiziert und stehen zur Reklamation der Rechnung zur Verfügung.
Akteure:	Kreditorenbuchhalter
Auslöser:	Lieferantenrechnung ist eingegangen.
Beschreibung:	1. Suche der zugehörigen Bestellung über Bestellnummer 2. Suche der zugehörigen Wareneingänge 3. Eingabe der in Rechnung gestellter Menge und des Preises sowie Abgleich mit Bestell- und Wareneingangsdaten 4. Verbuchen der Rechnung
Alternativen:	2a) Abbruch, wenn keine Bestellung vorliegt. Weiter mit 4b. 3b) Abbruch, wenn kein Wareneingang vorliegt. Weiter mit 4b. 4b) Abbruch, wenn Abweichung größer als Abweichungstoleranz
Erweiterungen:	1a) Suche über Lieferantennummer bzw. Artikelnummer, falls Bestellnummer nicht auf Rechnung vermerkt.
Priorität:	hoch, wegen Buchhaltungspflicht und Zahlungsverpflichtung
Häufigkeit:	hoch, mehrere Hundert Eingangsrechnungen/Monat
Offene Punkte:	zulässige Abweichungstoleranzen sind noch zu klären
Sonstiges:	-

Ergänztes Anwendungsfalldiagramm

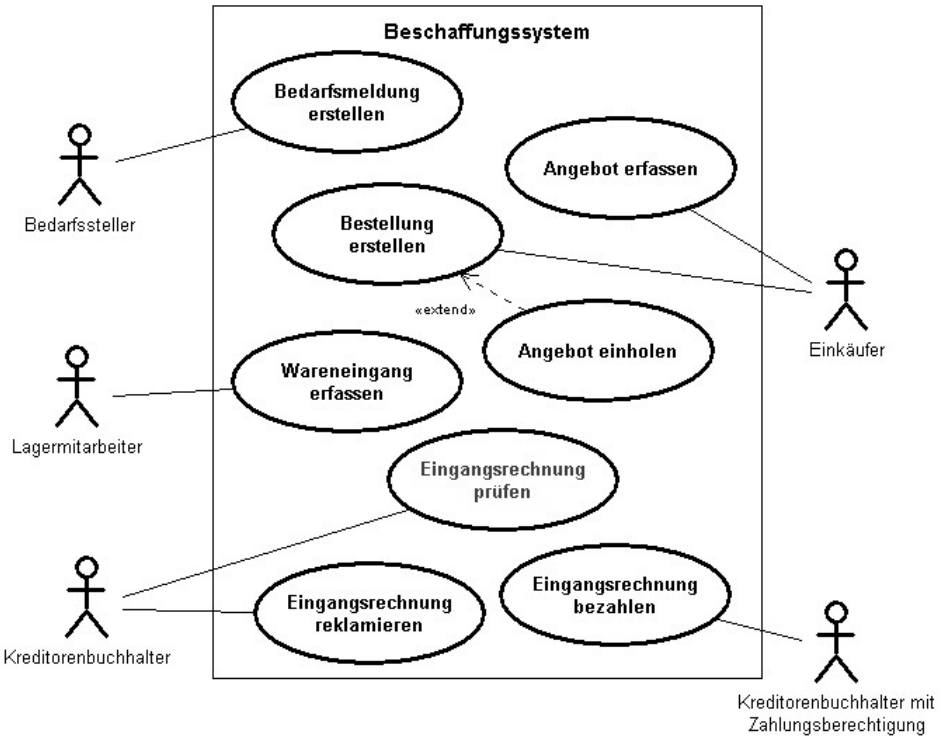

Abb. 2: Anwendungsfalldiagramm *Beschaffungssystem* (ergänzt)

3. Lösungen zu Fragen und Aufgaben von Kapitel 3

Fragen:

3.1 b)
3.2 a)
3.3 c)

Aufgabe 3.1

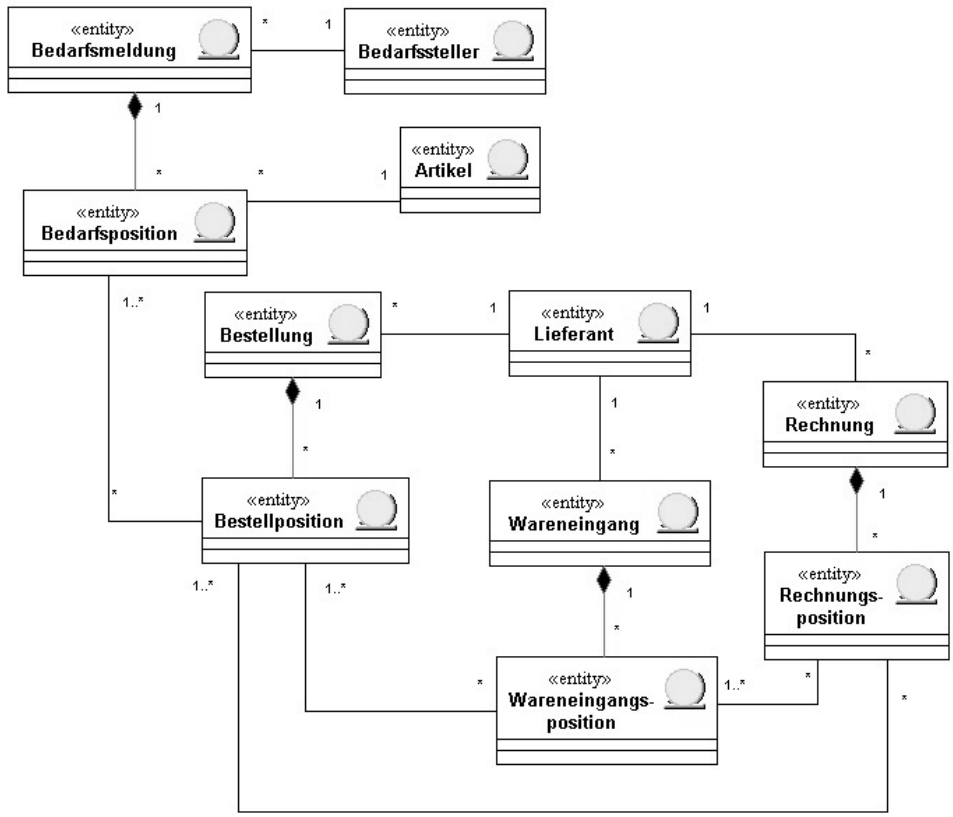

Abb. 3: Klassendiagramm *Beschaffungssystem*

Aufgabe 3.2

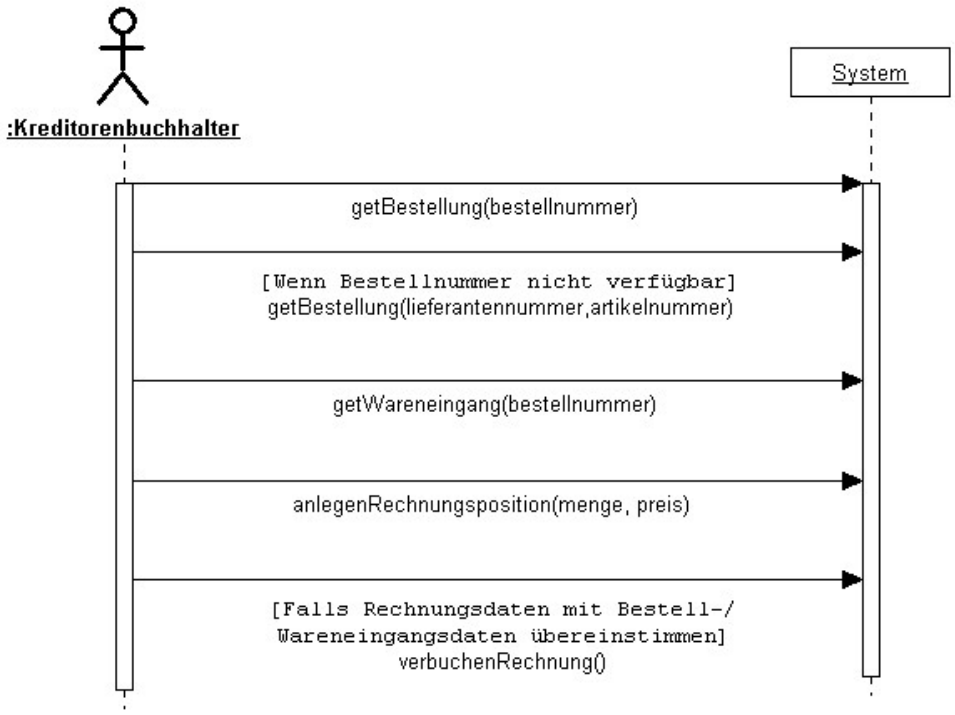

Abb. 4: System-Sequenzdiagramm ***Eingangsrechnung prüfen***

4. Lösungen zu Fragen und Aufgaben von Kapitel 4

Fragen:

4.1 d)
4.2 a)
4.3 a)

Aufgabe 4.1

Die **Sichtbarkeit über ein Attribut** ist eine sehr gebräuchliche Form der Sichtbarkeit in objektorientierten Systemen. Entsprechend dem Klassendiagramm in Abb. 3 wird in der Klasse *Bedarfsmeldung* ein Attribut vom Typ *Bedarfssteller* definiert. Damit hätte ein Objekt vom Typ *Bedarfsmeldung* eine Sicht auf das zugehörige *Bedarfssteller*-Objekt.

Die **Sichtbarkeit über Parameter** erlaubt dem Objekt A auf das Objekt B dadurch zuzugreifen, dass B als Parameter einer Operation von A übergeben wird. Damit handelt es sich um eine relativ beschränkte Sichtbarkeit, da diese nur in der Reichweite der Operation gegeben ist.

Die **lokale Sichtbarkeit** von Objekt A auf Objekt B wird dadurch erreicht, dass B als lokales Attribut in einer Operation der Klasse A deklariert wird. Dadurch ist die Gültigkeit der Sichtbarkeit auf die Reichweite der betreffenden Operation beschränkt. Die lokale Sichtbarkeit kann entweder dadurch erreicht werden, dass ein lokales Objekt B erzeugt und dem lokalen Attribut zugeordnet wird oder dadurch, dass dem lokalen Attribut das Objekt B als Rückgabewert eines Operationsaufrufs zugewiesen wird. Dies trifft sicherlich für die Operation *anlegenRechnungsposition()* der Klasse *EingangsrechnungPruefenHandler*. Diese bekommt als Ergebnis jeder Nachricht *getBestellposition()* ein Objekt vom Typ *Bestellposition* zurückgeliefert, das sie in einer Liste sammelt und an das Objekt der Klasse *Rechnung* weiterreicht.

Die **globale Sichtbarkeit** eines Objektes B für ein Objekt A existiert solange wie das Objekt B existiert. Es ist die am wenigsten übliche Form der Sichtbarkeit in objektorientierten Systemen. Nicht in allen Programmiersprachen sind globale Variablen möglich. Während dies in C++ möglich ist, trifft dies für Java nicht zu. Eine gerne verwendete Lösung globale Sichtbarkeit in objektorientierten Systemen zu ermöglichen ist die Verwendung des so genannten Singleton-Musters.

Aufgabe 4.2

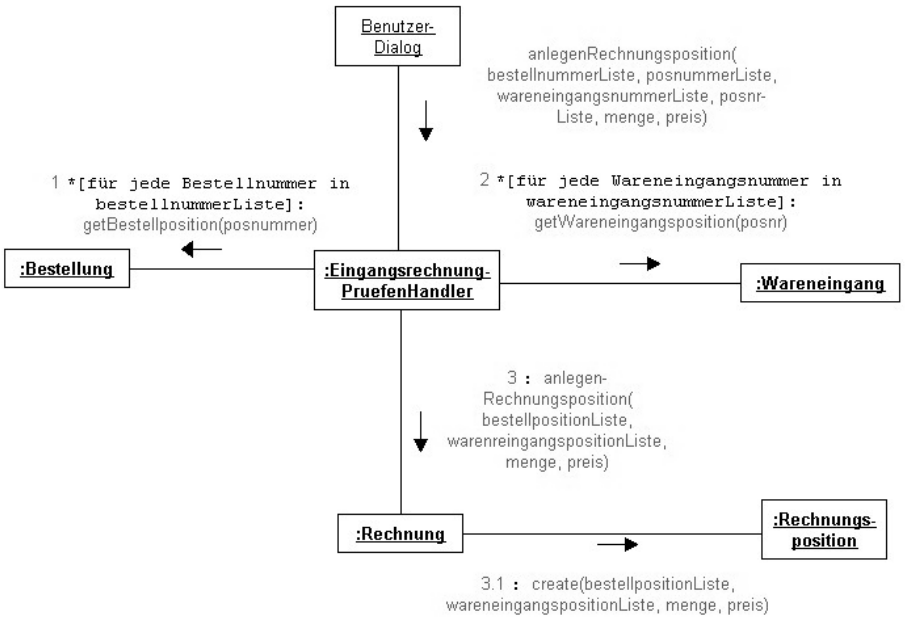

Abb. 5: Kommunikationsdiagramm zur System-Operation *anlegenRechnungsposition()*

5. Lösungen zu Fragen und Aufgaben von Kapitel 5

Fragen:

5.1 a)
5.2 b)
5.3 d)

Aufgabe 5.1

Der Dialog besteht im Prinzip aus drei Teilen. Der obere Teil betrifft Attribute der Bedarfsmeldung, im mittleren Teil werden die Positionsdaten erfasst und im unteren Teil werden die erfassten Positionsdaten zusammenfassend dargestellt. Während der untere und mittlere Teil aufgrund unterschiedlicher Struktur leichter unterscheidbar ist, wurde zwischen dem oberen und mittleren Teil eine Trennlinie eingezeichnet. Die Führungstexte wurden vereinheitlicht und die Ausrichtung

aufgrund stark unterschiedlicher Länge verändert. Die Schaltflächen wurden einheitlich bezeichnet und möglichst so angeordnet, dass ihre Zuordnung einfach ist. In der Positionsübersicht wurde vermieden unübliche Abkürzungen zu verwenden.

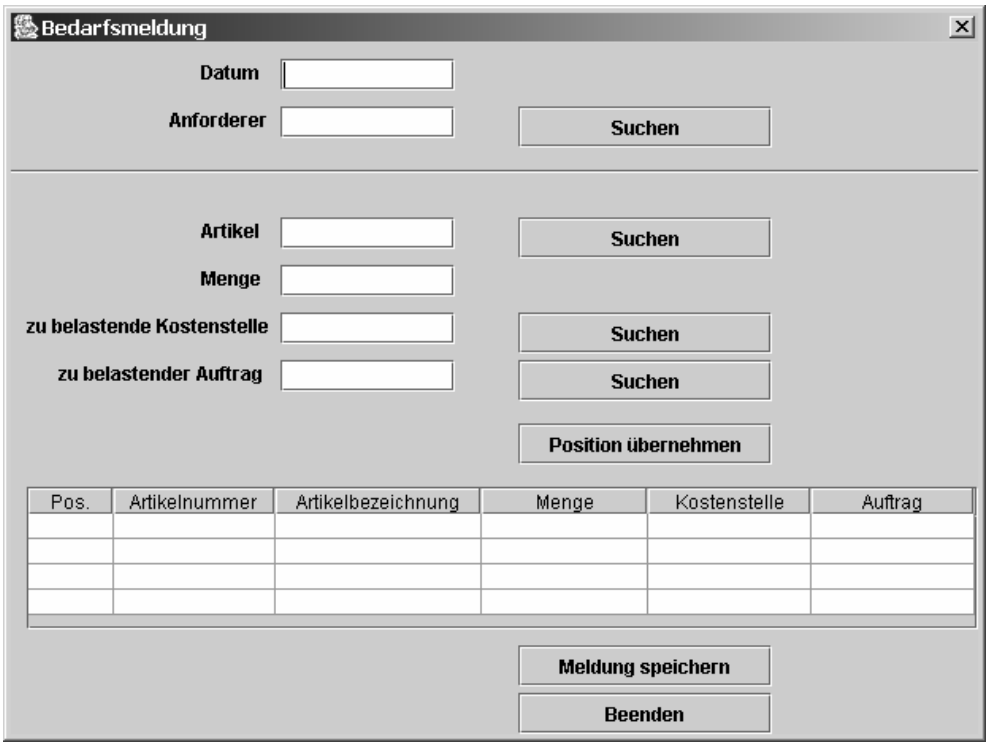

Abb. 6: Dialogentwurf *Bedarfsmeldung erfassen* (verbessert)

Aufgabe 5.2

Bei einer umfassenden Anwendung kann es sein, dass unterschiedliche Anwendungsfallmodelle für einzelne Subsysteme (Geschäftsprozesse) aufgestellt werden. Diese finden sich typischerweise im Anwendungsdialog als Menüpunkte wieder. In den einzelnen Anwendungsmenüs orientieren sich die Menüpunkte an den einzelnen Anwendungsfällen. Die Ablaufbeschreibung des Anwendungsfalls korrespondiert unmittelbar mit der Interaktions- bzw. Dialogfolge. Damit wird deutlich, dass die Anwendungsfallmodellierung prägenden Einfluss auf die Dialoggestaltung hat.

6. Lösungen zu Fragen und Aufgaben von Kapitel 6

Fragen:

6.1 b)
6.2 a)
6.3 b)

Aufgabe 6.1

Eine CASE-Umgebung basiert auf einer CASE-Plattform, welche die Integration einzelner CASE-Werkzeuge erlaubt.

Aufgabe 6.2

Bei MDA wird von einem evolutionären Schritt in der Software-Entwicklung gesprochen. Es wird ein Vergleich zu den Ursprüngen der speicherprogrammierten Rechenanlagen der späten 40er Jahre des letzten Jahrhunderts gezogen, bei denen das erste Mal vorgefertigte Programme und Bibliotheksmodule für immer wiederkehrende Rechenaufgaben wiederverwendet werden konnten. Der nächste Schritt war die Entwicklung höherer Programmiersprachen, z.B. Fortran durch John Backus im Jahr 1954, womit Programme einmal entwickelt und mittels plattformspezifischen Compilern auf unterschiedlichen Rechnern eingesetzt werden konnten. In diesem Sinne postuliert die MDA, dass eine Software-Lösung fachlich so detailliert beschrieben wird, dass mit Hilfe eines Plattform-Modells (dem Compiler vergleichbar) ein implementierungsabhängiges Modell (PSM) generiert werden könnte. Statt eines PSM könnte man sich vorstellen, dass auch unmittelbar Quellcode generiert wird. Dies würde ein großer Schritt in Richtung industrieller Software-Entwicklung bedeuten.

7. Lösungen zu Fragen und Aufgaben von Kapitel 7

Fragen:

7.1 b) und e)
7.2 a) und e)
7.3 a), c), d) und e)

Aufgabe 7.1

Die von Matthias genannten Ziele wie hohe Benutzerfreundlichkeit, hoher Grad an Ordnungsmäßigkeit und hohe Sicherheit sind sicherlich wichtige Ziele und wir

haben diese auch im Rahmen der Anforderungsanalyse als Software-Qualitätsmerkmale kennen gelernt. Diese Ziele beschreiben Merkmale der Software, welche der Auftraggeber bzw. der Benutzer als sichtbare Eigenschaften wahrnehmen kann. Termintreue ist ein Zielinhalt, der sich nicht auf das Software-Produkt, sondern auf den Prozess der Software-Erstellung bezieht. Auch dies ist für den Auftraggeber direkt spürbar. Wird die Software nicht rechtzeitig fertig, so kann das enorme wirtschaftliche Nachteile für den Auftraggeber bedeuten. Leichte Änderbarkeit, gute Wartbarkeit und hohe Wiederverwendbarkeit sind demgegenüber Eigenschaften, welche vom Auftraggeber bzw. Benutzer nicht unmittelbar wahrgenommen werden. Ein Mehr oder Weniger an Änderbarkeit drückt sich in dem Aufwand aus, der notwendig ist, um geforderte/notwendige Änderungen an einem Software-System vorzunehmen. Ähnlich verhält sich dies mit der Wartbarkeit, d.h. dem Aufwand, eine Wartungsanforderung, z.B. Fehlerbeseitigung, an einer Software zu erfüllen. Das bedeutet, dass derartige Eigenschaften insbesondere nach der erstmaligen Erstellung einer Software relevant werden. Ist die Änderbarkeit schwer, also der Aufwand für Änderungen hoch, so spürt dies der Auftraggeber dann, wenn Änderungen notwendig werden. Diese können durch Änderungen rechtlicher Regelungen oder notwendigen Änderungen der Geschäftspraxis und damit verbundenen Veränderungen in den Geschäftsprozessen verursacht sein. Die Wiederverwendbarkeit ist von den beiden anderen Zielen nicht unabhängig. Ist der Grad der Wiederverwendung einmal entwickelter Software-Komponenten in einem Software-System hoch, so wirkt sich dies auch positiv auf die Änderbarkeit bzw. Wartbarkeit aus. Denn wenn bei Änderungen Software geändert wird, die wiederverwendet wird, ist diese Änderung ja nur einmal durchzuführen, jedoch vielfach wirksam. Damit wird der mit Änderungen verbundene Aufwand auch geringer. Darüber hinaus kann die Wiederverwendung bereits existenter Software-Komponenten auch den erstmaligen Erstellungsaufwand für das Software-System reduzieren. Allerdings kann das Erstellen von wiederverwendbarer Software aufwändiger sein. Dies ist dadurch bedingt, dass der Abstraktionsgrad wiederverwendbarer Software i.d.R. höher ist. Damit wird deutlich, dass es für den Auftraggeber sehr wohl sinnvoll sein kann, leichte Änderbarkeit, hohe Wartbarkeit und hohe Wiederverwendbarkeit anzustreben. Dies stimmt insbesondere bei längerfristiger Betrachtung. Handelt es sich um Wegwerf-Software, z.B. eine temporäre Schnittstelle für die nächsten drei Monate, so haben diese Ziele sicherlich weniger Bedeutung.

Die unmittelbare Auswirkung der Ziele 'leichte Änderbarkeit', 'gute Wartbarkeit' und 'hohe Wiederverwendbarkeit' auf die Wirtschaftlichkeit, ergibt sich unmittelbar aus ihrer Definition. Eine einfache oder leichte Änderbarkeit bzw. gute Wartbarkeit drückt sich darin aus, dass der Aufwand für die Änderung bzw Ausführung der Wartungsaufgabe gering ist. Dieser Aufwand drückt sich primär in der Zeit aus, welche für die Änderung notwendig ist. Diese Zeit bedeutet Einsatz von

menschlicher Arbeitsleistung, welche zu Personalaufwand bzw. Personalkosten führt. Damit erklärt sich auch das Interesse der Geschäftsleitung an dem Erreichen derartiger Ziele. Da eine hohe Wiederverwendbarkeit zu einem hohen Anteil wiederverwendeter Software-Komponten in einem Software-System führen kann, wirkt dies, wie oben bereits erwähnt, positiv auf die Änderbarkeit bzw. Wartbarkeit. Damit wirkt auch Wiederverwendbarkeit langfristig auf die Reduzierung des Aufwands bzw. der Kosten.

Aufgabe 7.2

Tabelle 2: Tabellenschema

Konto(kontonummer, bezeichnung, kontoumsatz)
Beleg(belegnummer, belegdatum)
Belegzeile(positionsnummer, fkBelegnummer, betrag, fkKontonummer)
Sachkonto(fkKontonummer, kontoart, abstimmkontoFuerKontoart, buchungskreis)
Kreditor(fkKontonummer, umsatzsteuerId, fkDebitorenKontonummer,fkSachkontoAbstimmkonto)
Debitor(fkKontonummer, kreditlinie, fkKreditorenKontonummer, fkSachkontoAbstimmkonto)
Zahlungsbedingung(id, tage, prozent)
KreditorenZahlungsbedingung(fkKreditorenKontonummer, fkZahlungsbedingungId)
DebitorenZahlungsbedingung(fkDebitorenKontonummer, fkZahlungsbedingungId)

Bei der obigen Lösung wurde dem Class Table Inheritance-Muster gefolgt. Da nur wenige Attribute beispielhaft aufgeführt sind, kann nicht begründet entschieden werden, ob nicht das Single Table Inheritance-Muster auch sinnvoll wäre. Unterstellt man, dass die Attributstruktur relativ stabil ist, so wäre auch das Concrete Table Inheritance-Muster denkbar gewesen.

8. Lösungen zu Fragen und Aufgaben von Kapitel 8

Fragen:

8.1 b) und e)
8.2 a), b) und e)
8.3 a), c) und d)

Aufgabe 8.1

```
import java.util.*;
public class Gast implements Observer{
    private String name;
    private String vorname;
    private double rabatt;
    private Vector<Buchung> buchungen;
    private Integer nummer;
    private double umsatz = 0;
    public Gast(Integer nummer,String name, String vorname, double rabatt) {
        setNummer(nummer);
        setName(name);
        setVorname(vorname);
        setRabatt(rabatt);
        buchungen=new Vector<Buchung>();
    }
    public Vector<Buchung> getBuchungen() {
        return buchungen; }
    public void setBuchungen(Vector<Buchung> buchungen) {
        this.buchungen = buchungen; }
    public java.lang.String getName() {
        return name; }
    public void setName(java.lang.String name) {
        this.name = name; }
    public double getRabatt() {
        return rabatt;  }
    public void setRabatt(double rabatt) {
        this.rabatt = rabatt;  }
    public java.lang.String getVorname() {
```

```
        return vorname; }
    public void setVorname(java.lang.String vorname) {
        this.vorname = vorname;  }
    public java.lang.Integer getNummer() {
        return nummer; }
    public void setNummer(java.lang.Integer nummer) {
        this.nummer = nummer; }
    public double getUmsatz() {
        return umsatz; }
    public void setUmsatz(double umsatz) {
        this.umsatz = umsatz; }
    public void update(double betrag) {
        umsatz=umsatz+betrag;
    }
}
import java.util.*;
public class Buchung extends Observable  {
    private Integer buchungsnummer;
    private String datum;
    private Gast gast;
    private Vector<Buchungsposition> positionen;
    public Buchung(Integer buchungsnummer,String datum, Gast gast) {
        setBuchungsnummer(buchungsnummer);
        setDatum(datum);
        setGast(gast);
        positionen=new Vector<Buchungsposition>();
        this.attach(gast);
        this.attach(Hotel.getHotel()); }
    public double rechnungssummeNetto() {
        double summeOhneAktion=0, summeMitAktion=0;
        for(Buchungsposition eineBuchungsposition : positionen){
            if(eineBuchungsposition.getAktionsrabatt()==0){

summeOhneAktion=summeOhneAktion+eineBuchungsposition.positionsbetrag();
            }
            else{

summeMitAktion=summeMitAktion+eineBuchungsposition.positionsbetrag();
```

```
            }
        }
        return summeMitAktion+summeOhneAktion*(100-gast.getRabatt())/100;
    }
    public Integer getBuchungsnummer() {
        return buchungsnummer;  }
    public void setBuchungsnummer(Integer buchungsnummer) {
        this.buchungsnummer = buchungsnummer; }
    public java.lang.String getDatum() {
        return datum; }
    public void setDatum(java.lang.String datum) {
        this.datum = datum; }
    public Gast getGast() {
        return gast; }
    public void setGast(Gast gast) {
        this.gast = gast; }
    public java.util.Vector getPositionen() {
        return positionen; }
    public void setPositionen(java.util.Vector positionen) {
        this.positionen = positionen; }
    public void anlegenBuchungsposition(double menge,
      double aktionsrabatt, Leistung leistung) {
        Buchungsposition neueBuchungsposition= new
Buchungsposition(menge,aktionsrabatt,leistung);
        if(neueBuchungsposition.getAktionsrabatt()!=0){
          fireUpdate(neueBuchungsposition.positionsbetrag());
        }
        else{
            fireUpdate(neueBuchungsposition.positionsbetrag()*(100-
getGast().getRabatt())/100);
        }
        positionen.addElement(neueBuchungsposition);
    }
}
public class Buchungsposition {
    private double menge;
    private double aktionsrabatt;
    private Leistung leistung;
```

```
    public Buchungsposition(double menge,double aktionsrabatt, Leistung
leistung) {
        setMenge(menge);
        setAktionsrabatt(aktionsrabatt);
        setLeistung(leistung);
    }
    public double positionsbetrag() {
        return getMenge()*leistung.getPreis()*(100-aktionsrabatt)/100;
    }
    public double getAktionsrabatt() {
        return aktionsrabatt; }
    public void setAktionsrabatt(double aktionsrabatt) {
        this.aktionsrabatt = aktionsrabatt;   }
    public Leistung getLeistung() {
        return leistung; }
    public void setLeistung(Leistung leistung) {
        this.leistung = leistung; }
    public double getMenge() {
        return menge; }
    public void setMenge(double menge) {
        this.menge = menge; }
}
public class Leistung {
    private Integer nummer;
    private String bezeichnung;
    private double preis;
    public Leistung(Integer nummer, String bezeichnung,double preis) {
        setNummer(nummer);
        setBezeichnung(bezeichnung);
        setPreis(preis);
    }
    public java.lang.String getBezeichnung() {
        return bezeichnung; }
    public void setBezeichnung(java.lang.String bezeichnung) {
        this.bezeichnung = bezeichnung; }
    public Integer getNummer() {
        return nummer; }
    public void setNummer(Integer nummer) {
```

```
        this.nummer = nummer; }
    public double getPreis() {
        return preis; }
    public void setPreis(double preis) {
        this.preis = preis; }
}


import java.util.*;
public abstract class Observable {
    private Vector<Observer> observers = new Vector<Observer>();
    public Observable() { }
    public void attach(Observer observer) {
        observers.addElement(observer);}
    public void detach(Observer observer) {
        observers.remove(observer);}
    public void fireUpdate(double umsatz) {
       for(Observer observer : observers){
           observer.update(umsatz);
        }
    }}
public interface Observer {
    void update(double betrag);
}
import java.util.*;
public class Hotel implements Observer {
    private Hashtable<Integer, Leistung> leistungen;
    private Hashtable<Integer, Gast> gaeste;
    private Hashtable<Integer, Buchung> buchungen;
    private static Hotel hotel;
    private double umsatz = 0;
    private Hotel() {
        setBuchungen(new Hashtable<Integer, Buchung>());
        setGaeste(new Hashtable<Integer, Gast>());
        setLeistungen(new Hashtable<Integer, Leistung>());
}
    public static Hotel getHotel(){
```

```
    if (hotel==null){
      hotel=new Hotel();
    }
    return hotel; }
void anlegenLeistung(Integer nummer,String bezeichnung,double preis) {
    getLeistungen().put(nummer, new Leistung(nummer,bezeichnung,preis));
}

void anlegenGast(Integer nummer, String name, String vorname,
  double rabatt) {
    getGaeste().put(nummer,new Gast(nummer, name,vorname,rabatt));
}
void anlegenBuchung(Integer buchungsnummer,String datum,Gast gast) {
    Buchung neueBuchung=new Buchung(buchungsnummer,datum,gast);
    getBuchungen().put(buchungsnummer, neueBuchung);
    gast.getBuchungen().addElement(neueBuchung);
}
public double getUmsatz() {
    return umsatz; }
public void setUmsatz(double umsatz) {
    this.umsatz = umsatz; }
public void update(double betrag) {
    umsatz=umsatz+betrag; }
public Hashtable<Integer, Leistung> getLeistungen() {
    return leistungen;
}
public void setLeistungen(Hashtable<Integer, Leistung> leistungen) {
    this.leistungen = leistungen;
}
public Hashtable<Integer, Gast> getGaeste() {
    return gaeste;
}
public void setGaeste(Hashtable<Integer, Gast> gaeste) {
    this.gaeste = gaeste;
}
public Hashtable<Integer, Buchung> getBuchungen() {
    return buchungen;
```

```
    }
    public void setBuchungen(Hashtable<Integer, Buchung> buchungen) {
        this.buchungen = buchungen;
    }
}
```

Aufgabe 8.2

a)

```
public abstract class Teil {
    private int nummer;
    private String bezeichnung;
    public Teil() { }
    public Teil(int nummer, java.lang.String name) {
        this.nummer=nummer;
        this.bezeichnung=name;  }
    public abstract double getMek();
    public abstract void dazu(Teil teil);
    public abstract void weg(Teil teil);
}
public class Fremdbezugsteil extends Teil {
    private double preis;
    public Fremdbezugsteil() {}
    public Fremdbezugsteil(int nummer, java.lang.String bezeichnung, double
preis) {
        super(nummer,bezeichnung);
        this.preis=preis; }
    public double getMek() {
        return preis; }
    public void weg(Teil teil) {}
    public void dazu(Teil teil) {}
}
import java.util.*;
public class Eigenfertigungsteil extends Teil {
    private Vector<Teil> einzelteile;
    public Eigenfertigungsteil() { }
    public Eigenfertigungsteil(int nummer, java.lang.String name) {
```

```
        super(nummer,name);
        einzelteile=new Vector<Teil>();}
    public void dazu(Teil teil) {
        einzelteile.addElement(teil); }
    public double getMek() {
        double summe=0;
        for(Teil einTeil : einzelteile) {
            summe+=einTeil.getMek(); }
        return summe; }
    public void weg(Teil teil) {
        einzelteile.removeElement(teil); }
}
public class KompositumTest {
    public KompositumTest() {}
    public static void main(String[] args) {
        Fremdbezugsteil bein=new Fremdbezugsteil(1,"Tischbein",0.50);
        Fremdbezugsteil rahmen=new Fremdbezugsteil(2, "Rahmen", 1.0);
        Eigenfertigungsteil gestell=new Eigenfertigungsteil(3,"Gestell");
        for(int i=0;i<4;i++){
            gestell.dazu(bein); }
        gestell.dazu(rahmen);
        Fremdbezugsteil schraube=new Fremdbezugsteil(4,"Schraube",0.05);
        Fremdbezugsteil platte=new Fremdbezugsteil(5, "Tischplatte", 5.0);
        Eigenfertigungsteil tisch=new Eigenfertigungsteil(6,"Tisch");
        for(int i=0;i<8;i++){
            tisch.dazu(schraube); }
        tisch.dazu(platte);
        tisch.dazu(gestell);
        System.out.println("MEK vom Tisch: "+tisch.getMek());
        System.out.println("MEK vom Gestell: "+gestell.getMek()); }}
```

b)

```
public class Kontenplan {
....private  TreeMap<Integer, Konto> angelegteKonten;
....private static Kontenplan einzigerKontenplan=null;
private Kontenplan() {
....setAngelegteKonten(new TreeMap<Integer, Konto>());}
```

```
public static Kontenplan getEinzigerKontenplan(){
....if (einzigerKontenplan(=)=null) {
        einzigerKontenplan = new Kontenplan(); }
        return einzigerKontenplan;
} }
```

9. Lösungen zu Fragen und Aufgaben von Kapitel 9

Aufgabe 9.1

Auf der Webseite finden Sie den Code im Projekt WABOOPAufgabe. Die Lösung wurde als weiterer Anwendungsfall in die Anwendung *Studiengang durchführen* integriert. Die hinzugefügten Teile sind aufgrund der eingehaltenen Namenskonventionen leicht erkennbar.