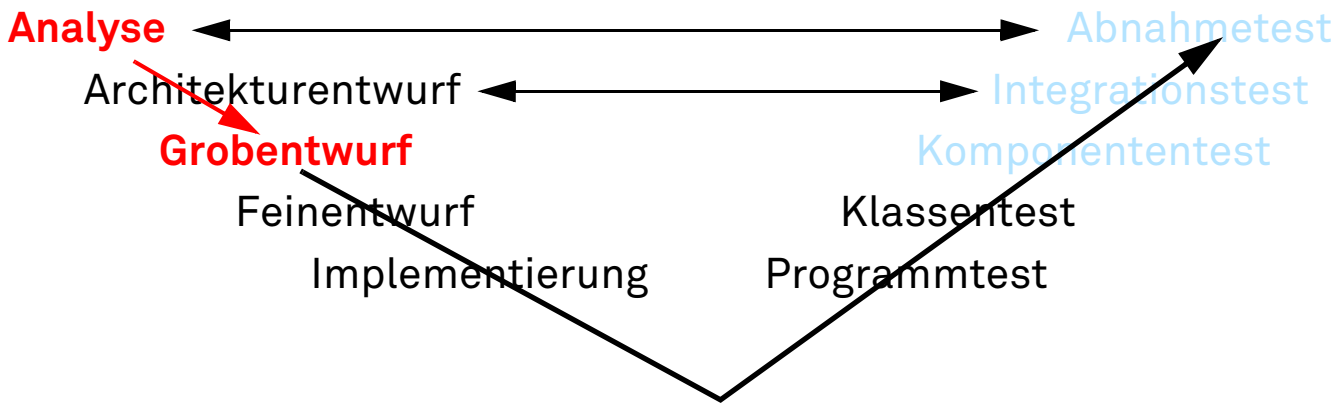
Folien zur Vorlesung
Softwaretechnik

Abschnitt 5.7: Übergang Analyse – Grobentwurf

Vorgehensmodell

V-Modell



- Noch offen ist eine genaue Beschreibung des Prozesses, der von den Ergebnissen der Analyse zum GrobEntwurf führt.

- In diesem Prozess müssen die Anforderungen aus Benutzer/Betreiber-Sicht in ein technisches Lösungskonzept überführt werden.

Ergebnisse der Analyse

Pflichtenheft:

- Anwendungsfalldiagramme mit
  - textuellen, tabellarischen Beschreibungen
  - Sequenzdiagrammen
  - Aktivitätsdiagrammen
- Problembereichsmodellierung mit
  - Klassendiagrammen
- Skizzen von Benutzungsschnittstellen
  - zugehörige Abläufe eventuell als Aktivitätsdiagramme

Alle Diagramme modellieren die Anforderungen an das zu erstellende System aus der Sicht von Benutzern und Betreibern.

Ergebnisse des Grobentwurfs

- Klassendiagramme mit
  – Attributen
  – Methoden
- Sequenzdiagramme (für Folgen von Aufrufen von Methoden)
- Aktivitätsdiagramme (für Abläufe in Methoden)

Alle Diagramme modellieren Eigenschaften des zu erstellenden Systems aus einer technischen Sicht.

Ergebnisse des Grobentwurfs (Fortsetzung)

- Klassendiagramme mit
  – Attributen
  – Methoden
- Sequenzdiagramme (für Aufruffolgen von Methoden)
- Aktivitätsdiagramme (für Abläufe in Methoden)

Alle Diagramme modellieren Eigenschaften des zu erstellenden Systems aus einer technischen Sicht.

⇒ **Erkenntnis:**
In Analyse und Entwurf
werden die gleichen Diagrammtypen
mit unterschiedlichen Zielsetzungen verwendet!

⇒ **Folgerung**
Die verschiedenen Diagrammtypen lassen sich universell einsetzen!

Problembereichsanalyse

Vorgehensweise bei der Erstellung des Problembereichsmodell in der Analyse:

- Beim Herausarbeiten von Anwendungsfällen werden Entitäten ermittelt:
  – Objekte, Attribute, Klassen, Beziehungen.
- Anschließend werden
  – gleiche Objekte zusammengefasst und auf einen Platzhalter reduziert,
  – Objekte zu Klassen verallgemeinert,
  – die Zuordnung von Attributen zu Klassen vorgenommen,
  – Klassen mit gleichen Aufgaben zusammengefasst,
  – ähnliche Klassen in Hierarchien angeordnet und
  – die Beziehungen zwischen den Klassen festlegt.
- Abschließend
  wird das Modell überprüft
  – und der Vorgang möglicherweise wiederholt.

Problembereichsanalyse (Fortsetzung)

Regeln für das Problembereichsmodell:

- Das Bereichsmodell beschreibt die Realität der Problemwelt.
- Das Bereichsmodell zeigt keine technische Lösungen.
- Jedes Element des Bereichsmodells bezieht sich auf ein Ding (eine Entität) aus der Problemwelt.
- Auf jede Klasse muss in mindestens einem Anwendungsfall Bezug genommen werden.
- Jede Klasse sollte mindestens ein Attribut oder mehr als eine Assoziation besitzen.
- Im Normalfall sollten bei der späteren Ausführung des Systems von jeder Klasse mehrere Objekte erzeugt werden.

Übergang: Analyse → Grobentwurf

Ausgangspunkt ist das Problembereichsmodell (Klassendiagramm) aus der Analyse.

- Klassen aus dem Problembereichsmodell sollen möglichst erhalten bleiben, um einen Bezug zwischen Analyse und Entwurf herzustellen und die Verständlichkeit zu erhöhen.
- Aus technischer Sicht notwendige Klassen werden hinzugefügt.
- Aus technischer Sicht notwendige Methoden werden hinzugefügt.
- Für die Ausführung notwendige Beziehungen werden hinzugefügt.
- Das so entstandene Modell wird anschließend verbessert.

Ziel des Vorgehens ist das Erstellen des technisch orientierten Klassendiagramms für den Entwurf.

Übergang: Analyse → Grobentwurf (Fortsetzung)

Konstrukion der Klassenstruktur des Entwurfs:

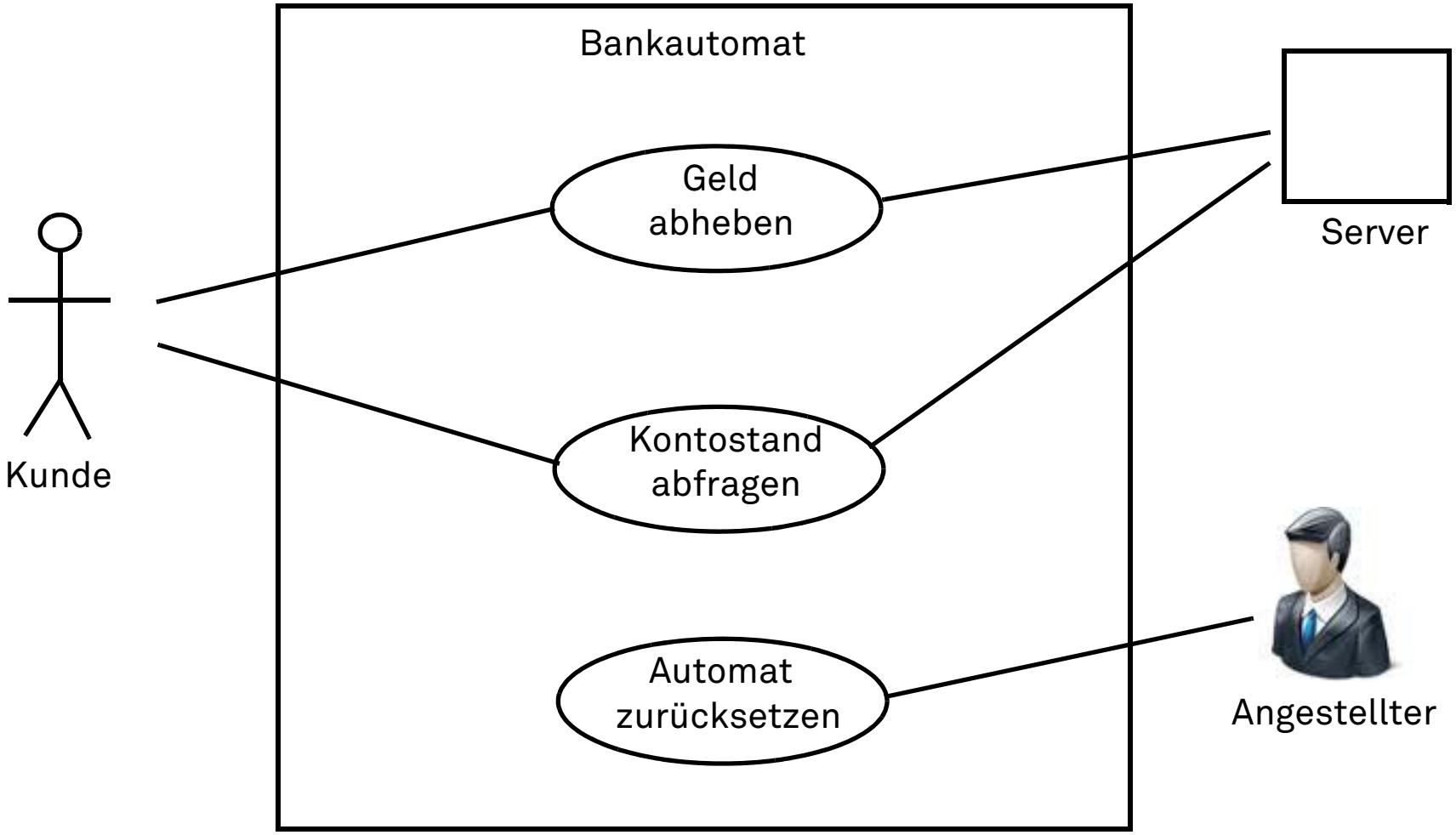
- Ausgangspunkt Problembereichsmodell:
  Die Klassen des Problembereichsmodells werden zu **Entitätsklassen** (und dienen der Datenhaltung).

- Ausgangspunkt iAnwendungsfallmodell:

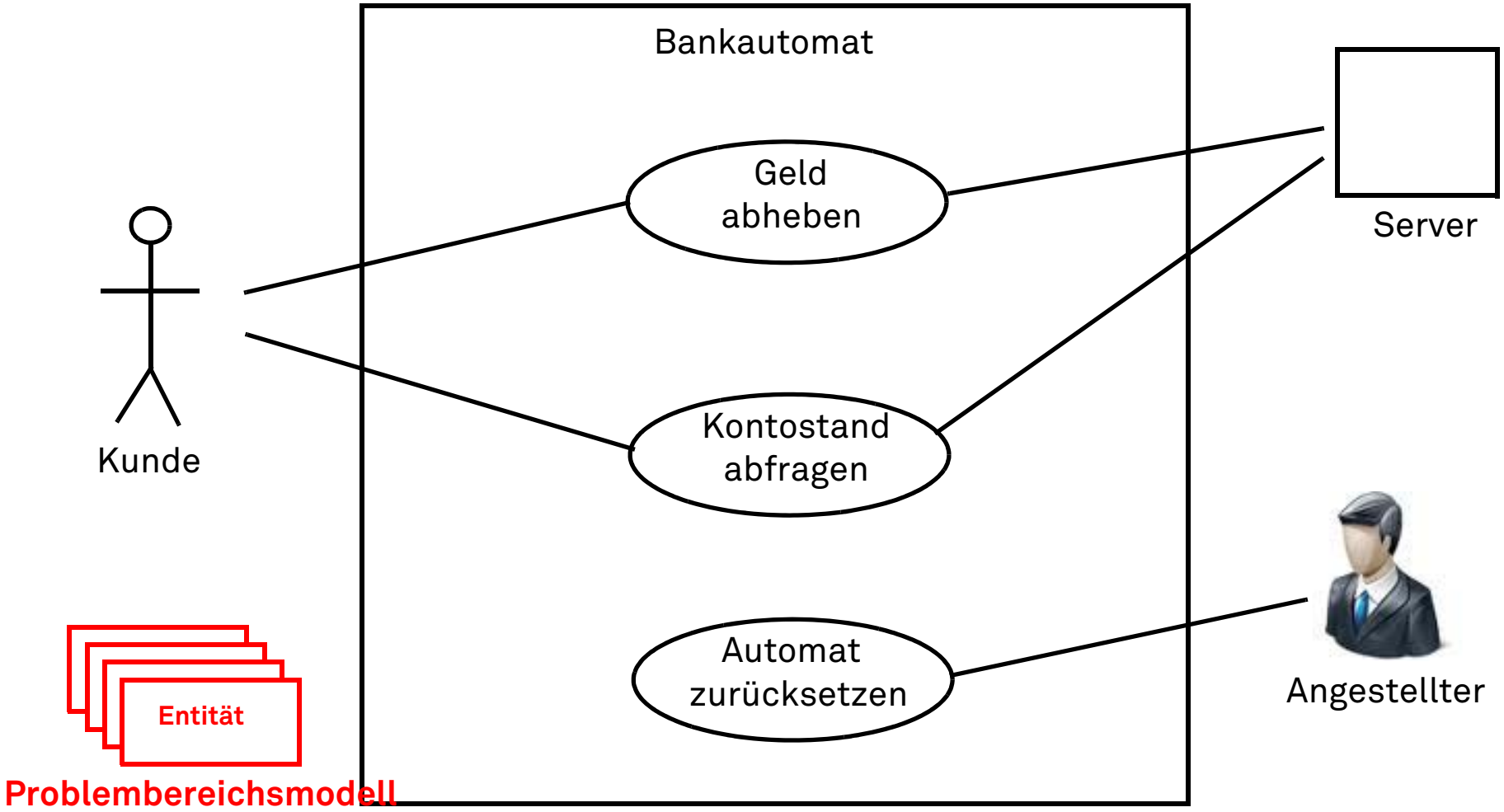
  – Zu jedem Anwendungsfall wird eine **Steuerungsklasse** geschaffen,
  die mindestens eine Methode zur Steuerung des Ablaufs des Anwendungsfalls enthält.

  – Zu jeder Assoziation mit einem Akteur wird eine **Schnittstellenklasse** geschaffen,
  die die Kommunikation mit dem Akteur abwickelt.

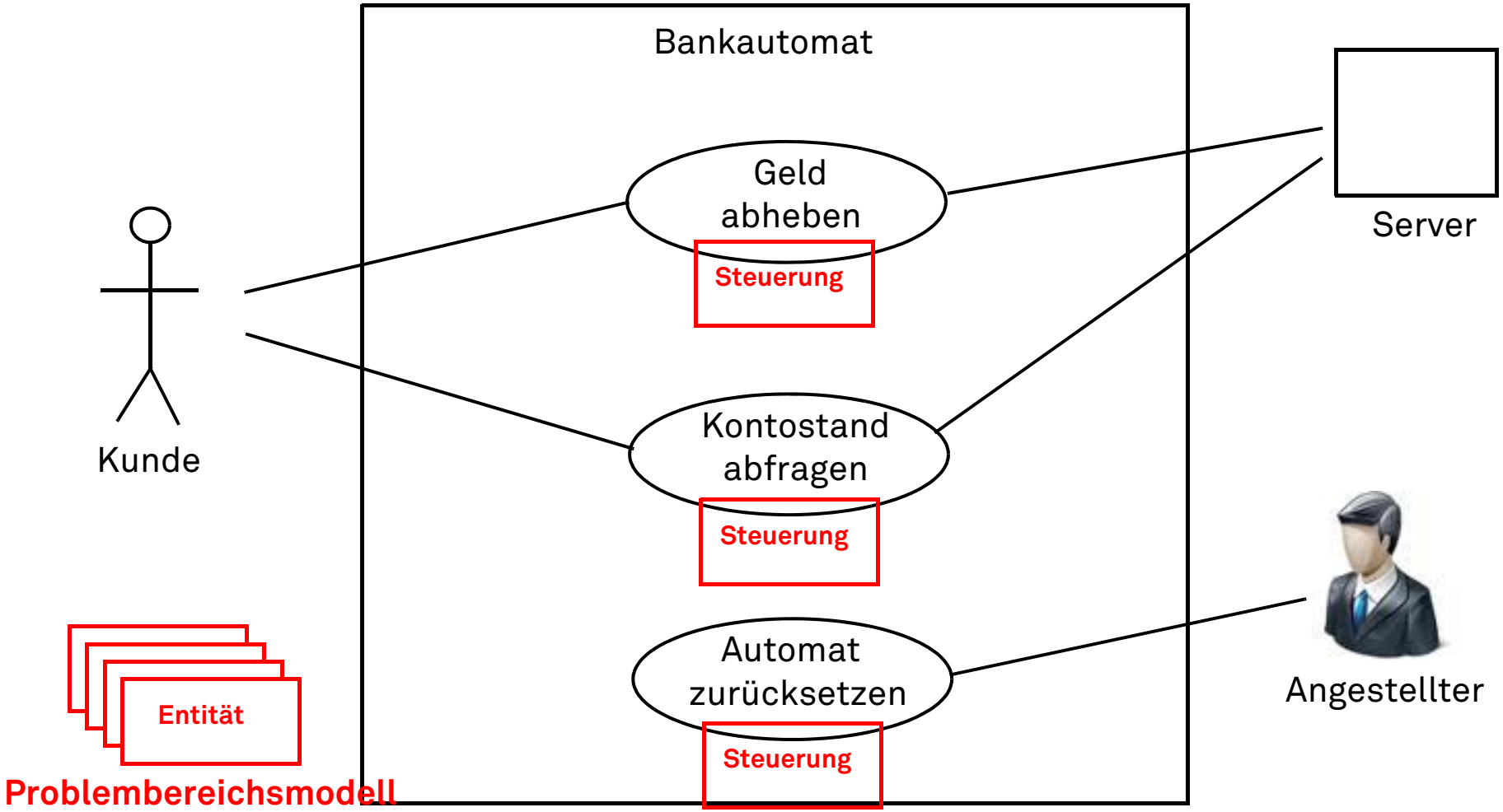
Übergang: Analyse → Grobentwurf (Beispiel)



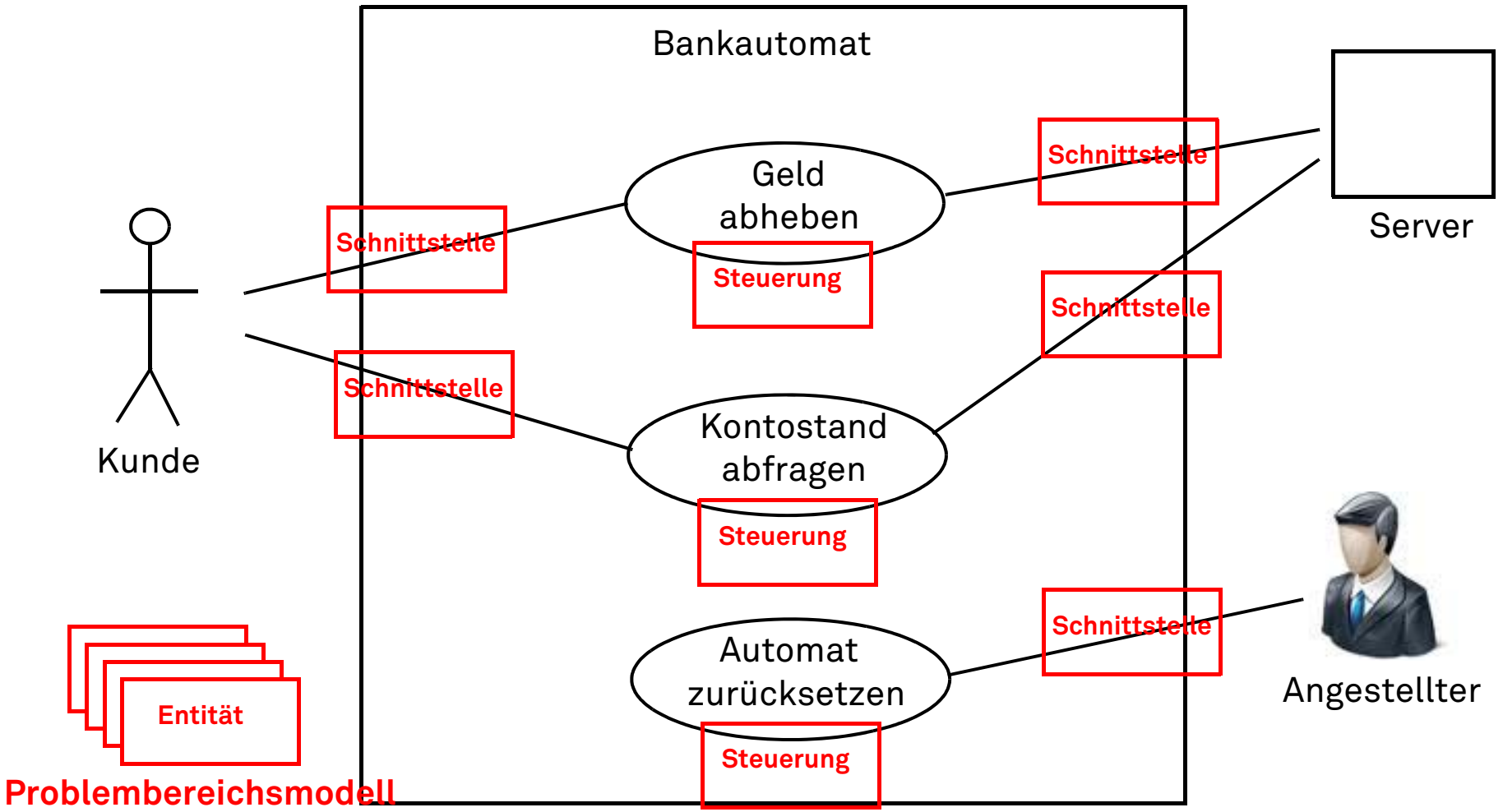
Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)



Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)



Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)



Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)

Konstrukion der Klassenstruktur des Entwurfs:

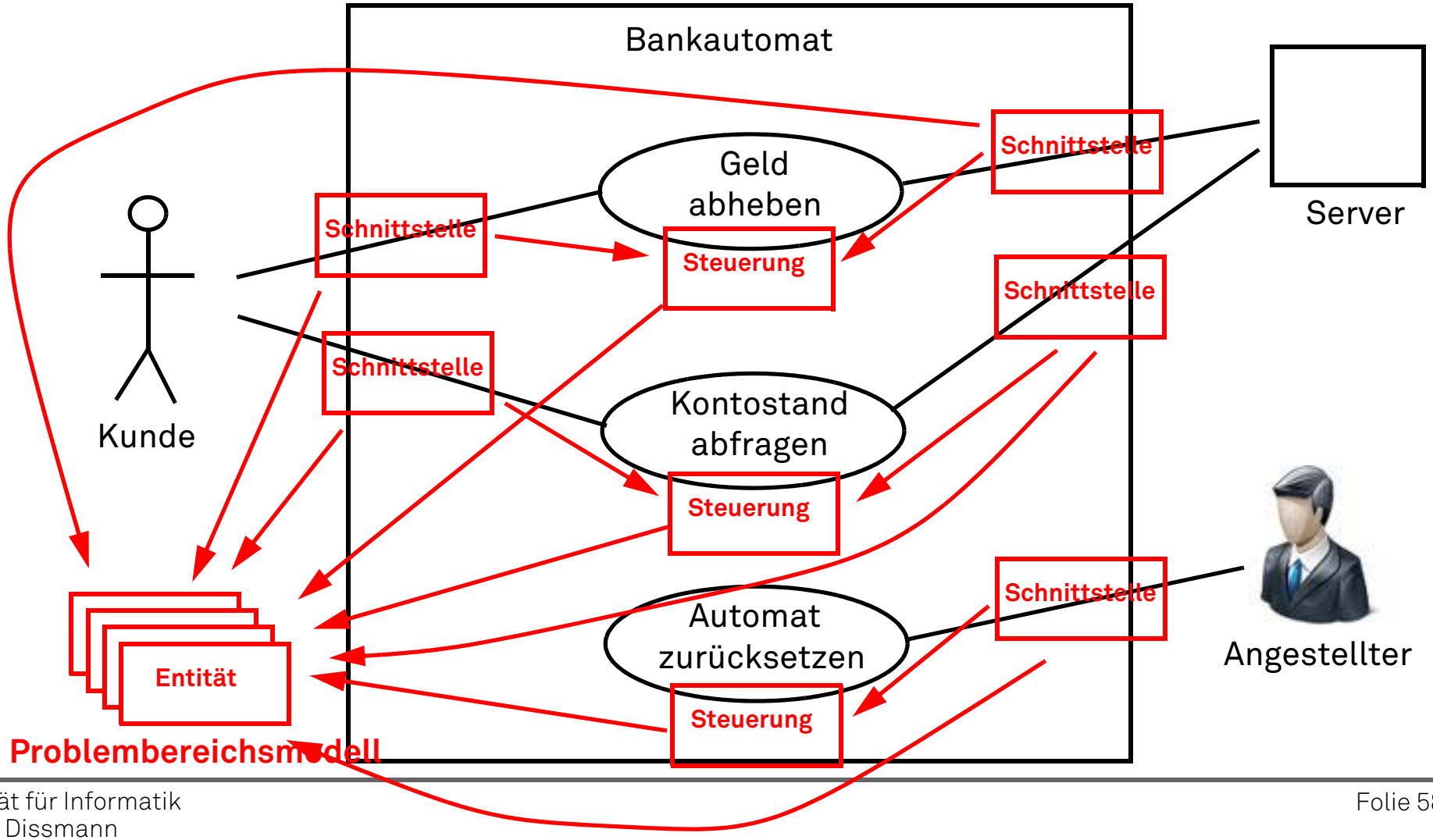
- **Entitätsklassen**
  enthalten in der Regel die Daten, die in dem System benötigt werden.

- **Steuerungsklassen**
  nutzen oder manipulieren die in den Entitätsklassen verwalteten Daten.

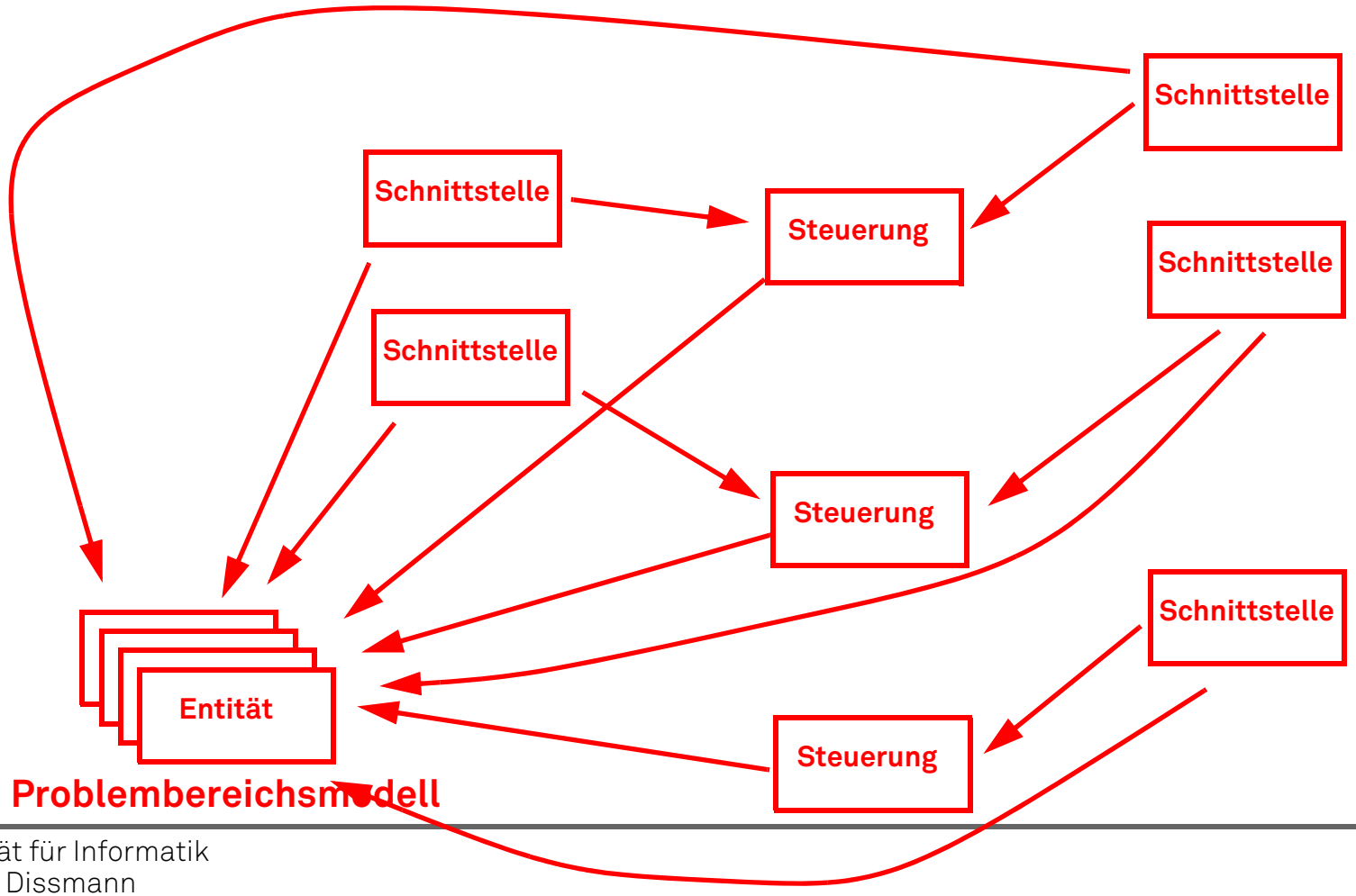
- **Schnittstellenklassen**
  – nutzen die Steuerungsklassen zum Anbieten von Funktionalität für einen Akteur,
  – nutzen Entitätsklassen zum Abrufen von Daten und zur Speicherung von Eingaben.

Dieser Aufgabenzuteilung liegt die Idee der MVC-Architektur zugrunde.

Übergang: Analyse → Grobentwurf (Beispiel) (Beispiel) (Fortsetzung)



Übergang: Analyse → Grobentwurf (Beispiel) (Beispiel) (Fortsetzung)



Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)

Ziele des Entwurfs:

- Es werden die in der Problemanalyse noch fehlende Beziehungen zwischen Klassen ergänzt, die für den Ablauf notwendig sind.
- Es werden fehlende technische Aspekte ergänzt.
- Die technische Umsetzbarkeit wird vorbereitet.
- Die organisatorische Umsetzbarkeit wird vorbereitet:
  Klassen werden nicht nur als technische Einheiten betrachtet,
  sondern dienen bei der Realisierung auch der Zuordnung von Aufgaben an Entwickler.
- Die durch bereits realisierte Programmteile gegebenen Schnittstellen müssen berücksichtigt werden.

- Gegebenenfalls muss auch die programmtechnische Umsetzung
  in einer bestimmten Programmiersprache muss vorbereitet werden.

Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)

Ergänzung technischer Aspekte

- Komplexe Attribute werden zu einer entsprechenden Klasse erweitert.
  Beispiele: Währungen, Adressen
- Multiplizitäten im Klassendiagramm
  werden durch geeignete Klassen realisiert.
  Beispiele: Listen von Objekten, Mengen von Objekten, Tabellen von Objekten
- Methoden für Konstruktion und zum Abbau von Strukturen werden ergänzt.
  Beispiele: Konstruktoren, Methoden zum Aufbau von Verbindungen zwischen Objekten
- Methoden werden angeglichen.
  Ähnliche Aufgaben werden durch Methoden mit ähnlichen Schnittstellen und ähnlichem Verhalten gelöst
- Die Kommunkationsabläufe werden angeglichen.
  Ähnliche Kommunikationsabläufe zwischen verschiedenen Klassen werden in ähnlicher Weise organisiert. Dabei werden auch Entwurfsmuster berücksichtigt.
- Algorithmen werden von Problemdetails abgetrennt.
  Algorithmen identifizieren, generische Klassen bilden.

Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)

Lebenszyklen von Objekten organisieren

- Das Erzeugen von Objekten wird organisiert.
  Rahmenbedingungen für das Erzeugen und Initialisieren festlegen.
- Das Vernichten von Objekten wird organisiert.
  Rahmenbedingungen und Abschlussaktionen festlegen
  (– hängt auch von der verwendeten Programmiersprache ab).
- Die Zuständigkeit für das Erzeugen von Objekten wird festgelegt.
  Gegebenenfalls spezielle Klassen ergänzen, die Objekte anderer Klassen erzeugen.
- Die Zuständigkeiten für die Vernichtung werden festgelegt.
  Spezielle Zustände definieren, entsprechende Klassen ergänzen.

Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)

Konzeption der Benutzungsschnittstelle

- Benutzungsschnittstellen sind ereignisgesteuert:
  - Die fensterbasierte Oberfläche ist objektorientiert strukturiert.
  - Der Benutzer löst ein Ereignis aus (durch Cursor-Position, Tastendruck, Mausklick ...).
  - Die Semantik des Ereignisses hängt vom betroffenen Oberflächen-Objekt ab.
  - Die aktivierten Oberflächen-Objekte rufen beim Auftreten eines Ereignisses Methoden von Objekten der Schnittstellen- oder Steuerungsklassen auf.
- Es bestehen aber vielfältige gegenseitige Abhängigkeiten:
  Steuerungsobjekt öffnet Fenster, Ereignisse bestimmen Ablauf im Steuerungsobjekt
  Der Fokus (d.h. die Position des Zeigers) bestimmt den Wechsel der Kontrolle,
  die Auswahl des aktiven Teils der Anwendung erfolgt durch den Benutzer.
- Sicherstellen eines einheitlichen Steuerungskonzepts.
  Alternativen:
  - Die Steuerungsklasse kontrolliert Aufbau und Änderungen von Fensterinhalten.
  - Die Schnittstellenklasse steuert sich selbst.

Übergang: Analyse → Grobentwurf (Beispiel) (Fortsetzung)

nach der – eventuell zyklisch wiederholten –
Ausführung der vorgestellten Schritte

- sind die Anforderungen aus dem Pflichtenheft in den Entwurf überführt worden,
- liegt ein realisierbares technisches Konzept vor, das die Anforderungen umsetzt.

Es müssen nun noch solche Verbesserungen am Entwurf vorgenommen werden, die die einige technischen Aspekte stärker berücksichtigen:

- Steuerung von Abläufen
- Datenhaltung
- Realisierbarkeit verbessern
- Performanz verbessern
- Implementierung vorbereiten

Ergänzungen des Grobentwurfs

Konzeption der Ablaufsteuerung

- Objektorientierte Systeme steuern sich selbst:
  – Methodenaufrufe bestimmen den Wechsel der Kontrolle zwischen den Objekten.
  – Die Steuerung des gesamten Systems ist auf viele Objekte verteilt.
- Eine einheitliche Strategie verbessert Verständlichkeit.
- Übergreifende Mechanismen (z.B. Fehlerbehandlung) müssen auch übergreifend geregelt werden.
- Eine klare Trennung von steuernden und von reagierenden Klassen erleichtert die Konstruktion der Abläufe und insbesondere auch das Testen.
- Nebenläufigkeit:
  – sich ausschließende Aktivitäten identifizieren
  – kritische Bereiche ermitteln
  – Synchronisationsmechanismen festgelegen
- Konzeption der persistenten (= dauerhaften) Datenhaltung:
  – Festlegen von Orten der Datenaufbewahrung (Datei/Datenbank)
  – Festlegen von Zeitpunkten für die Persistierung

Ergänzungen des Grobentwurfs (Fortsetzung)

Verbesserung der Realisierung von Klassen und deren Beziehungen

- Hinzunahme zusätzlicher Attribute/Beziehungen, um Abläufe zu vereinfachen oder beschleunigen (z.B. Pufferung).
- Hinzunahme redundanter Attribute, um die Ausführung zu beschleunigen.
- Gleiche Abläufe in verschiedenen Klassen werden
  mit den zugehörigen Attributen zusammengefasst und ausgelagert.
  Eventuell muss eine Oberklasse oder eine generische Klasse geschaffen werden.
- Teile einer Klasse werden ausgelagert, um deren Komplexität zu verringern.
- Hinzunahme zusätzlicher direkter Assoziationen zwischen Klassen,
  um mehrstufige Delegation zu vermeiden.
- implementierungsgerechte Umgestaltung von Klassen:
  – Hinzunahme zusätzlicher Klassen zur Umsetzung von *n:m*-Beziehungen
  – Hinzunahme zusätzlicher Klassen zur Umsetzung von Assoziationsklassen
  – Hinzunahme zusätzlicher Klassen zur Umsetzung von Kompositionen
  – Festlegen der Navigationsrichtung für ungerichtete Assoziationen

Verbesserung des Grobentwurfs (Fortsetzung)

Der Übergang vom Entwurf zur Implementierung ist unscharf:
UML-Editoren erzeugen z.B. Coderahmen und verschieben so Aspekte aus der Implementierung in den Entwurf.

Anpassung an die vorgesehene Programmiersprache

- Geeigneten Typen für Attribute festlegen.
- Nutzung von Bibliotheksklassen vorsehen und vorbereiten.
  Gegebenenfalls muss eine erneute Anpassung der Klassenstruktur erfolgen.
- Insbesondere kann Wiederverwendung auch durch die Konkretisierung von bereits vorhandenen generischen Klassen erfolgen.
- Umgestaltung von den in der ausgewählten Programmiersprache nicht realisierbaren Strukturen des Entwurfs.

Zusammenfassung

Das Ergebnis der vorgestellten Überarbeitungen/Ergänzungen ist ein

Entwurf,

- der die Anforderungen umsetzt und
- die technische Realisierung in einer bestimmten Programmiersprache ermöglicht.

Anmerkungen:

- Alle beschriebenen Überarbeitungen/Ergänzungen finden auf der konzeptionellen Ebene statt – also **bevor** Programmtext erstellt wird.
- Die Überarbeitung von bereits existierenden Programmen kann nach ähnlichen Regeln erfolgen und wird als Refactoring bezeichnet.

Hinweis

Die hier vorgestellten Entwicklungsschritte können nur schwer im Rahmen überschaubarer Übungsaufgaben durchgeführt werden.

Das Üben dieser Schritte erfolgt daher im Rahmen des

Softwarepraktikums.

Folien zur Vorlesung
Softwaretechnik

Abschnitt 5.8: Agile Vorgehensmodelle

Entwicklungsprozesse – kritische Betrachtung

Die Folge der wohldokumentierten Phasen
Analyse, Konzeption/Entwurf, Realisierung/Implementierung, Überprüfung/Test

wird in anderen Ingenieurwissenschaften in ähnlicher Form angewandt:
Architektur, Maschinenbau, Elektrotechnik, Anlagenbau, ...

aber:

Software ist immateriell

⇒
- leichte Änderbarkeit
- fast beliebige Änderbarkeit
- Modell/Prototyp kann zugleich Endprodukt sein
- Möglichkeiten einer visuellen Prüfung sind eingeschränkt

⇒ andere Entwicklungsprozesse sind möglich

Entwicklungsprozesse – kritische Betrachtung (Fortsetzung)

Probleme, die bei jeder Softwareentwicklung auftreten können:

- Verständnisschwierigkeiten mit dem Problembereich
  fallen erst spät bei der Konkretisierung (= Implementierung) auf.
- Der Aufwand für die Analyse und deren Dokumentation ist hoch.
- Spätere Änderungen müssen in allen Dokumenten vorgenommen werden.
- Die Phasen sind nicht gleichgewichtig:
  Implementierung und Testen benötigen viel Zeit.
- Empirische Untersuchung (CHAOS-Studie, Standish Group) behauptet:
  – nur ca. 16 % der Softwareentwicklungen sind erfolgreich,
  – 53 % sind nur teilweise erfolgreich,
  – 31 % werden ohne Erfolg abgebrochen.
- Gründe für Misserfolg von Softwareprojekten:
  – Zeitüberschreitung,
  – Budgetüberschreitung,
  – unzureichende Qualität (fehlerhafte Funktionalität),
  – falsch ermittelte Anforderungen,
  – zu spät erkannte zu hohe Komplexität.

Agile Vorgehensmodelle

Unter dem Begriff *Agile Vorgehensweise* werden zusammengefasst:

- größere Flexibilität
- weniger "Bürokratie"
- weniger Dokumentation
- stärkere Fokussierung auf die Programmiertätigkeit

- erste Veröffentlichung (Beck 1999):
  eXtreme Programming (XP)

- weiteres Beispiel:
  Scrum

Literatur: Hanser, Eckhart: Agile Prozesse: Von XP über Scrum bis MAP, S. 1-45, S. 61-77.
http://link.springer.com/chapter/10.1007/978-3-642-12313-9_3
http://link.springer.com/chapter/10.1007/978-3-642-12313-9_5

Vorgehensmodell eXtreme Programming (XP)

Prinzipien des eXtreme Programming:

- Alle beteiligten Personen betreiben ständige, persönliche Kommunikation.
- Die Entwickler handeln schnell, flexibel und selbstständig.
- Das Programm bildet das zentrale Dokument.
- Die Programmkonstruktion wird bewusst einfach gehalten.
- Das Programm wird ständig verändert.
- Das Programm besitzt jederzeit eine hohe Qualität.

Vorgehensmodell eXtreme Programming (XP) (Fortsetzung)

Praktiken des eXtreme Programming

- *on-site customer*:
  Der Kunde ist direkt in die Entwicklung eingebunden und Teil des Entwicklungsteams.
  – *Vorteil*: Fragen/Probleme können sofort geklärt werden.
  – *Nachteil*: Aufwand für den Kunden ist hoch, «entbehrliche» Mitarbeiter mit großer Kompetenz geben.
- *planning game*:
  Das Produkt wird durch Planspiele interaktiv vom ganzen Team(inkl Kunde) ermittelt.
- *metaphor*:
  – Fachbegriffe werden durch den Kunden in Form von *user stories* erstellt.
  – Es wird ein Glossar der benutzten Fachbegriffe erstellt.
- *simple design*:
  Es wird bewusst auf die Entwicklung allgemeingültiger Strukturen verzichtet.
  Es wird bewusst auf technisch anspruchsvolle Lösungen verzichtet.
  – *Vorteile*: Kosten werden reduziert, unmittelbare Qualität wird verbessert
  – *Nachteil*: eventuell späteres Überarbeiten notwendig

Vorgehensmodell eXtreme Programming (XP) (Fortsetzung)

Praktiken des eXtreme Programming (Fortsetzung)

- *pair programming*:
  Ein Entwickler (Driver) programmiert, ein zweiter (Observer) beobachtet die Arbeit.
  Die Rollen und die Partner werden häufig gewechselt.
  – *Vorteile*: Steigerung der Qualität durch Kontrolle, schwere Aufgaben werden bei der Lösung diskutiert, Entwickler lernen voneinander
  – *Nachteil*: doppelter Personalbedarf
- *collective ownership*:
  Alle Ergebnisse sind kollektives Eigentum des Teams. Die Aufgaben wechseln (sehr) häufig, jeder Entwickler bearbeitet alles.
  – *Vorteile*: weniger Probleme bei Inkompetenz, Krankheit, Urlaub usw.,
  – *Nachteil*: Spezialisten werden nicht adäquat eingesetzt, ständiges Einarbeiten in geänderten Code
- *coding standards*:
  – Alle Programme halten sich an strenge Codestandards.
  – Codestandards sind die Grundvoraussetzung, damit Paarprogrammierung und kollektives Eigentum angewandt werden können.

Vorgehensmodell eXtreme Programming (XP) (Fortsetzung)

Praktiken des eXtreme Programming (Fortsetzung)

- *collective ownership*:
  – Alle Ergebnisse sind kollektives Eigentum.
  – Das Team ist gemeinsam für allen Code verantwortlich.
  – Die Aufgaben wechseln (sehr) häufig, jeder Entwickler bearbeitet alles.
  – *Vorteile*:
    - Ausfälle durch Krankheit, Urlaub, Wechsel führen nicht zu Problemen.
    - Code wird durch wechselnde Bearbeiter normiert und verbessert.
    - Einzelne Entwickler werden bei Problemen durch das Team unterstützt.
  – *Nachteil*:
    - Spezialisten können nicht adäquat eingesetzt werden.
    - Es ist immer wieder Einarbeitung in geänderten Code notwendig.
- *coding standards*:
  Alle Programme halten sich an strenge Codierungsstandards als Grundvoraussetzung für *pair programming* und *collective ownership*.
- *short releases*:
  Eine Vorausplanung erfolgt nur in sehr kurzen Zeitabschnitten (Stunden/Tage).
  – *Vorteil*: flexible Anpassung an Kundenwünsche möglich.
  – *Nachteil*: kein systematisches Hinarbeiten auf ein Gesamtziel

Vorgehensmodell eXtreme Programming (XP) (Fortsetzung)

Praktiken des eXtreme Programming (Fortsetzung)

- *continous integration*:
  Alle Neuentwicklungen werden sofort integriert, es liegt immer ein aktuelles, ausführbares Programm vor.
  – *Vorteil*: Kunde sieht immer das ganze Produkt und kann es prüfen.
  – *Nachteil*: Entwicklung beginnt zwangsläufig mit der Benutzungsoberfläche
- *test first*:
  Testfälle werden vor der Implementierung einer Methode festgelegt und ersetzen eine detaillierte Spezifikation. Ständiges Testen ermöglicht die ständige Integration.
  – *Vorteil*: Qualität der Implementierung wird jederzeit sichergestellt.
- *refactoring*:
  Die Weiterentwicklung wird für eine Überarbeitungsphase unterbrochen, in der das gesamte Programm technisch restrukturiert und verbessert wird.
  – *Vorteil*: Überarbeitung erfolgt bedarfsorientiert und explizit.
  – *Nachteil*: zusätzlichen Aufwand, da die Einzelmaßnahmen aufwändig sein können.
- *40 hour week*:
  Annahmen: Überstunden zeigen unzureichender Planung, Entwickler werden mit langer Arbeitsdauer unproduktiver und liefern schlechtere Qualität

Vorgehensmodell eXtreme Programming (XP)

Gesamtbewertung:

- eXtreme Programming ist anders als *klassische* Softwareentwicklung.
- eXtreme Programming lässt sich schwer planen und überwachen.
- Eine Idee zur Kostenplanung ist:
  Es wird einfach solange inkrementell geplant und weiterentwickelt,
  bis der Kunde zufrieden oder das Budget aufgebraucht ist.
- eXtreme Programming ist ein dogmatischer Ansatz (= alle Praktiken sind unverzichtbar).
- eXtreme Programming erfordert kompetente Kunden.
- eXtreme Programming erfordert kompetente Entwickler.
- eXtreme Programming kann nur in Teams mit überschaubarer Größe eingesetzt werden.

aber:

- Testgetriebene Entwicklung *(test first)* wird inzwischen auch in der *klassischen* Entwicklung als Teilschritt am Übergang Entwurf/Implementierung eingesetzt.
- *refactoring* (mit entsprechenden Überarbeitungsphasen) wird inzwischen auch in der *klassischen* Entwicklung eingesetzt.

Vorgehensmodell Scrum

- Begriff:
  Scrum = Gedränge im Rugby (organisiertes Chaos)

- Eigenschaften:
  - Rahmen ist nicht so dogmatisch wie beim eXtreme Programming.
  - Schwerpunkt liegt auch auf Agilität.
  - Einsatz auch nur in kleinen Gruppen mit bis zu 10 Entwicklern.
    (Vollzeit im Projekt beschäftigt, Arbeitsplätze in einem Raum)

- Rollen
  - *pig* = Entwickler
  - *scrum master* = Moderator: überwacht die Einhaltung der Prozessregeln
  - *product owner*: vertritt den Auftraggeber im Team
  - *chicken* = Informationsquelle

Vorgehensmodell Scrum (Fortsetzung)

- Das Projekt wird in *sprints* organisiert:
  – *sprint* dauert bis zu 30 Tagen.
  – Umfang/Ziel wird vorher festgelegt und mit dem *product owner* abgestimmt.
  – Ergebnis wird vom *product owner* im *sprint review* bewertet.
  – Prozess wird in *retrospective* vom Team bewertet.

- *daily scrum*
  ist die tägliche Arbeitsplanung in der Gruppe.

Vorgehensmodell Scrum (Fortsetzung)

Artefakte

- Anforderungen
  sind (natürlichsprachlich formulierte) *stories* :
  Wer braucht was wozu/warum?
- *stories* werden im *backlog* (Arbeitsrückstand) gesammelt.
- Ausgewählte *stories* bilden den *sprint backlog*.
- *sprint burndown chart* zeigt den Arbeitsfortschritt.

- *product increment* ist das Ergebnis eines *sprint*.

Vorgehensmodell Scrum (Fortsetzung)

Gesamtbewertung:

- Scrum versucht,
  große Projekte in überschaubare Zwischenschritte *(sprints)* aufzuteilen.
- Scrum läßt sich dadurch etwas besser planen und überwachen als XP.
- Scrum verzichtet (wie XP) auf aufwändige Dokumentation.
- Scrum erfordert kompetente Kunden.
- Scrum kann ebenfalls nur in Teams mit überschaubarer Größe eingesetzt werden.
- Scrum kann etwas besser mit heterogenen Kompetenzen der Mitarbeiter umgehen.

Zusammenfassung agile Methoden

- Agile Methoden stellen
  die Programmier-Tätigkeit und den Programmier-Experten mehr in den Vordergrund.
- Agile Methoden setzen darauf, dass Software einfach änderbar ist.
- Agile Methoden setzen darauf, dass Software verständlich gestaltet werden kann.
- Agile Methoden erfordern viel Kommunikation im Team und mit dem Auftraggeber.
- Agile Methoden erfordern geeignete Werkzeugunterstützung für die Teamarbeit.

Folien zur Vorlesung
Softwaretechnik

Teil 6: Zusammenfassung

Abschluss Vorlesung Softwaretechnik

Inhalte

- UML
  - Klassendiagramm/Objektdiagramm
  - Aktivitätsdiagramm
  - Sequenzdiagramm
  - Anwendungsfalldiagramm
- Testen
  - funktionsorientierter Test
  - strukturorientierter Test
  - JUnit
- Vorgehensmodelle
  - Software-Architekturen
  - Entwurfsprozess
  - agile Softwareentwicklung
- Entwurfsmuster

Abschluss Vorlesung Softwaretechnik (Fortsetzung)

Die Inhalte represäntieren den **«State-of-the-Art»** der einfachen Softwaretechnik.

fehlende Inhalte

- vertiefte Darstellung aller Inhalte
- formale Techniken zur Spezifikation
- (formale) Techniken zur Spezifikation spezieller Eigenschaften (z.B. Sicherheit)
- Prozesse zur Entwicklung spezieller Software
- Projektplanung und -durchführung
- unterstützende Softwarewerkzeuge

Inhaltsverzeichnis