Strukturierung interaktiver Softwaresysteme in SE2

Petra-Becker-Pechau, Jörg Rathlev, Axel Schmolitzky, Christian Späh

Im Folgenden beschreiben wir kompakt einige *Entwurfsregeln*, die die Strukturierung *interaktiver Softwaresysteme* (SE2: Java-Systeme mit einer grafischen Benutzungsschnittstelle) erleichtern. Dabei gelten bewusst einige Einschränkungen, die diese Regeln relativ einfach halten: Es handelt sich um interaktive Desktop-Systeme, die nicht mit Verteilung oder Kooperation umgehen müssen (Einzelplatzsysteme) und die auch nicht auf einer Datenbank aufsetzen (u.a. keine Transaktionen).

Diese Entwurfsregeln erleichtern die Erstellung interaktiver Software, weil sie bei Entwurfsentscheidungen als Ratgeber dienen können. Sie bieten eine konkrete Anleitung, wie sich Softwaresysteme so strukturieren lassen, dass gängige Entwurfsprinzipien (wie beispielsweise der *Entwurf nach Zuständigkeiten*) beachtet werden.

Die Regeln benennen eine kleine Menge an *Elementtypen*, aus denen sich ein interaktives System zusammensetzt. Diese Elementtypen orientieren sich am *Werkzeug&Material-Ansatz* (kurz: WAM-Ansatz), der am Arbeitsbereich Softwaretechnik der Uni Hamburg entwickelt wurde. Dieser Ansatz ist ausgerichtet auf interaktive Systeme und eignet sich somit gut als Ausgangsbasis. Die vorliegenden Regeln setzen jedoch *nicht* die Kenntnis des WAM-Ansatzes voraus.

1 Übersicht

Die Elementtypen *Material*, *Fachwert*, *Service* und *Werkzeug* bilden die Basis eines interaktiven Systems. Abbildung 1 stellt ihren typischen Zusammenhang dar.

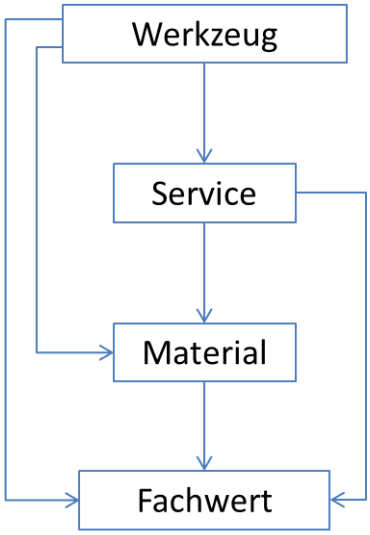

Abbildung 1: Übersicht der Elementtypen

Werkzeuge modellieren die interaktiven Teile eines Systems, sie haben eine *grafische Benutzungsschnittstelle* (engl.: graphical user interface, kurz GUI). Sie visualisieren Materialien und Fachwerte und greifen dafür auch auf Services zu. Sie erlauben der Benutzerin/dem Benutzer, Materialien zu bearbeiten, zu erstellen oder zu beseitigen.

Materialien realisieren veränderliche anwendungsfachliche *Gegenstände*, wie beispielsweise Konten oder Formulare. Fachwerte verkörpern anwendungsfachliche *Werte*, wie z.B. Kontonummern oder Geldbeträge. Services bieten fachliche Dienstleistungen an, die systemweit zur Verfügung stehen sollen; sie verwalten häufig Materialien und dienen in der Regel dafür, materialübergreifende Operationen anzubieten, wie beispielsweise die Ausleihe von Medien. Fachwerte, Materialien und Services sorgen für ihre interne Konsistenz, u.a. mit Hilfe des *Vertragsmodells*.

2 Die Elementtypen im Detail

Im Folgenden werden die Elementtypen näher beschrieben. Dabei gilt:

- In seiner programmtechnischen Umsetzung kann ein Elementtyp aus mehreren Klassen bestehen. Dies gilt insbesondere für Werkzeuge.
- Für jeden Elementtyp (*Werkzeug*, *Material*, *Service*, *Fachwert*) gibt es ein eigenes Java-Paket. Darunter kann es bei Bedarf noch eine weitere Aufteilung geben, z.B. im Werkzeug-Paket für verschiedene Werkzeuge.
- In jedem objektorientierten System können (unabhängig von den gewählten Elementtypen) *fachliche* und (*programmier-*)*technische* Klassen unterschieden werden.
    - Fachlich motivierte Klassen stehen in direktem Zusammenhang mit anwendungsfachlichen Begriffen, wie *Konto*, *Medium*, *Kontonummer*, *Verleihservice*; sie modellieren Konzepte des Anwendungsbereichs einer Software und sind üblicherweise mit den Anwendern diskutierbar.
    - Technisch motivierte Klassen hingegen sind zuständig für technische Aspekte wie grafische Oberflächen, Verteilung oder Persistenz. Der Anteil der technischen Klassen ist in realen Systemen meist deutlich größer als der des fachlichen Kerns.
    - Fachliche und technische Zuständigkeiten werden getrennt. Für jede Klasse in einem Java-System sollte die Frage gestellt (und beantwortet) werden: Ist dies eine fachliche oder eine technische Klasse?
- Die Elementtypen *Material*, *Fachwert* und *Service* werden durch fachliche Klassen modelliert, die somit auch eine fachlich motivierte Schnittstelle haben; *Werkzeuge* hingegen bieten ihre fachliche „Schnittstelle“ interaktiv gegenüber dem Benutzer an, die Schnittstellen der implementierenden (technisch motivierten) Klassen sind eher technisch geprägt.
- Einige Entwurfsregeln sind bewusst restriktiv, um gut in SE2 umsetzbar zu sein.

Elementtyp *Material*

Materialien realisieren anwendungsfachliche Gegenstände, die üblicherweise veränderlich sind, wie z.B. Konten oder Formulare. Sie werden von der Benutzerin/dem Benutzer mit Werkzeugen bearbeitet und sind Teil des Arbeitsergebnisses. Sie entstehen und vergehen im Laufe des Arbeitsprozesses.

Entwurfsregeln:

- Ein Material hat ausschließlich fachliche Aufgaben. Das heißt, dass ein Material keinerlei technische Aufgaben übernimmt, die beispielsweise die Persistenz betreffen oder das GUI-Framework (in SE2: *Swing*).
- Ein Material hat nur Beziehungen zu anderen Materialien und Fachwerten.
- *Vereinfachung für SE2*: Für jedes Material, für das es im Anwendungsbereich ein Original gibt, existiert im System genau ein passendes Material-Exemplar (ein Java-Objekt). Beispiel: für jede DVD im Regal einer Mediathek existiert genau ein Exemplar der DVD-Klasse im Mediathek-System.
- Ein Material informiert nicht aktiv über Änderungen seines Zustandes (es kann somit nicht im Beobachter-Muster beobachtbar sein).
- Eine Material-Klasse sichert die Konsistenz ihrer Exemplare über das Vertragsmodell.
- Zu jeder Material-Klasse gibt es eine Testklasse.

Elementtyp *Fachwert*

Fachwerte verkörpern anwendungsspezifische Werte, wie z.B. Kontonummern oder Geldbeträge. Ein Fachwert ist unveränderlich. Fachwerte werden durch *Werttypen* modelliert. Fachwerte werden konzeptuell nicht erzeugt oder vernichtet, die Wertemenge eines Werttyps ist konstant.

Entwurfsregeln:

- Fachwerte sind passiv – sie verändern keine fachlichen Objekte.
- Ein Fachwert kennt nur Fachwerte und andere Werttypen.
- Ein Fachwerttyp wird durch eine Klasse modelliert; eine solche Klasse nennen wir *Wertklasse*, ihre Exemplare *Wertobjekte*.
- Die Konstruktoren einer Wertklasse können öffentlich sein; die Erzeugung von Wertobjekten kann aber auch durch Fabrikmethoden gekapselt werden.
- Es können für denselben fachlichen Wert durchaus mehrere Wertobjekte existieren; dann muss die Wertklasse in Java jedoch die Operationen *equals* und *hashCode* redefinieren. Klienten dürfen dann Wertobjekte nur mit *equals* vergleichen, nie auf Referenzgleichheit.
- Eine Wertklasse wird in Java als final deklariert.
- Die Felder einer Wertklasse werden in Java als final deklariert.
- Eine Wertklasse sichert die Konsistenz ihrer Exemplare über das Vertragsmodell.
- Zu jeder Wertklasse gibt es eine Testklasse.

Den bekanntesten Werttyp in Java (neben den primitiven Werttypen) definiert die Wertklasse *String*.

Elementtyp *Service*

Services bieten fachliche Dienstleistungen an, die systemweit zur Verfügung stehen sollen. Sie dienen in der Regel dafür, materialübergreifende Operationen anzubieten. Im Gegensatz zu Materialien gibt es von jedem Service nur ein Exemplar im System (Beispiel: Kundenstamm).

Entwurfsregeln:

- Services arbeiten häufig auf Materialien und/oder mit anderen Services.
- Services können Materialien verwalten. Beispielsweise ist der Service „Medienbestand“ in der Mediathek dafür zuständig, die Medien zu verwalten. Der Service hält die Information, welche Medien im System existieren. Er erlaubt es, Medien hinzuzufügen oder zu löschen.
- Services werden an zentraler Stelle erzeugt und verdrahtet, beispielsweise in einer StartUp-Klasse, und den Werkzeugen bei Bedarf als Konstruktorparameter übergeben.
- *Vereinfachung für SE2*: Services liefern Referenzen auf die Original-Materialien, nicht auf Kopien.
- Services können aktiv über Änderungen ihres Zustandes (bzw. der verwalteten Materialien) informieren, sie können *beobachtbar* sein (für Werkzeuge oder andere Services, *für Beobachter*). Wird ein Material geändert, das ein Service verwaltet, dann muss der Service über eine Update-Operation verfügen, mit der die Änderung allen Beobachtern bekannt gemacht wird.
- Ein Service sichert die Konsistenz seiner Leistungen über das Vertragsmodell.
- Zu jedem Service gibt es eine Testklasse.
- Services muss es nicht in jedem interaktiven System geben.
- Services können Persistenzaufgaben kapseln (also zuständig für die dauerhafte Speicherung von Materialien sein).

Elementtyp *Werkzeug*

Werkzeuge dienen zur Benutzerinteraktion. Mit Werkzeugen können Benutzerinnen und Benutzer über eine grafische Schnittstelle (GUI) interaktiv Materialien ansehen und bearbeiten. Ein Werkzeug übernimmt genau eine fachliche Aufgabe, die in einem kurzen Satz gut beschreibbar sein sollte.

In SE2 zerlegen wir Werkzeuge immer in eine *Werkzeug-Klasse* und eine *UI-Klasse (UI = User Interface)*.

Werkzeug-Klasse

Die Werkzeug-Klasse verarbeitet die Benutzereingaben und stößt die passenden fachlichen Operationen (an Materialien, Fachwerten und Services) an. Sie vermittelt also zwischen der grafischen Schnittstelle der UI-Klasse und den fachlichen Klassen. Sie setzt an ihrer UI-Klasse die anzuzeigenden Materialien und Fachwerte und registriert sich als Listener an den Elementen der grafischen Oberfläche (*Widgets*), um von Benutzereingaben zu erfahren. Sie übernimmt selber keine fachlichen Aufgaben, sondern delegiert diese an Materialien und Services. Sie reagiert auf Benutzungsinteraktionen, indem sie auf Materialien und/oder Services passende fachliche Operationen ausführt und/oder die grafische Oberfläche aktualisiert.

Entwurfsregeln:

- Die Werkzeug-Klasse bietet keine fachlichen Operationen an (ihre „fachliche Schnittstelle“, gegenüber dem Benutzer, ist die GUI).
- Die Werkzeug-Klasse erhält ihr Material als Konstruktorparameter, über Setter (wenn das Material austauschbar sein soll) oder holt sich dieses über Services.
- Der Werkzeug-Klasse werden benötigte Services als Konstruktorparameter übergeben.
- Die Werkzeug-Klasse erzeugt ein Exemplar ihrer UI-Klasse im eigenen Konstruktor.
- Die Werkzeug-Klasse erzeugt Listener für die UI-Widgets, die passende Aktionen ausführen.
- Die Werkzeug-Klasse holt sich die UI-Widgets aus der UI-Klasse (über Getter), um die eigenen Listener daran zu registrieren und/oder die Anzeige zu aktualisieren.
- Die Werkzeug-Klasse gibt die anzuzeigenden Materialien oder Fachwerte in die GUI. Dies geschieht mit Swing folgendermaßen:
  - Die Werkzeug-Klasse holt sich von der UI-Klasse über einen Getter ein Swing-Widget oder (falls vorhanden) ein selbst geschriebenes Swing-Model.
  - An diesem Widget/dem Swing-Model werden die anzuzeigenden Informationen gesetzt.
  - Sofern nötig, werden anzuzeigende Materialien/Fachwerte mit einem Formatierer adaptiert. (Details siehe unten, Abschnitt über Formatierer)
- Die Werkzeug-Klasse hält den Werkzeug-Zustand (sofern es einen gibt, der unabhängig vom Material-Zustand wichtig ist).
- Die Werkzeug-Klasse braucht keine Testklasse.

UI-Klasse

Die UI-Klasse eines Werkzeugs hat die Aufgaben, die Widgets zu erzeugen, zu layouten und zu verwalten.

Entwurfsregeln:

- Eine UI-Klasse stellt die für ihre Werkzeug-Klasse relevanten UI-Elemente über Getter an ihrer Schnittstelle zur Verfügung. Dazu gehören auch eigene Swing-Models.
- Eine UI-Klasse erbt *nicht* von UI-Framework-Klassen wie JFrame oder JPanel, um die eigene Schnittstelle schmal zu halten. Sie definiert als oberste UI-Komponente üblicherweise einen JFrame.
- Eine UI-Klasse sollte paketintern (default-Sichtbarkeit im Werkzeug-Paket) deklariert werden.
- Eine UI-Klasse verwendet nur Importe aus dem UI-Framework.

- Eine UI-Klasse hat keine statischen Abhängigkeiten zu anderen Elementtypen (außer zu eigenen Swing-Models).
- Eine UI-Klasse kann auch selbst geschriebene Widgets verwenden. Selbst geschriebene Widgets dürfen (im Gegensatz zur UI-Klasse) Abhängigkeiten zu Fachwerten haben, aber nicht zu anderen Elementtypen.
- Zu einer UI-Klasse gibt es keine Testklasse.

Zusammengesetzte Werkzeuge

Werkzeuge mit umfangreicher Funktionalität können in kleinere Werkzeuge zerlegt werden, deren UIs ineinander geschachtelt sein können. Sie werden damit zu *zusammengesetzten Werkzeugen*.

Entwurfsregeln:

- Werkzeuge werden entweder als *Top-Level-Werkzeug* oder als *Subwerkzeug* konstruiert. Subwerkzeuge können wiederum Subwerkzeuge enthalten, so dass eine Baumstruktur entsteht.
- Der direkte Vorgänger eines Subwerkzeugs in dieser Baumstruktur wird als sein *Kontextwerkzeug* bezeichnet.
- Das Top-Level-Werkzeug wird von der Startup-Klasse erzeugt.
- Kontextwerkzeug-Klassen erzeugen ihre Subwerkzeug-Klassen und versorgen diese mit Materialien und Services.
- Kontextwerkzeug-Klassen können ihre Subwerkzeug-Klassen beobachten und so auf fachliche Änderungen reagieren.
- Ein Subwerkzeug sollte genau eine fachlich nachvollziehbare Aufgabe übernehmen und deshalb sein Kontextwerkzeug maximal über einen Typ von Änderungen informieren.
- Die UI-Klasse eines Top-Level-Werkzeugs definiert als oberste UI-Komponente einen JFrame, Subwerkzeuge haben als oberste UI-Komponente in der Regel ein JPanel, sie können aber ebenso in einem eigenen JFrame angezeigt werden.
- Bei der Konstruktion mit Java Swing können Subwerkzeuge ihr JPanel für die Kontextwerkzeug-Klasse über eine sondierende Operation zur Verfügung stellen. Die Kontextwerkzeug-Klasse baut in diesem Fall ihre GUI aus den Panels ihrer Subwerkzeuge zusammen, indem es diese an seine UI-Klasse im Konstruktor weitergibt.

3 Zusätzliche Klassen zur Werkzeugkonstruktion

Formatierer

Formatierer dienen als *Adapter* für Materialien und/oder Fachwerte gegenüber Swing. Wir verwenden zwei Arten von Formatierern:

1. *toString*-Formatierer: Ohne Formatierer würde Swing direkt die Operation *toString* am darzustellenden Exemplar aufrufen. Dies ist aus folgenden Gründen nicht sinnvoll:
   - Das darzustellende Exemplar wäre auf eine einzige String-Darstellung festgelegt; Unterschiedliche Darstellungen an verschiedenen UI-Stellen und Mehrsprachfähigkeit wären nicht möglich.
   - In Java ist es üblich, die Operation *toString* lediglich für das Debugging zu verwenden.
2. *Model*-Formatierer: Können verwendet werden, wenn eigene Swing-Models zum Einsatz kommen.

Entwurfsregeln:

- Ein Formatier kann ein oder mehrere Materialien und/oder Fachwerte adaptieren. Im Folgenden sprechen wir kurz von *darzustellenden Exemplaren.*
- Formatierer erhalten ihre darzustellenden Exemplare als Konstruktorparameter.
- Formatierer rufen nicht *toString* an den darzustellenden Exemplaren auf, sondern bauen entweder den Anzeige-String selber zusammen (Beispiel: "Kino: " + Saal.gibName()) oder rufen Operationen der Exemplare auf, die eine geeignete String-Darstellung liefern.
- *toString*-Formatierer:
  - Ein Formatierer redefiniert die vom Typ *Object* geerbte Operation *toString*, indem er in seiner *toString*-Methode geeignet auf die darzustellenden Exemplare zugreift.
- *Model*-Formatierer:
  - Sie bieten für jeden darzustellenden Wert eine Operation an, die eine geeignete Darstellung, wie einen Text oder eine Grafik, liefert (beispielsweise eine Operation für jede Tabellenspalte). Diese Operation(en) werden von dem eigenen Swing-Model aufgerufen.
- Ein Formatierer wird nicht benötigt, wenn das Swing-Widget kein eigenes Model hat (Beispiel: JLabel). Dann fragt stattdessen das Werkzeug den Anzeige-String ab (etwa mit der Operation „gibFormatiertenSting“ oder „Uhrzeit.gibDoppelpunktDarstellung“) und übergibt diesen String an das Widget.
- Formatierer gehören zur Benutzungsschnittstelle, gemeinsam mit den Werkzeugen. Sie werden als paketinterne Klasse modelliert (default-Sichtbarkeit).
- Formatierer erfordern keine eigene Testklasse.

Eigene Swing-Models

Eigene Swing-Models sind technische Klassen, die unter Umständen bei der Werkzeug-Konstruktion mit Swing benötigt werden (beispielsweise als Model für eine *JList*).

Entwurfsregeln:

- Ein eigenes Swing-Model darf auf einem Material, aber nicht auf Services arbeiten.
- Eigene Swing-Models können direkt auf die fachlich sinnvollen Operationen der darzustellenden Exemplare zugreifen, genau wie Formatierer (s.o.); für sie wird somit kein Formatierer benötigt.
- Wenn gewünscht, kann für ein eigenes Swing-Model zusätzlich ein Formatierer erstellt werden. Das Model greift dann nur auf den Formatierer zu und dieser wiederum auf die darzustellenden Exemplare (s.o.).
- Soll das eigene Swing-Model eine Grafik darstellen, so wählt es diese Grafik selber, anhand des darzustellenden fachlichen Zustands. Diesen fragt es an Materialien / Fachwerten / Services / Formatierer ab.
- Jedes eigene Swing-Model hat eine Testklasse.

4 Begründungen und Erläuterungen

In diesem Abschnitt werden für interessierte Personen einige der Entwurfsentscheidungen begründet und erläutert. Für die Konstruktion eines konkreten Systems ist das Lesen dieses Abschnitts nicht notwendig.

Gründe dafür, dass Materialien keine Events verschicken:

- Es passt nicht zur Metapher vom passiven „Material“.
- Aufwändig zu programmieren: Werkzeuge, die auf Material-Mengen arbeiten, müssten sich bei jedem einzelnen Material als Listener anmelden.
- Ist sehr kompliziert, wenn man mit Kopien arbeitet, da Listener an verschiedenen Kopien hängen und nicht benachrichtigt werden, wenn eine Kopie sich ändert.
- Über mehrere Prozessräume noch problematischer.

Gründe dafür, dass wir keine FK-IAK-Trennung vornehmen:

- Vorteile von FK-IAK-Trennung sind seit der Einführung von Services in WAM nur noch gering, da die fachlichen Aufgaben der FK nun meist von Services übernommen werden.
- Die FK wird so sehr dünn, delegiert nur an Services.
- Einzige Aufgabe der FK, die bleibt: Werkzeugzustand speichern. Kommt nicht oft vor.
- FK-IAK-Trennung wird somit aus unserer Sicht nur in Spezialfällen benötigt, wäre hier eine Lösung ohne Problem.
- Ist deutlich komplexer als Mono-Werkzeuge durch das Beobachter-Muster zwischen FK und IAK.
- Hinweis für WAM-Kenner: wir bauen somit sog. Mono-Werkzeuge

Heuristiken für den Werkzeugschnitt:

- Es gibt nicht die eine Lösung.
- Zu große Werkzeuge sollten in mehrere Subwerkzeuge zerlegt werden (Ziel: Verständlichkeit!).

  Die *potentielle* Wiederverwendbarkeit von Subwerkzeugen sollte bei der Zerlegung keine Rolle spielen (wäre Upfront-Design).

Grund dafür, dass UI-Klassen keine Testklasse brauchen:

- GUI ist nicht einfach über Unit-Tests testbar

Grund dafür, dass Werkzeug-Klassen keine Testklasse brauchen:

- UI ist nicht über Unit-Tests testbar und Werkzeugklasse lässt sich nicht ohne UI ausführen

Hinweis zu Formatierern:

- Für JFace-Kenner: Mit JFace wären Formatierer nicht nötig, da deren Funktion dort von LabelProvidern übernommen wird.

Gründe dafür, dass Services in SE2 die Original-Materialien herausgeben und keine Kopien.

- Mit Kopien zu arbeiten ist komplizierter, weil:
  - Das Arbeiten mit Kopien erfordert IDs, eindeutige Ids zu erstellen ist aufwändig
  - Equals wird schwieriger (fachlich)
  - Material-Änderungen führen dazu, dass Kopie und Original wieder zusammengeführt werden müssen