

## 7 Databinding

»Beziehungen schaden nur dem, der sie nicht hat.«

—Volksmund

In Enterprise-Anwendungen geht es überwiegend darum, die aus dem Backend geladenen Daten, in anderen Worten: das *Model*, in der Oberfläche anzuzeigen und die vom Benutzer dort eingegebenen Daten wieder zurück in das Backend zu befördern. In den überwiegenden Fällen stellt die Oberfläche für jedes Attribut eines zu bearbeitenden Geschäftsobjekts ein entsprechendes Eingabefeld zur Verfügung.

Die Eclipse-Plattform bietet für diese Beziehungen von Objektattributen und Eingabeelementen das Eclipse Databinding als Framework an. Dieses Kapitel widmet sich dem Thema Databinding, da es einen wichtigen Teil des UI abbildet. Im ersten Abschnitt *Idee und Grundgedanke* gehe ich auf das grundlegende Konzept und das Ziel der Verwendung eines Databindings ein. Im Anschluss daran wird die Synchronisation der verbundenen Elemente detailliert beleuchtet, die neben der Transposition der Werte auch die Konvertierung und Validierung der Werte beinhalten kann. Zum Abschluss dieses Kapitels zeige ich anhand von Codebeispielen, wie das Eclipse Databinding für Eclipse-RCP-Anwendungen verwendet werden kann.

### 7.1 Idee und Grundgedanke

In vielen Fällen ist es eher trivial, die Widgets mit den Werten der Attribute des darzustellenden Modells zu füllen. Man nimmt den String-Wert des Attributs und setzt ihn in das Text Widget. Beim Speichern der Daten liest man die modifizierten Werte aus dem Text-Widget und schreibt sie zurück in das Modell. Es erscheint auf den ersten Blick etwas übertrieben, für diesen Zweck ein eigenes Framework zu entwickeln. Zudem darf die Frage gestellt werden, ob das bekannte Entwurfsmuster MVC (Model View Controller), das man häufig als Grundlage zur Darstellung der Daten heranzieht, die Aufgabe nicht bereits wunderbar löst.

### 7.1.1 Komplexe Verbindungen

*Datenkonvertierung*

Bezüglich der Frage, ob überhaupt ein eigenes Framework vonnöten ist, ist in Betracht zu ziehen, dass wesentlich komplexere Verbindungen als die gerade beschriebenen existieren. Beispielsweise müssen numerische Werte, Attribute mit Datumsinformationen oder andere individuelle Objekte mit einem Textfeld oder einem anderen Widget dargestellt werden. Natürlich kann man diese notwendigen Funktionen auch manuell codieren, die Databinding-Idee bietet für diese Anforderung allerdings eine konzeptionelle Lösung. Zudem nimmt es dem Entwickler hier eine Menge Arbeit ab und sorgt für ein konsistentes *Look and Feel* des Codes.

### 7.1.2 MVC – Model-View-Controller

Betrachten wir nun das erwähnte MVC-Entwurfsmuster etwas genauer. MVC definiert zunächst drei Komponenten, die nach bestimmten Regeln miteinander kommunizieren.

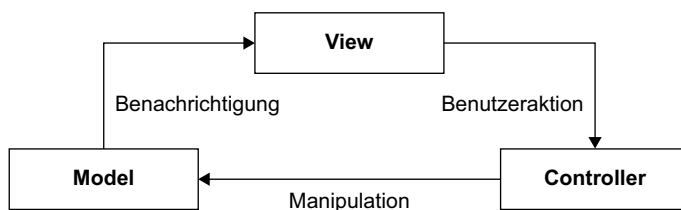
*MVC-Komponenten*

**Model** Das Model beinhaltet die Daten als die zur Zeit gültigen Werte. Sobald die Daten sich verändern, wird der View benachrichtigt, damit die aktualisierten Daten dargestellt werden können.

**View** Der View ist für die Darstellung des Models bzw. dessen Daten zuständig. Sobald er vom Model benachrichtigt wird, greift er auf das Model, holt die aktuellen Daten und stellt diese dar. Interaktionen des Benutzers, die im View geschehen, werden von diesem an den Controller weitergegeben, der entsprechend reagiert und ggf. das Model anpasst.

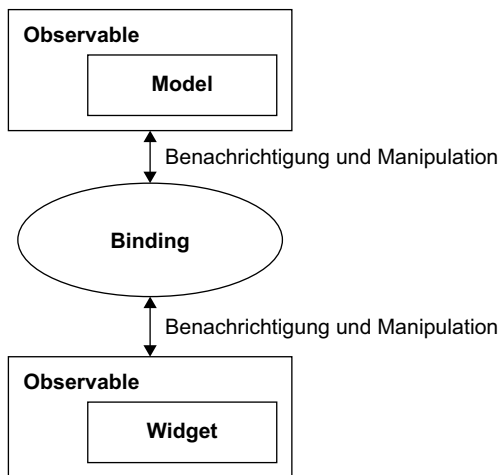
**Controller** Der Controller ist das steuernde Element. Er enthält Regeln und Funktionen, die über den View getriggert werden und Manipulationen auf dem Model nach sich ziehen.

**Abb. 7-1**  
MVC definiert die Interaktions- und Kommunikationswege der einzelnen Komponenten.



MVC geht also davon aus, dass der View respektive die Widgets die Daten darstellen können bzw. wissen, wie sie dies bewerkstelligen. So

wird impliziert, dass das Widget weiß, wie mit dem vom Model bereitgestellten Daten umzugehen ist. Wie wir aber schon festgestellt haben, ist das nicht immer ganz trivial und stellt damit den Stolperstein dar. Da durchaus unterschiedliche Datentypen im Model und View vorkommen können, ist für die Darstellung zunächst Logik notwendig, die ein passendes Format erzeugt. Dieser Code ist in vielen Fällen nicht wirklich auf technische, also dem Widget zugehörige, Aspekte zu beschränken, sondern steht oft auch in einem sehr fachlichen Kontext. Mitunter repräsentieren Widgets nicht nur *primitive* Daten, sondern eben auch komplexe fachliche Objekte. Somit ist das Databinding kein Controller im eigentlichen Sinne, sondern steht genauer gesagt als Mittler zwischen dem View und dem Model und sorgt für deren Synchronisation.



**Abb. 7-2**

Das Schema des Databindings folgt im Ansatz dem MVC-Entwurfsmuster, allerdings stellt es keinen echten Controller dar.

### 7.1.3 Synchronisation

Das Databinding definiert dabei zunächst lediglich die Verbindung einzelner Oberflächenelemente mit einzelnen Attributen eines Objekts. Diese Verbindung wird dann bei bestimmten Ereignissen, wie beispielsweise Veränderung des Models oder der Widgets, synchronisiert. Dabei werden grundsätzlich die zwei Richtungen der Synchronisation unterschieden:

**Model2Widget** Die Werte der Attribute des verbundenen Objektes werden in die Widgets kopiert, bisherige Werte werden überschrieben.

**Widget2Model** Der Inhalt des Widgets wird jeweils als Wert des Attributes des verbundenen Objektes gesetzt, bisherige Werte werden überschrieben.

*Richtungen der Synchronisation*

Die Synchronisation der vorhandenen Verbindungen erfolgt dann in einzelnen Phasen.

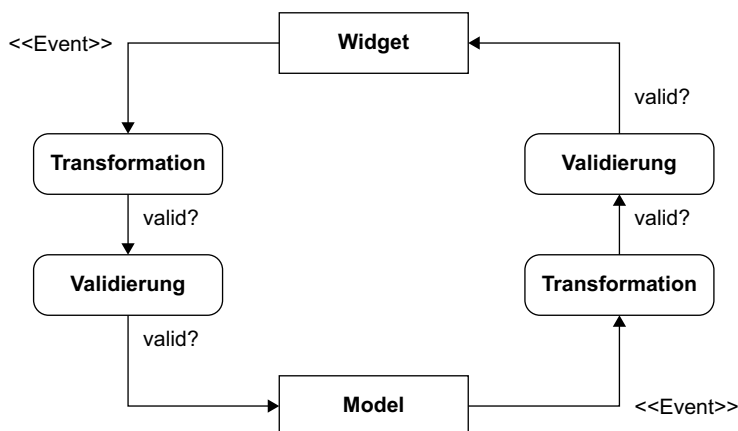
## 7.2 Konzept der Synchronisation

Databinding bedeutet also Synchronisation von Modell und Widgets und geschieht in Phasen. Die einzelnen Phasen hängen dabei eng mit zwei immanenten Komponenten zusammen, die bei einer Synchronisation benötigt werden:

Komponenten der Synchronisation

- Transformation
- Validierung

**Abb. 7-3**  
Das Databinding beinhaltet die beiden Phasen Transformation und Validierung.



Für eine Synchronisation können (und müssen) beide Komponenten Transformation und Validierung gekoppelt werden, die dann zu einer bestimmten Phase automatisch aufgerufen werden.

### 7.2.1 Transformation

Der Datentyp des zu konvertierenden Objekts ist in vielen Fällen nicht *kompatibel* mit dem Datentyp des Widgets. Als Beispiel kann hier die Anzeige eines einfachen numerischen Werts in einem Text-Widget genannt werden. Der Integer-Wert des Attributs muss für die Richtung *Model2Widget* in einen String umgewandelt werden, da ein Text ausschließlich Strings darstellt. Dieser Weg ist relativ trivial, da die String-Klasse eine solche Transformation anbietet. Die Richtung *Widget2Model* erfordert die Umwandlung des String-Wertes in den erforderlichen Integer-Wert. Diese Richtung ist insofern interessanter, als

hier Ausnahmen auftreten können, beispielsweise für den Fall, dass der String-Wert nicht in einen Integer-Wert transformiert werden kann. In einem solchen Fall wird die Synchronisation logischerweise unterbrochen, da die Rohdaten nicht in das passende Format transformiert und damit nicht persistiert werden können.

*Ausnahmen*

### 7.2.2 Validierung

Optional können auch Validierungsschritte in einen Synchronisationslauf integriert werden. Hier können dann semantische Überprüfungen entweder auf den *Rohdaten* des Widgets oder auf den transformierten Daten im Zielformat für das Attribut des anhängenden Objekts eingebunden werden. Wichtig ist dies beispielsweise für Intervalle, Datums-grenzen oder komplexere Validierungsregeln. Ebenso wie bei der Transformation können für jede Richtung unterschiedliche Validierungsregeln definiert werden. Ziel der Validierung ist, den Synchronisationslauf im Fehlerfall zu unterbrechen und damit zu verhindern, dass ungültige Daten in das Ziel geschrieben werden.

*Richtungsabhängige  
Validierungsregeln*

### 7.2.3 Phasen der Synchronisation

Wie in den vorherigen Absätzen schon beschrieben, besteht eine Synchronisation aus mehreren Phasen:

**Lesen der Daten** Abhängig von der Richtung werden die Daten aus der Quelle (entweder Model oder Widget) gelesen.

**Transformation der Daten** Zunächst wird das Quell- und Zielformat ermittelt. Das geschieht durch Analyse des Datentyps des Attributs im Modell und des vom Widget unterstützten Datentyps. Für diese Transformation wird der entsprechende Konverter ermittelt. Mittels des Konverters wird der eine Datentyp in den anderen überführt.

**Setzen der Daten** Abhängig von der Richtung werden die Daten in das Ziel (entweder Model oder Widget) geschrieben.

Zwischen die einzelnen Phasen können nun die Validierungen eingehängt werden. Somit kann sowohl der originäre Wert des Widgets, also das Rohformat, als auch das Ergebnis der Transformation einer Validierung unterzogen werden.

### 7.2.4 Synchronisationsarten

Die Synchronisation zwischen Widget und Objekt kann in unterschiedlichen Arten konfiguriert werden. Mit Arten sind hier die Auslöser der

Synchronisation gemeint. Grundsätzlich werden für die Richtung *Widget2Model* drei Auslöser für eine Synchronisation unterschieden:

*Auslöser für eine Synchronisation*

- On Modify** Sobald sich der Wert im Widget durch eine Benutzereingabe in irgendeiner Art und Weise ändert, beispielsweise durch Eingabe von Text, wird die Synchronisation ausgelöst. Für die Synchronisation bedeutet dies, dass die Konvertierung ggf. nicht erfolgreich durchgeführt werden kann, da der Wert des Widgets eventuell noch nicht vollständig eingegeben ist (beispielsweise bei Eingabe eines Datums).
- On Change** Die Synchronisation wird ausgelöst, sobald sich die im Widget dargestellten Daten ändern. Die eigentliche Manipulation löst hier keine Synchronisation aus, erst wenn der Benutzer den Wert des Widgets vollständig manipuliert hat und die Eingabe abgeschlossen ist, wird die Synchronisation ausgelöst.
- On Save** Die Synchronisation wird nicht durch Veränderungen der Daten ausgelöst, sondern erfolgt durch einen manuellen Aufruf auf dem Databinding. Typischerweise wird die Synchronisation dann vor dem Speichern aufgerufen, um die Validität der Daten zu überprüfen.

Für die Richtung *Model2Widget* ist es üblich, dass Änderungen sofort propagiert werden. Man geht davon aus, dass Manipulationen am Modell grundsätzlich persistent und damit auch relevant für die Darstellung sind. Somit wird hier üblicherweise die Synchronisationsart *On Modify* verwendet.

## 7.3 Validierung

Neben dem Feature der automatisierten Synchronisation von Modell und Oberfläche ist die damit einhergehende automatische Validierung ein sehr angenehmer Nebeneffekt. Da die Validierung auf den semantischen Aspekt der Daten schaut, kann sie allerdings auch komplexere Schritte beinhalten.

### 7.3.1 Validierungstypen

Man unterscheidet zwei Arten von Validierungen anhand der zugrunde liegenden Komplexität bzw. der benötigten Daten.

#### Validierung auf Feldebene

Validierung auf Feldebene sind einfache Validierungen, die lediglich die Daten des verbundenen Widgets bzw. Attributs des Objekts benötigen.

*Nutzung der Felddaten*

Sie beziehen sich beispielsweise auf Intervalle, Datumsbereiche oder Gültigkeiten. Diese Art der Validierung ist wenig rechenintensiv und kann im Normalfall ad hoc im Client umgesetzt werden.

### Validierung auf Objektebene

Hierzu im Gegensatz stehen komplexere Validierungen auf Objektebene. Hier werden über das verbundene Feld hinaus Informationen benötigt, um die semantische Korrektheit zu überprüfen. Diese Daten sind entweder bereits im Objekt vorhandene Daten oder Daten von anderen indirekt verbundenen Objekten, die erst über die Persistenzschicht geladen werden müssen.

*Nutzung von  
zusätzlichen Daten des  
Objekts*

**Clientseitige Objektvalidierung** In einigen Fällen ist die Korrektheit eines Feldes nur in Kombination mit einem anderen Feld zu überprüfen. Soll beispielsweise ein Gültigkeitszeitraum angegeben werden, so muss das Gültigkeitsende nach dem Gültigkeitsbeginn liegen. Diese Validierungen benötigen typischerweise nur Daten, die auf der Clientseite bereits vorhanden sind.

**Serverseitige Objektvalidierung** Als Beispiel für eine Validierung, die zusätzliche Daten über die Persistenzschicht benötigt, kann die Prüfung einer Bestellung angeführt werden. Für die Validierung der eingegebenen Produktnummer muss zunächst gegen den Produktkatalog geprüft werden. Wenn die Produktnummer gültig ist, muss der Bestand geprüft werden, ob die angegebene Bestellmenge verfügbar ist. Beide Validierungen benötigen Informationen, die sinnvollerweise nicht auf dem Client vorliegen. Zudem ist die zweite Validierung abhängig vom Ergebnis der ersten Validierung.

Komplexe Validierungen sind eher rechenintensiv bzw. benötigen tendenziell weitergehende Daten. Beide Aspekte sprechen dafür, diese Validierungsschritte serverseitig durchzuführen, um Speicherbedarf, benötigte Prozessorzeit und Kommunikationsbedarf auf Clientseite zu minimieren.

### 7.3.2 Ergebnis der Validierung

Das Ergebnis der Validierung hat mehrere Aspekte. Zum einen soll mitunter die aktuell ausgeführte Funktion unterbrochen werden. Dies ist zumeist für *On Save*-Validierung der Fall, wenn Fehler auftauchen. Eine Unterbrechung ist bei *On Modify* und *On Change* nicht gewünscht, da es hier darum geht, dem Benutzer möglichst früh eine fehlerhafte Eingabe darzustellen und Hinweise zur Korrektur zu geben.

*Unterbrechung der  
Funktion*

## Meldungen

### Arten von Meldungen

Das Ergebnis einer Validierung muss nicht unbedingt einen Fehler darstellen. In allen Fällen, *On Modify*, *On Change* und *On Save*, können neben Fehlern auch Warnungen oder Informationen auftauchen, die innerhalb der Validierungslogik bemerkt wurden. Das Ergebnis einer Validierung ist also kein einfacher boolescher Rückgabewert, sondern stellt ein Ergebnisobjekt dar. Dieses Objekt beinhaltet dabei neben dem Validierungsergebnis, beispielsweise *Fehler*, *Warnung* und *Information*, häufig eine fachliche und/oder technische Meldung. Unabhängig davon, was das Ergebnis einer Buchungsvalidierung ist, kann eine Meldung wie *Bitte überweisen Sie den Betrag innerhalb der nächsten 10 Tage* zusätzlich Ergebnis einer Validierung sein. Das Validierungsergebnis muss dann auf eine Unterbrechung der gerade ausgeführten Aktion hin interpretiert werden. Eine Fehlermeldung kann beispielsweise eine fehlgeschlagene Validierung bedeuten, eine Warnung bzw. Information zeigt dagegen eine erfolgreiche Validierung an.

### Ergebnisobjekte

## Darstellung

Das Ergebnis der Validierung, also die vorhandenen Meldungen, muss dem Benutzer in einer verständlichen Art und Weise präsentiert werden. Es gibt hier zwei unterschiedliche Ansätze.

### Zentrale Fehlerliste

Zum einen kann eine zentrale Fehlerliste verwendet werden, die vorhandene Meldungen möglichst kontextsensitiv zum gerade aktiven Objekt anzeigt. Dieses Vorgehen hat allerdings einen Nachteil, denn die Meldungen stehen in keinem direkten Bezug zum fehlerhaften Widget. Zwar kann die Meldung textuell und über einen Link mit dem Widget verbunden werden, optisch sind beide Elemente jedoch voneinander getrennt.

### Integrierte Fehlermeldungen

Die zweite Alternative integriert die Fehlermeldungen in die Eingabemaske, die fehlerhaften Elemente erhalten einen entsprechenden Marker. Auch dieses Vorgehen hat Nachteile. Zum einen kann im Normalfall aus Platzmangel die Fehlermeldung nicht direkt dargestellt werden, da sonst das Layout der Maske gestört wird. So tauchen die Meldungen dann im Normalfall erst beim *Mouse-Over* auf dem Marker auf. Zum anderen sind in Eingabemasken, die mit Reitern arbeiten, nicht alle Fehler in ihrer Gesamtheit sichtbar, da sie auf die einzelnen Reiter verteilt sind.

In der Praxis hat sich eine Kombination aus beiden Vorgehensweisen bewährt. Eine kontextsensitive Fehlerliste schafft den Überblick über die gesamte Fehlerliste, die Kennzeichnung der tatsächlichen fehlerhaften Widgets ermöglicht eine schnelle Navigation zu den zu korrigierenden Elementen.

### 7.3.3 Einbindung der Validierung

Bei der Einbindung der Validierung in das Databinding sollte auf ein einheitliches Vorgehen bzw. eine einheitliche Umsetzung geachtet werden. Wenn die einzelnen Validierungskomponenten dieselben Mechanismen verwenden, um registriert, gesteuert, aufgerufen und verarbeitet zu werden, können die fachlich getriebenen Logiken sehr einfach wiederverwendet werden. Idealerweise erweitert man das Standard-Databinding-Framework um die speziellen Funktionen und Algorithmen, die benötigt werden. So nutzt man die etablierten Mechanismen und stellt sicher, dass der vorgegebene Ablauf eingehalten wird.

## 7.4 Eclipse Databinding

Das Eclipse Databinding stellt ein Databinding-Framework dar. Es bietet Klassen und Methoden, um eine Benutzeroberfläche mit Attributen eines Objekts zu verbinden und die Werte jeweils zu synchronisieren.

### 7.4.1 DataBindingContext

Die zentrale Klasse des Eclipse Databindings ist der sogenannte `DataBindingContext`. Der Kontext ermöglicht es, Verbindungen zwischen einem Widget und einem Objektattribut zu erstellen. Das so erzeugte `Binding` arbeitet allerdings nicht direkt auf dem Widget und einem Objektattribut, es verwendet für beide Seiten Adapter, sogenannte `IObservableValue`. Ein `IObservableValue` arbeitet nach dem Observer-Entwurfsmuster[11], es überwacht demzufolge den Zustand des adaptierten Objekts. Im Fall des Databindings ist dieses Objekt zum einen das Widget und zum anderen das Attribut des Objekts. Ein `Binding` erhebt individuelle Informationen für jede Richtung, somit wird sowohl ein `IObservableValue` für das Widget als auch ein `IObservableValue` für das Attribut des Objektes benötigt, welche verbunden werden sollen.

*Bindings*

*Observer-  
Entwurfsmuster*

### 7.4.2 IObservableValue

Das Eclipse Databinding liefert sowohl für die gängigen Widgets als auch für Attribute eines Objektes bereits `IObservableValue`-Klassen mit aus, die zum Überwachen für ein `Binding` verwendet werden können.

#### Widgets überwachen

Die Factory Klasse `SWTObservables` bietet statische Methoden, um mit Angabe des gewünschten Eventtyps ein entsprechendes `IObservableValue` zu erzeugen. Der Eventtyp ist jenes Event, das einen automatischen

*Die Klasse  
SWTObservables*

Synchronisationslauf zwischen View und Modell initiiert. Das so erzeugte `IObservableValue` kann dann direkt für ein `Binding` verwendet werden. Für spezielle Widgets ist die Implementierung einer eigenen `IObservableValue`-Klasse notwendig.

### Objekte überwachen

Das Observer-Entwurfsmuster impliziert, dass Veränderungen auf dem überwachten Objekt propagiert werden. Hierfür muss eine solche Veränderung jedoch überhaupt verfolgt werden können. Zur Überwachung eines Attributs eines Objekts gibt es zwei unterschiedliche Ansätze, abhängig davon, ob das überwachte Objekt Manipulationen der Attribute propagiert oder nicht.

*JavaBeans  
überwachen*

**JavaBeans** In der ersten Variante wird also davon ausgegangen, dass das überwachte Objekt die Änderungen entsprechend der JavaBean-Spezifikation [34] propagiert, beispielsweise unter Zuhilfenahme des `PropertyChangeSupport`. Dieser ermöglicht es, `PropertyChangeListener` für bestimmte Attribute zu registrieren. Die Setter-Methoden der JavaBean müssen hier allerdings im Falle einer Manipulation die Methode `firePropertyChange(String, Object, Object)` des `PropertyChangeSupport` aufrufen, um die angemeldeten `PropertyChangeListener` zu benachrichtigen. Mit der Klasse `BeansObservables` kann für solche JavaBeans ein entsprechendes `IObservableValue` für ein bestimmtes Attribut erzeugt werden. Dieses Vorgehen hat den Vorteil, dass Manipulationen im Modell direkt an das Widget gemeldet werden und automatisch ein Update erfolgt.

*POJOs überwachen*

**POJOs** Die überwiegende Zahl von Objekten, seien es Value Objects (VO), Data Transfer Objects (DTO) oder O/R-Mapper-Objekte wie Hibernate Beans, entsprechen leider nicht der JavaBean-Spezifikation. Für solche Objekte kann mit der Klasse `PojoObservables` für Attribute ein entsprechendes `IObservableValue` erzeugt werden, das im `Binding` verwendet wird. Hier führt eine Manipulation im Modell allerdings nicht zur Änderung des verbundenen Widgets, da der entsprechende Benachrichtigungsmechanismus fehlt. Änderungen auf dem Modell müssen also entweder durch manuelles Auslösen der Synchronisation oder durch Manipulation auf den `IObservableValues` erfolgen.

### 7.4.3 Synchronisationsarten

*UpdateValueStrategy*

Das Eclipse Databinding bietet zur Steuerung der Synchronisation die Klasse `UpdateValueStrategy` an. Ein `Binding` benötigt zwei `UpdateValueStrategy`-Objekte: jeweils eines für jede Richtung, also *Model2Widget*

und *Widget2Model*. Eine Strategie erlaubt es zum einen, die Wirkungsweise der Synchronisation zu definieren. Dies geschieht über die sogenannte *Update Policy*, wobei zwischen den folgenden Arten gewählt werden kann:

- NEVER** Die Werte der Quelle werden nicht in das Ziel synchronisiert.
- ON REQUEST** Die Werte werden nur durch manuelles Aufrufen der Synchronisation von der Quelle in das Ziel synchronisiert. Dies beinhaltet dann sowohl den Aufruf der Transformation als auch den Aufruf der registrierten Validierungen.
- CONVERT** Änderungen auf den Quellen werden verfolgt, die Konvertierungen und Validierungen werden automatisch aufgerufen. Allerdings werden dann die Werte nicht automatisch von der Quelle in das Ziel synchronisiert. Dies geschieht nur beim manuellen Aufrufen der Synchronisation.
- UPDATE** Änderungen auf den Quellen werden verfolgt, die Konvertierungen und Validierungen werden automatisch aufgerufen. Die Werte werden automatisch von der Quelle in das Ziel synchronisiert.

*Update Policies*

Die *Update Policy* steuert also, ob die Synchronisation automatisch erfolgt oder nicht. Eine automatische Synchronisation wird dann durch die Konfiguration des *IObservableValue* getriggert. Die Synchronisationsart definiert sich beim Eclipse Databinding also für jedes Binding durch die Kombination aus der Konfiguration des *IObservableValue* und der *Update Policy* der entsprechenden *UpdateValueStrategy*.

#### 7.4.4 Konvertierung

Die *UpdateValueStrategy* ist das steuernde Objekt des Bindings, daher wird auch hier die Konvertierung der anhängenden Werte konfiguriert. Sowohl das *IObservableValue* der Quelle als auch das des Ziels definieren den Datentyp des beherbergten Werts. Für die Konvertierung zwischen den beiden Datentypen wird ein Converter benötigt. Für die Konvertierung zwischen den Standarddatentypen stehen bereits Converter zur Verfügung. Andernfalls bietet die *UpdateValueStrategy* die Möglichkeit, einen benutzerdefinierten Converter zu verwenden, dieser wird dann mittels `setConverter(Converter)` gesetzt.

*Converter*

#### 7.4.5 Validierung

Für die Validierung werden im Rahmen des Eclipse Databindings sogenannte *IValidator*-Objekte verwendet. Mit der Methode `valida-`

*IValidator*

te(Object) kann dieses ein IStatus-Objekt als Validierungsergebnis zurückliefern. Die Rückgabe definiert somit zum einen das Validierungsergebnis und liefert ggf. die anhängende Meldung.

### Phasen

Entsprechend der beschriebenen Phasen können IValidator-Objekte für unterschiedliche Zeitpunkte bei der UpdateValueStrategy registriert werden. Die Phasen können auf beiden Seiten einer Verbindung genutzt werden.

Zeitpunkte für  
Validierungen

**setAfterGetValidator(IValidator)** Der übergebene IValidator wird aufgerufen, nachdem der Wert aus der Quelle gelesen wurde. Der Datentyp, der in der validate(Object)-Methode übergeben wird, entspricht dem des IObservableValue der Quelle.

**setAfterConvertValidator(IValidator)** Der übergebene IValidator wird aufgerufen, nachdem der Wert aus der Quelle gelesen und mittels des passenden Converters transformiert wurde. Der Datentyp, der in der validate(Object)-Methode übergeben wird, entspricht dem des IObservableValue des Ziels.

**setBeforeSetValidator(IValidator)** Der übergebene IValidator wird analog zu dem *afterConvert* IValidator aufgerufen. Der Unterschied ist hier von technischer Natur und bezieht sich auf die *Update Policy* der UpdateValueStrategy. Der *beforeSet* IValidator wird im Gegensatz zum *afterConvert* IValidator im Falle einer *CONVERT Update Policy* nicht aufgerufen.

### Ebenen

Ein IValidator bekommt abhängig von der gewählten Phase zur Validierung den Feldwert entweder transformiert oder nicht transformiert übergeben. Die Validierung auf Feldebene ist somit kein Problem. Interessant wird es bei Validierungen auf Objektebene.

**Clientseitige Objektvalidierung** Da der IValidator wie gesagt keinen direkten Zugriff auf das gesamte Objekt hat, benötigt er für eine Validierung auf Objektebene eine Referenz. Für die Validierung ist allerdings der aktuelle Zustand des Objekts, d.h. der Zustand aus der Maske, notwendig. Hier empfiehlt sich, die zusätzlich benötigten Daten über eine *On Modify*- bzw. *On Change*-Validierung in das Objekt zu synchronisieren. So entspricht der Zustand des Objekts dem der Maske. Die Validierung kann also direkt auf dem Objekt ausgeführt werden, wenn alle notwendigen Daten zur Verfügung stehen.

**Serverseitige Objektvalidierung** Für die Fälle, bei denen zusätzliche, nicht im Objekt bzw. in der Maske vorhandene Daten benötigt werden, ist ein Zugriff auf das Backend notwendig. Hier eignet sich die Nutzung einer *On Modify*- bzw. *On Change*-Validierung eher nicht, da jede Eingabe automatisch mit einem Serverzugriff verbunden wäre. Um die Zugriffe auf den Server zu reduzieren, sollte daher auf eine *On Save*-Validierung zurückgegriffen werden. Betrachtet man die Funktionsweise des Databinding ist dieses Vorgehen allein jedoch immer noch unvorteilhaft. Jedes Binding würde vor dem Speichervorgang einen separaten Zugriff zum Server tätigen, um das eigene Validierungsergebnis zu erfragen. Um die Serverzugriffe auf ein Minimum zu reduzieren, muss die Systematik der Validierung ein wenig verändert werden.

Die serverseitige Validierung erfolgt im Idealfall über einen einzigen Aufruf des Servers. Dieser Aufruf überprüft sämtliche Aspekte des Objekts, welche nicht clientseitig überprüft werden können. Als Ergebnis werden wie gewohnt die Meldungen für die entsprechenden Attribute bzw. Widgets zurückgeliefert. Doch passt dieses rein objektzentrierte Vorgehen zunächst nicht zur Systematik des Eclipse Databinding, das mit den Bindings auf Attributebene prüft. Beide Ansätze lassen sich jedoch über eine leicht modifizierte Verwendung der *IValidator*-Objekte vereinen.

*Serverseitige  
Validierungsroutinen*

Da die eigentliche Validierung komplett serverseitig erfolgt, bleibt für das *IValidator*-Objekt die Aufgabe, festzustellen, ob für dessen abhängendes Attribut vom Server Fehler gemeldet wurden. So kann der zentrale Validierungsmechanismus über den *DataBindingContext* weiterverwendet werden. Die serverseitige Validierungsroutine kann jederzeit aufgerufen werden, wobei auftretende Fehler automatisch durch den Databinding-Mechanismus einheitlich dargestellt werden.

*Clientseitige  
Interpretation*

### 7.4.6 Darstellung

Das Eclipse Databinding verfolgt im Standard einen feldbezogenen Ansatz zur Darstellung der Fehler. So erlaubt das Databinding nicht nur Bindings für ein Attribut und ein Widget bezogen auf dessen Inhalt, sondern es sind auch Bindings für ein Attribut und eine Eigenschaft des Widgets möglich. Beispielsweise kann der Name eines Projekts mit der Hintergrundfarbe des zugehörigen Widgets verbunden werden. Für die korrekte Übersetzung wird dann allerdings ein eigener Converter benötigt, der den Attributwert in die entsprechende Farbe umwandelt. Die Klasse *SWTObservable* bietet für mehrere Eigenschaften *IObservableValues* an:

Weitere Möglichkeiten

**observeForeground(Control)** Ändern der Vordergrundfarbe  
**observeBackground(Control)** Ändern der Hintergrundfarbe  
**observeEditable(Control)** Ändern des »editierbar«-Attributs  
**observeEnabled(Control)** Ändern des »verfügbar«-Attributs  
**observeVisible(Control)** Ändern der Sichtbarkeit  
**observeToolTipText(Control)** Ändern des Tooltips  
**observeSelection(Control)** Ändern der Auswahl  
**observeMin(Control)** Ändern des Minimalwerts  
**observeMax(Control)** Ändern des Maximalwerts  
**observeFont(Control)** Ändern der Schrift

One-way-  
Synchronisation

Die `IObservableValue`s werden nur für die Widgets angeboten, für die sie auch sinnvoll einzusetzen sind bzw. für die sie implementiert werden können. Wichtig ist hierbei, dass nur für die Richtung *ModelToTarget* eine Transformation, also eine transformierende `UpdateValueStrategy` definiert wird. Um zu vermeiden, dass die jeweilige Eigenschaft des Widgets als Wert für das Attribut in das Objekt geschrieben wird, benötigt die `UpdateValueStrategy` für die Richtung *TargetToModel* eine *NEVER Policy*.

**Listing 7.1**  
Binding für die  
Hintergrundfarbe eines  
Text-Widgets.

```
// Anlegen des IObservableValue für die Hintergrundfarbe
ISWTObservableValue descriptionBackground = SWTObservables.
    observeBackground(textDescription);
UpdateValueStrategy updateValueStrategy =
    new UpdateValueStrategy();
updateValueStrategy.setConverter(new IConverter(){
    public Object getFromType() {
        return String.class;
    }

    public Object getToType() {
        return Color.class;
    }

    public Object convert(Object fromObject) {
        String description = (String) fromObject;
        // endet die Beschreibung mit 'gelb'
        // färbt sich der Hintergrund gelb
        if(description != null &&
            description.endsWith("gelb")){
            return ResourceManager.
                getColor(SWT.COLOR_YELLOW);
        }
        return null;
    }
});
```

```
// Anlegen des Bindings mit den Observable Values
// und einer NEVER policy für TargetToModel
bindingContext.bindValue(descriptionBackground, description, new
    UpdateValueStrategy(UpdateValueStrategy.POLICY_NEVER),
    updateValueStrategy);
```

Es ist so möglich, Fehler direkt am Eingabeelement darzustellen. Leider ist dieses Vorgehen insofern etwas mühsam, als dass für jedes Widget ein weiteres Binding angelegt werden muss. Zusätzlich wird mit dieser Vorgehensweise ein Teil der Validierungslogik nicht von der Validierung selbst in Form des `IValidator` gelöst, sondern durch den `IConverter`, also von der Konvertierung. Wie bereits erwähnt, fehlt auch eine Komplettansicht aller Fehler des entsprechenden Editors.

### Zentrale Fehlerliste

Für die Implementierung einer zentralen Fehlerliste gibt es zwei Möglichkeiten. Zum einen kann der schon existierende *Problems View* der Plattform verwendet werden. Da dieser allerdings ebenfalls für Laufzeitfehler und ähnliche Meldungen verwendet wird, kann auch ein eigener View implementiert werden, der die Eingabefehler als Liste (oder Baum) präsentiert. Der View funktioniert dann ähnlich wie die *Outline*, also kontextsensitiv zum aktiven Editor. Allerdings ergibt sich hier der Effekt, dass Fehlermeldung und Eingabeelement nicht zusammen angezeigt werden.

### Forms-Fehlerdisplay

Als Weg der Mitte, also der Kombination aus feldbezogener Darstellung und einer Fehlerliste, dient für Editoren das *Forms API*. Es ermöglicht, unter Einbindung der für Forms-Editoren standardmäßig eingesetzten `ScrolledForm`, zum einen die Darstellung einer zentralen Fehlerliste und mit Verwendung von sogenannten *Decorator*-Objekten zusätzlich die Kennzeichnung der fehlerhaften Widgets. Die hier vorgestellte Lösung basiert auf einem Snippet der *IBM*.

**Fehlerliste** Wie bereits erwähnt, stellt bei diesem Vorgehen die `ScrolledForm` des Editors den Dreh- und Angelpunkt dar. Sie stellt zwei wichtige Dinge zur Verfügung, zum einen den `IMessageManager`, der eine Menge von Nachrichten verwalten kann.

**Listing 7.2** // Anlegen des benötigten ManagedForm  
 Erstellen des IManagedForm managedForm = **new** ManagedForm(formToolkit,  
 IMessageManagers scrolledForm);  
 // Die ManagedForm bietet dann den MessageManager  
 IMessageManager messageManager = managedForm.getMessageManager();

Darüber hinaus bietet die ScrolledForm eine sogenannte *Message Area*, einen Bereich neben dem Titel, in dem Nachrichten angezeigt werden können. Die dem IMessageManager hinzugefügten Nachrichten werden dann automatisch von ihm in der *Message Area* angezeigt. Sollten mehrere Nachrichten vorhanden sein, taucht die Anzahl der vorhandenen Nachrichten auf.

Um nun eine zentrale Liste aller Nachrichten des Editors zu erhalten, wird auf der Message Area ein zusätzliches Popup registriert, welches diese Liste erzeugt und darstellt.

**Listing 7.3** // Registrieren eines HyperlinkListeners, der beim  
 Erstellen des Popups // Klicken auf die Message Area das Popup öffnet  
 zur Darstellung aller scrolledForm.getForm().addMessageHyperlinkListener(**new**  
 vorhandenen HyperlinkAdapter() {  
 Nachrichten **public void** linkActivated(HyperlinkEvent event) {  
**if** (currentErrorShell != **null**) {  
 currentErrorShell.dispose();  
 currentErrorShell = **null**;  
 }  
 String title = event.getLabel();  
 Object href = event.getHref();  
 Point hl =  
 ((Control)event.widget).toDisplay(0, 0);  
 hl.x += 10;  
 hl.y += 10;  
 currentErrorShell = **new** Shell(scrolledForm.getForm().  
 getShell(), SWT.ON\_TOP | SWT.TOOL);  
 currentErrorShell.setImage(getImage(scrolledForm.getForm().  
 getMessageType()));  
 currentErrorShell.setText(title);  
 FillLayout fillLayout = **new** FillLayout();  
 fillLayout.marginHeight = 4;  
 fillLayout.marginWidth = 1;  
 currentErrorShell.setLayout(fillLayout);  
 currentErrorShell.setBackground(ResourceManager.getColor(SWT.  
 COLOR\_WHITE));  
 FormText text = formToolkit.createFormText(currentErrorShell,  
**true**);  
 configureFormText(scrolledForm.getForm(), text);  
**if** (href instanceof IMessage[]){

```

        text.setText(createErrorShellContent(
            (IMessage[]) href), true, false);
    }
    currentErrorShell.setLocation(h1);
    currentErrorShell.pack();
    currentErrorShell.open();
}
});

```

Im Quellcode tauchen zwei Methodenaufrufe auf, die näher erläutert werden müssen.

Die Methode `createErrorShellContent(IMessage[])` wandelt die registrierten Messages in den Text des Popups inklusive Icon um.

```

private String createErrorShellContent(IMessage[] messages) {
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);
    pw.println("<form>");
    // Jede Message als Listeneintrag darstellen
    for (int i = 0; i < messages.length; i++) {
        IMessage message = messages[i];
        pw.print("<li vspace=\"false\" \" +
            \"style=\"image\" indent=\"16\" \" +
            \"value=\"");
        switch (message.getMessageType()) {
            case IMessageProvider.ERROR:
                pw.print("error");
                break;
            case IMessageProvider.WARNING:
                pw.print("warning");
                break;
            case IMessageProvider.
                INFORMATION:
                pw.print("info");
                break;
        }
        pw.print("> <a href=\"");
        pw.print(i + "\"");
        pw.print(">");
        if (message.getPräfix() != null)
            pw.print(message.getPräfix());
        pw.print(message.getMessage());
        pw.println("</a></li>");
    }
}
pw.println("</form>");

```

#### Listing 7.4

*Erstellen des Textes für die vorhandenen Nachrichten*

```
pw.flush();
return sw.toString();
```

Wie zu erkennen ist, handelt es sich bei dem Content um einfaches HTML. Es besteht aus einem *form*-Tag, das eine Menge von *li*-Tags, also eine Liste, führt. Jede Message bekommt einen eigenen Eintrag in dieser Liste spendiert.

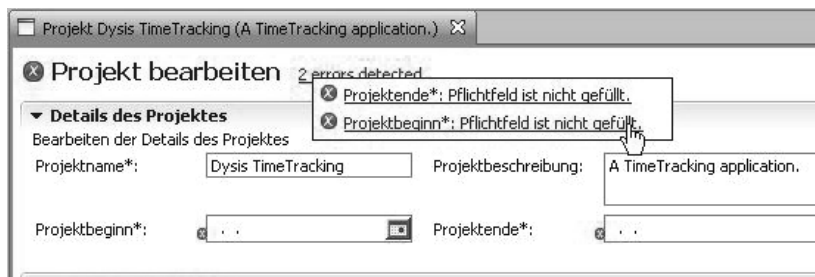
Ist die Message mit einem Control verbunden, so soll auf dieses navigiert werden können. Dies ermöglicht die Methode `configureFormText(Form, FormText)`.

**Listing 7.5**  
Einrichten des Links  
der Messages auf das  
anhängende Widget

```
private void configureFormText(final Form form, FormText formText)
{
    formText.addHyperlinkListener(new HyperlinkAdapter() {
        public void linkActivated(HyperlinkEvent e) {
            String is = (String) e.getHref();
            try {
                int index = Integer.parseInt(is);
                IMessage[] messages = form.getChildrenMessages();
                IMessage message = messages[index];
                Control c = message.getControl();
                ((FormText) e.widget).getShell().dispose();
                if (c != null && !c.isDisposed()){
                    c.setFocus();
                }
            }
            catch (NumberFormatException exception) {
                exception.printStackTrace();
            }
        }
    });
}
```

Es legt einen `HyperlinkListener` auf die einzelnen Message-Texte. Ist mit der dargestellten `IMessage` ein `Control` verbunden, so wird es fokussiert, sollte auf die Message geklickt werden.

**Abb. 7-4**  
Forms-Fehlerdisplay



**Messages und Decorator** Die offen gebliebene Frage ist nun, wie die Messages erzeugt und dem IMessageManager zur Verfügung gestellt werden. Grundlage der Messages ist zunächst einmal das Ergebnis der registrierten IValidator-Objekte der anhängigen Bindings. Um nun aber das Ergebnis direkt zu verwenden, werden die Validatoren mit einer Wrapper-Klasse umhüllt.

```
public IValidator adapt(final IValidator validator, final Object
    validatorId, final Control control, final String message) {
    return new IValidator() {
        public IStatus validate(Object value) {
            IStatus status = validator.validate(value);
            if (status.isOK()) {
                messageManager.removeMessage(validatorId, control);
            }
            else {
                messageManager.addMessage(validatorId, message, null,
                    statusToMessageType(status), control);
            }
            return status;
        }
    };
}
```

**Listing 7.6**  
Wrappen der  
IValidator-Objekte

Dieser Wrapper delegiert den Aufruf und den umhüllten IValidator und fügt dem IMessageManager ggf. die entsprechende Meldung hinzu. Die Übergabe des Control-Objekts bei der Erstellung der Message bewirkt das Anzeigen des Decorators am Widget des unterliegenden Bindings.

**Messages aus der serverseitigen Validierung** Die Darstellung der Messages, die aus der serverseitigen Validierung hervorgehen, kann sehr leicht integriert werden. Das Codebeispiel verwendet einen sogenannten ValidationMessageManager. Dieses Objekt sucht aus der Liste von IValidationMessage diejenigen heraus, die zu einem bestimmten Schlüssel, dem sogenannten *referenceKey*, passen. Diese Zuordnung muss zum einen serverseitig erfolgen, aber clientseitig auch interpretierbar sein. Jeder Validator muss wissen, auf welchen *referenceKey* er hört.

```
public IStatus validate(Object object) {
    List<IValidationMessage> validationMessages =
        validationMessageManager.getValidationMessages(referenceKey)
        ;
    if (!validationMessages.isEmpty()) {
        StringBuffer errorMessageBuffer = new StringBuffer();
        int type = IMessageProvider.INFORMATION;
    }
}
```

**Listing 7.7**  
Wrappen der  
validate()-Methode der  
IValidator-Objekte

```
for (IValidationMessage validationMessage : validationMessages
) {
    errorMessageBuffer.append(validationMessage.getMessage());
    errorMessageBuffer.append(";");
    validationMessageManager.validationMessageRead(
        validationMessage);
    if (type == IMessageProvider.INFORMATION) {
        type = validationMessage.getType();
    }
    else if (type == IMessageProvider.WARNING &&
        validationMessage.getType() == IMessageProvider.ERROR)
    {
        type = validationMessage.getType();
    }
}
return new Status(messageTypeToStatus(type), Activator.
    PLUGIN_ID, errorMessageBuffer.toString());
}
return Status.OK_STATUS;
}
```

Die einzelnen `IValidationMessages` werden textuell konkateniert und als `IStatus` zurückgegeben. Adaptiert man diesen `IValidator` nun mit dem beschriebenen Wrapper, werden die serverseitig erstellten Nachrichten automatisch im Editor analog zu den clientseitigen Messages dargestellt. Die Validierung erfolgt so über einen einheitlichen Mechanismus und wird in einer ebenfalls einheitlichen Weise dargestellt.