Das Ende des Refresh Buttons

Wir beschreiben eine Client-Server Architektur, in der Clients über gegenseitige Objektänderungen informiert werden und damit ihr Datenmodell und GUI automatisch aktuell halten. Auf Aktualisierungsschaltflächen kann daher verzichtet werden. In einer solchen Architektur ist es wichtig, dass Objektveränderungen kontrolliert erfolgen. Dazu unterwerfen wir die Domänenobjekte auf Seite der Clients einem strikten Lebenszyklus, welcher stets konsistente Zustände garantiert. Unsern Ansatz haben wir in Java umgesetzt und in einem Projekt zusammen mit dem Paul Scherer Institut auf seine Tauglichkeit überprüft.

Daniel Kröni | daniel.kroeni@fhnw.ch

Am Protonentherapiezentrum des Paul Scherrer Instituts [PSI] werden Tumore mit Protonen beschossen, um die Tumorzellen zu schädigen. Die dazu notwendige Anlage ist komplex und muss vor jedem Einsatz umfangreichen Tests unterzogen werden. Die verantwortlichen Operatoren der Bestrahlungsanlage treffen kritische Entscheidungen basierend auf den Testresultaten. Daher ist es wichtig, dass die Bildschirmmasken nie veraltete Daten anzeigen. Die dazu notwendige Datenakutalisierung soll ohne Betätigung einer *Refresh*-Schaltfläche automatisch erfolgen.

Um die Bildschirmmasken automatisch aktuell zu halten, haben wir folgenden Ansatz gewählt: Das Domänenmodell [DM] wird zentral auf einem Server verwaltet und die Aktualisierungen werden allen verbundenen Clients mitgeteilt. Diese Clients arbeiten auf ihrer individuellen, lokalen Kopie eines Teils des Domänenmodells. Betrachten wir dazu Abbildung 1: Client *A* modifiziert lokal ein Domänenobjekt (1). Die lokalen Veränderungen sind in seinem GUI unmittelbar sichtbar. Um die Änderungen zu speichern, wird das Objekt zum Server transferiert (2). Der Server übernimmt die Änderung in sein zentrales Modell (3) und veröffentlicht[1] die Differenz (4). Jeder Client reagiert auf Differenzmeldungen und aktualisiert damit seine lokale Kopie (5). Die Objekte feuern dabei Änderungsnotifikationen, welche schliesslich mittels Databinding zu einer Aktualisierung des GUIs führen [DBind].

Die Clients der vorgestellten Architektur beherbergen einen signifikanten Teil der Programmlogik (Rich Clients). Daher macht es Sinn, dem Programmierer der Client-Software ein komfortables und navigierbares Datenmodell anzubieten.

Im ersten Abschnitt dieses Berichts präsentieren wir ein Konzept, das es uns ermöglicht, den Clients das ganze Domänenmodell des Servers anzubieten, ohne die ganze Datenbank auf den Client zu laden. In diesem verteilten System ist es kritisch, das Domänenmodell konsistent zu halten. Im zweiten Abschnitt zeigen wir, dass die Domänenobjekte kontrolliert verändert werden müssen, um Konsistenz zu garantieren. Wie diese Architektur in Java umgesetzt werden kann, beschrieben wir im dritten Abschnitt. In Abschnitt vier erklären wir, wie die Differenzobjekte modelliert werden können. Wir schliessen den Bericht mit einem kurzen Fazit.

Umgang mit Referenzen

Das Domänenmodell einer Businessapplikation repräsentiert die Objekte der Anwendungsdomäne. In unserer Anwendung sind das z.B. Qualitätschecks für die Bestrahlungsmaschine und die dazu erfassten Messwerte. Solche Domänenobjekte enthalten Referenzen, d.h. Verweise auf weitere Domänenobjekte sowie primitive Daten. Normalerweise werden solche Datenmodelle in Objektgraphen abgebildet. Wir haben für unsere Anwendung jedoch ein spezielles Modell realisiert. Anstelle von Referenzen auf andere Objekte werden nur die IDs der referenzierten Domänenobjekte abgelegt. Bei jedem Referenzzugriff, wird das referenzierte Objekt anhand der entsprechenden ID ausfindig gemacht. Dieses Speichermodell ist vergleichbar mit dem Entitätenmodell einer relationalen Datenbank, in welchem Verweise zwischen Datensätzen auf Fremdschlüsseln basieren. Die Werte von Attributen werden hingegen direkt abgelegt.

Durch dieses Speichermodell werden die Domänenobjekte voneinander entkoppelt. Ein Domänenobjekt referenziert nicht direkt weitere Domänenobjekte, sondern beinhaltet nur die Information (die IDs), welche Domänenobjekte von ihm referenziert werden. Statt eines Objektgraphen haben wir nun eine Menge von unabhängigen Objekten. Um von einer Objekt ID zum entsprechenden Objekt zu gelangen, wird eine *Resolve*-Funktion benötigt. Wir trennen also die Eigenschaft einer Referenz in zwei Konzepte: einerseits die Identifikation und anderseits die

1 Im Sinne des Publish / Subscribe Messaging Architektur Musters [PubSub]

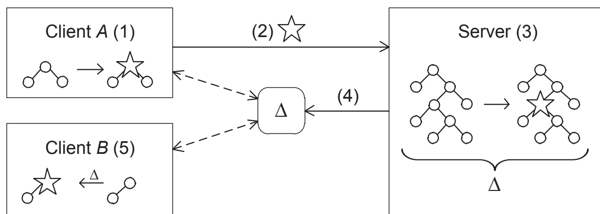

Abbildung 1: Ablauf beim Speichern von veränderten Daten

Bereitstellung des referenzierten Objektes. Diese Trennung hat zwei entscheidende Vorteile:

- Abstraktion über Referenzzugriffe: erlaubt unterschiedliche Lookup-Strategien;
- Entkopplung der Objekte: Objekte können separat prozessiert werden.

Die Trennung erlaubt uns sowohl auf dem Server als auch auf den Clients die gleichen Domänenmodellklassen einzusetzen. Der einzige Unterschied zwischen Client- und Servermodell ist die verwendete *Resolve*-Funktion. Während auf dem Server die *Resolve*-Funktion auf die Datenbank zugreift und die benötigten Objekte lädt, sucht sie beim Client das Objekt in einem lokalen Objekt-Cache. Ist das Objekt dort nicht auffindbar, so wird es vom Server angefordert. Folgende Vorteile lassen sich bei diesem Speichermodell feststellen:

- Jedes Domänenobjekt kann separat serialisiert werden. Weil für Referenzen nur IDs übertagen werden, können keine Alias-Probleme auftreten.
- Da immer zuerst der Cache durchsucht wird, kann garantiert werden, dass für jedes logische Objekt nur eine Instanz pro Client existiert. Dadurch muss nur diese eine Instanz aktuell und konsistent gehalten werden.
- Die referenzierten Objekte werden erst beim Zugriff geladen. Durch die Zwischenschaltung eines Objekt-Caches kann garantiert werden, dass nur beim ersten Referenzzugriff auf die Version des Servers zurückgegriffen wird. Das heisst, nur benötigte Objekte werden übertragen und das auch erst bei Bedarf (lazy loading). Um unnötigen Kommunikationsaufwand zu verhindern, wäre es auch denkbar, bestimmte Referenzen als eager zu markieren, mit der Bedeutung, dass so referenzierte Objekte zusammen mit dem referenzierenden Objekt übertragen werden.
- Auf Client und Server sind die gleichen Domänenmodellklassen im Einsatz. Der Client arbeitet auf demselben komfortabel navigierbaren Domänenmodell wie der Server. Das vereinfacht die Entwicklung des Systems, da keine Datentransferobjekte [DTO] benötigt werden.
- Nicht mehr verwendete Domänenobjekte können vom Garbage Collector abgeräumt werden. Die Resolve-Funktion bzw. der Objekt-Cache kann eine passende Caching Strategie (z.B. LRU) implementieren. Werden abgeräumte Objekte trotzdem wieder benötigt, so müssen sie erneut vom Server nachgeladen werden.

Zusammengefasst bietet dieses Speichermodell die Vorteile von leichtgewichtigen DTOs kombiniert mit der komfortablen Navigation eines Domänenmodells. Zu beachten ist, dass das vorgestellte Domänenmodell nur die Daten und Verknüpfungen enthält. Businesslogik wird nicht in den Domänenobjekten implementiert, sondern über Services angeboten. Auch das Erzeugen, Modifizieren und Löschen von Domänenobjekten wird über Services abgehandelt. Services bieten klare Transaktionsgrenzen und klare Zugriffspunkte, um Ausnahmen, Logging und Zugriffsrechte zu behandeln. Diese Trennung der Daten von der Businesslogik wiederspricht der Idee der objektorientierten Programmierung und wird von OO Puristen kritisiert [Fow].

Die Domänenobjekte dürfen nur innerhalb einer explizit gestarteten Transaktion modifiziert werden. Dies ist notwendig, da jederzeit asynchrone Objektdifferenzen vom Server eintreffen können und die entstehenden Konflikte festgestellt werden müssen.

Als Nachteil ist zu erwähnen, dass wir zwar bezüglich API *Örtlichkeitstransparenz* erreicht haben, die Referenzzugriffe aber von unerwarteten Seiteneffekten wie Netzwerkfehler betroffen sein können.

Lebenszyklus eines Domänenobjektes

In der skizzierten Architektur arbeiten mehrere Clients auf einem lokalen Ausschnitt des Domänenmodells. Wichtig ist, dass diese Ausschnitte mit dem vom Server zentral verwalteten Modell synchron gehalten werden. Um dies zu erreichen, publiziert der Server Differenzobjekte, die Änderungen am Domänenmodell beschreiben. Eine vom Server publizierte Differenz muss von jedem Client angewendet werden, der das entsprechende Objekt bei sich lokal hat. Falls der Client dieses Objekt bereits selber lokal verändert hat, sollen seine Änderungen natürlich nicht einfach

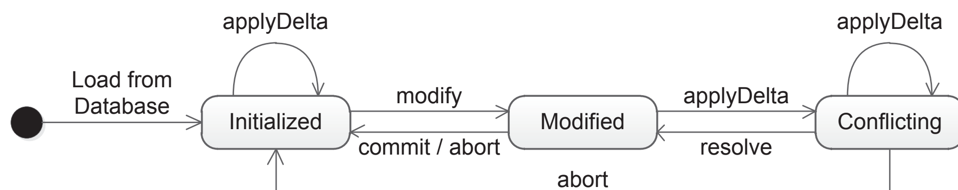

Abbildung 2: Lebenszyklus eines Domänenobjektes

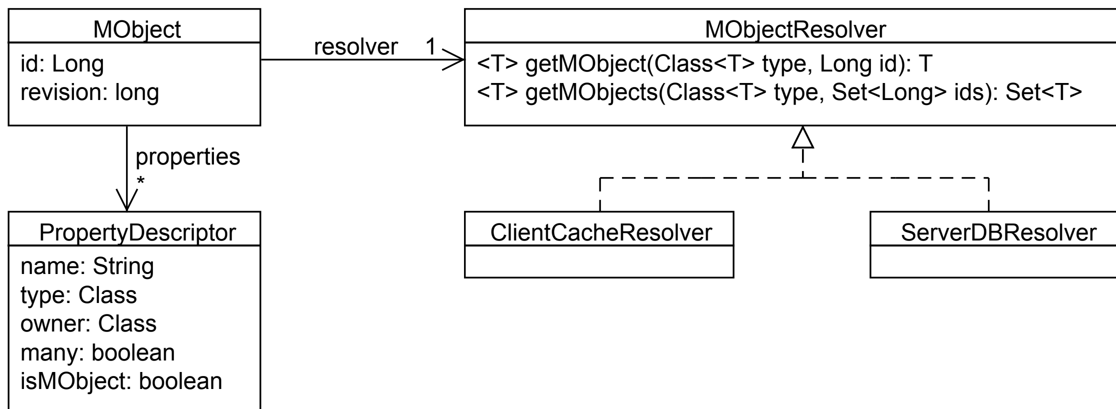

Abbildung 3. MObject und MObjectResolver

überschrieben werden. Daher ist es wichtig, dass Objektveränderungen kontrolliert erfolgen. Dazu unterwerfen wir die Domänenobjekte auf Seite der Clients einem strikten Lebenszyklus, welcher stets konsistente Zustände garantiert (Abb. 2).

Ein frisch aus der Datenbank geladenes Objekt ist im Zustand *Initialized*. In diesem Zustand darf nur lesend zugegriffen werden (Setter werfen in diesem Zustand entsprechende Ausnahmen). Eingehende Differenzen werden direkt übernommen. Das heisst, die Daten und die Versionsnummer des Objektes werden aktualisiert. Dabei bleibt das Objekt im Zustand *Initialized*. Das Objekt ist während der Aktualisierung gelockt und kann weder gelesen noch geschrieben werden.

Mit dem Aufruf der Methode *modify()* wird das Objekt in den Zustand *Modified* gebracht. In diesem Zustand können die Variablen des Objekts via Setter verändert werden. Um die vorgenommenen Änderung für alle Clients verfügbar zu machen, muss das veränderte Objekt über den Persistenzdienst zum Server übertragen werden. Auf dem Server wird zuerst die Differenz berechnet und dann wird das Objekt in einer Datenbank persistiert. Im Falle eines erfolgreichen Datenbank Commits wird die Differenz über eine Messaging-Infrastruktur publiziert, wo jeder Client als Subscriber registriert ist. Der Client, welcher die Persistierung ausgelöst hat, bestätigt die erfolgreiche Speicherung mit einem Commit auf dem lokalen Objekt. Dabei werden die lokal geänderten Daten zur neuen, stabilen Version und das Objekt ist wieder in einem konsistenten Zustand. Mit dem Aufruf von *abort()* können die vorgenommen Änderungen verworfen werden. Die Revision bleibt dabei unverändert.

Es ist nun möglich, dass zwei Clients *A* und *B* zur gleichen Zeit ihre lokale Kopie des gleichen Datenobjektes modifizieren. Nachdem *A* sein verändertes Objekt auf dem Server gespeichert hat und *B* über diese Änderung informiert worden ist, entsteht bei *B* ein Konflikt, da *B* zwei (unterschiedliche) Änderungen für das gleiche Objekt berücksichtigen muss. Deshalb ist im Zustand *Conflicting* speichern nicht erlaubt – was im entsprechenden Service überprüft wird. Stattdessen muss *B* den Konflikt auflösen. Ein solcher Konflikt wird üblicherweise nicht automatisch aufgelöst, sondern von einem Benutzer interaktiv behandelt. Die eingehenden Änderungen werden abgefragt und dem Benutzer in einem Dialog präsentiert, wo dieser entscheiden muss, wie er mit

```
public class MObject {
    Long getId(){...}
    Long getRevision(){...}

    // Generic getters and setters
    <T> T genericSingleGet(PropertyDescriptor<T> propDesc){...}
    <T> void genericSingleSet(PropertyDescriptor<T> propDesc, T newValue){...}

    <T> Set<T> genericManyGet(PropertyDescriptor<T> propDesc){...}
    <T> void genericManySet(PropertyDescriptor<T> propDesc, Set<T> newValue){...}

    // Property change notifications used by databinding
    void addPropertyChangeListener(PropertyChangeListener l) {...}
    void removePropertyChangeListener(PropertyChangeListener l){...}

    // Transactional modification
    IModificationTracker modify(IConflictHandler h) {...}
    IObjectDelta computeDelta(){...}
    void applyDelta(IObjectDelta d) {...}
}
```

Listing 1: Signaturen der öffentlichen Methoden der Klasse MObject

```
<T> T genericSingleGet(PropertyDescriptor<T> propDesc){
  if(propDesc.isMObject()) {
    Long id = idStorage.get(propDesc);
    return resolver.getMObject<T>(propDesc.type, id);
  } else {
    return (T) attributeStorage.get(propDesc);
  }
}

<T> T genericSingleSet(PropertyDescriptor<T> propDesc, T newValue){
  checkAccess();
  if(propDesc.isMObject()) {
    idStorage.put(propDesc, ((MObject)newValue).getID());
  } else {
    attributeStorage.put(propDesc, newValue);
  }
}
```

Listing 2: Implementierung von genericSingleGet() und genericSingleSet()

der Situation umgehen will: Er kann entweder den neuen Wert übernehmen und seine lokale Änderung damit überschreiben oder er kann seine lokale Änderung als aktuellen Wert behalten und damit den neuen Wert überschreiben. Natürlich kann er auch eine neue Eingabe machen. Nach der Konfliktauflösung wird der Konflikt für die entsprechende Variable als gelöst markiert (analog zu „Mark as merged" in SVN) und erst wenn alle Konflikte gelöst sind, wird der Objektzustand wieder auf *Modified* gesetzt. Vom Zustand *Modified* aus kann nun persistiert werden.

Umsetzung in Java

Die vorgestellte Architektur könnte mit unterschiedlichsten Technologien realisiert werden. Aus persönlichen Präferenzen haben wir uns für Java Technologien entschieden. Als Server Laufzeitumgebung wird der auf OSGi basierende *SpringDM Server* eingesetzt [SpDM]. Die Persistenzschicht basiert auf dem *Java Persistence API* und Oracle als Datenbank [JPA]. Synchrone Kommunikation zwischen Client und Server ist mittels RMI realisiert [RMI] und die Differenzdistribution mittels *Java Messaging Service* [JMS] realisiert worden. JMS entkoppelt den Server von den Clients durch Zwischenschaltung eines Topics. Das Topic fungiert als Anschlagsbrett, wo der Server Differenzmeldungen veröffentlicht und damit alle angemeldeten Clients über alle Zustandsänderungen informiert. Als Basis für den Client setzen wir die *Eclipse Rich Client Platform* zusammen mit *Eclipse Databinding* ein [RCP, EDB].

Im Folgenden bezeichnen wir die in diesem System partizipierenden Domänenobjekte als *MObjects* (Kurzform für *Managed Objects*), da sie speziell verwaltet (aktualisiert) werden. Der Name ist zugleich der Bezeichner der gemeinsamen Basisklasse aller Domänenklassen (Abb. 3).

Jedes *MObject* bietet die Funktionalität, die Differenz zwischen dem modifizierten Zustand und der Ausgangsversion zu berechnen. Es ist nun anzustreben, diese Differenzen generisch abzubilden und nicht für jede *MObject*-Klasse eine entsprechende Differenzklasse zu definieren. Analoges gilt für die Berechnung der Differenzen. Diese soll ebenfalls generisch sein und nicht für jede *MObject*-Klasse spezifisch implementiert werden müssen. Dies bedingt, dass die *MObjects* zur Laufzeit inspiziert werden können, d.h. dass ihre Daten aufgelistet und auf die zugewiesenen Werte sowie deren Datentypen zugegriffen werden können.

Die Daten (Attribute und Referenzen) eines *MObjects* werden durch eine Menge von *PropertyDescriptors* beschrieben. Die Klasse *PropertyDescriptor* modelliert eine Eigenschaft in Form eines Namens und Datentyps. Kann die modellierte Eigenschaft nicht nur einen einzigen Wert vom gegebenen Typ, sondern eine ganze Menge solcher Werte aufnehmen, so wird dies mit dem Flag *many* gekennzeichnet.

Die bereits eingeführte *Resolve*-Funktion wird über die Schnittstelle *MObjectResolver* abgebildet. *MObjectResolver* wird von zwei Klassen implementiert. Dem *ClientCacheResolver* und dem *ServerDBResolver*.

Nun betrachten wir die zentrale Klasse *MObject* (Listing 1). Jedes Objekt hat eine eindeutige ID und eine Revisionsnummer, welche die Version eines Objektes identifiziert. Das heisst, wann immer ein Datenobjekt verändert und erfolgreich auf dem Server persistiert wird, so wird auch seine Revisionsnummer inkrementiert. Mittels *genericXGet()* und *genericXSet()* kann auf die Attribute und Referenzen eines beliebigen Objektes

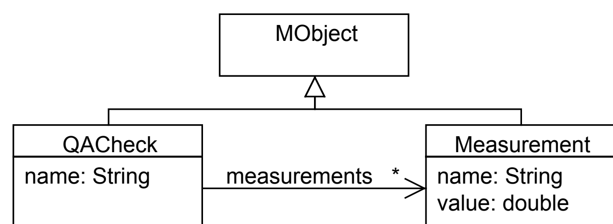

Abbildung 4: Beispiel eines Domänenmodells

```
public class QACheck extends MObject {

  public static final PropertyDescriptor<String> NAME =
   descriptor(QACheck.class, "name", String.class, false)

  public static final PropertyDescriptor <Measurement> MEASUREMENTS =
   descriptor(QACheck.class, "measurements", Measurement.class, true);

  public String getName() {
    return genericSingleGet<String>(NAME);
  }

  public void setName(String newName) {
    genericSingleSet<String>(NAME, newName);
  }

  public Set<Measurement> getMeasurements() {
    return genericManyGet<Measurement>(MEASUREMENTS);
  }

  public void setMeasurements(Set<Measurement> newMeasurements) {
    genericManySet<Measurement>(MEASUREMENTS, newMeasurements);
  }
}
```

Listing 3: QACheck.java

zugegriffen werden (*X* sei ein Platzhalter für *Single* oder *Many*). Diese Methoden werden weiter unten noch genauer besprochen. Beim Aufruf eines Setters, werden alle registrierten *PropertyChangeListeners* benachrichtigt, so dass mittels Databinding die Änderungen auf dem GUI widergespiegelt werden können.

In der Methode *modify()* wird für jedes Objekt eine Transaktion gestartet, bevor es verändert wird. Dies ist notwendig, um mit dem *IConflictHandler* mögliche Konflikt behandeln zu können.

Jedes *MObject* muss die Differenz zwischen seiner Ausgangsrevision und der vollzogenen Änderungen bestimmen können ($\Delta := O_{new} - O_{old}$). Diese Funktionalität wird durch die Methode *computeDelta() implementiert.* Symmetrisch dazu kann eine solche Differenz mit der Methode *applyDelta()* auf ein Objekt angewendet werden, um es auf den Stand der nächst höheren Revision zu bringen ($O_{new} := O_{old} + \Delta$). Solche Differenzobjekte werden vom Server asynchron zur Verfügung gestellt. Es ist daher eine zentrale Aufgabe des Clients, seine Datenobjekte in einem konsistenten Zustand zu halten.

Listing 2 zeigt die Implementierung der beiden Methoden *genericSingleGet()* und *genericSingleSet()*. Wenn in der *get*-Methode die angeforderte Eigenschaft eine Referenz auf ein *MObject* beschreibt, so wird mittels der lokal gespeicherten ID das Objekt über den *MObjectResolver* angefordert. Andernfalls kann der Wert des Attributes direkt zurückgegeben werden. In der *set*-Methode wird als erstes überprüft, ob das Objekt überhaupt verändert werden darf und falls nicht, wirft die Methode *checkAccess()* eine Ausnahme. Wenn es sich um eine Referenz handelt, wird nur die ID des Objektes abgelegt. Andernfalls wird das Attribut selbst abgelegt.

MObjectResolver abstrahiert den Zugriff auf referenzierte *MObjects*. Der *Resolver* auf dem Server verwendet einen JPA *EntityManager*, um aus der Datenbank Referenzen zu laden. Der *Resolver* auf dem Client verwendet einen Objekt-Cache

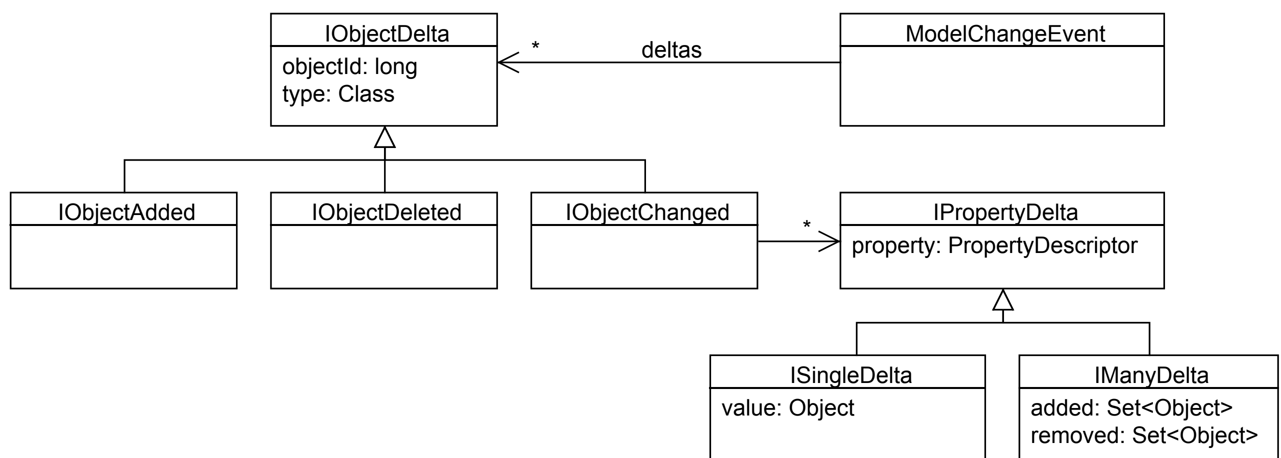

Abbildung 5: Klassendiagramm für Differenzobjekte

und lädt das angeforderte Objekt nur dann vom Server, wenn es nicht bereits lokal vorhanden ist. Die Unifikation über den Objekt-Cache garantiert, dass von jedem Datenobjekt nur eine Instanz pro Client existiert.

Implementierung der Domänenmodellklassen

Zur Illustration der zuvor beschriebenen Konzepte, betrachten wir ein einfaches Domänenmodell (Abb. 4). Wie bereits erwähnt ist *MObject* die gemeinsame Basisklasse der Domänenmodellklassen. Die Klasse *QACheck* modelliert in unserem PSI-Projekt einen Qualitätssicherung-Check mit dem *Attribut name* und der *Referenz measurements*. Die Klasse Measurement beschreibt einen am Protonenbeschleuniger gemessenen Wert. In Listing 3 betrachten wir nun die Implementierung der Klasse *QACheck*.

Die Klasse besteht aus zwei *PropertyDescriptors* und je einem Getter-/Setter-Paar. Die Deskriptoren sind statisch und gelten somit für alle Instanzen dieser Klasse. *Public* sind sie, weil für das Databinding auf die Namen der Properties verwiesen werden muss.

Der erste Deskriptor heisst *NAME* und beschreibt ein *Attribut* der Klasse *QACheck* mit dem Namen „name" vom Typ String. Jeder *QACheck* hat nur einen einzigen Namen, deshalb ist das *many*-Flag auf *false* gesetzt. Der zweite Deskriptor beschreibt die *Referenz* namens „measurements" und ist als *many* gekennzeichnet. Jeder *QACheck* kann entsprechend auf mehrere Messungen verweisen. Die Getter und Setter delegieren die Aufrufe an die zuvor beschriebenen generischen Zugriffsmethoden der Klasse *MObject*.

Der Quellcode der Klasse *QACheck* wirkt etwas aufgebläht und bei vielen Properties kann eine Domänenklasse rasch unübersichtlich und anfällig auf Flüchtigkeitsfehler werden. Daher haben wir im PSI-Projekt die Domänenklassen allesamt aus einem Modell automatisch generiert.

Modellierung von Differenzen

Bisher haben wir gesehen, wie Clients und Server ihre Datenmodelle mithilfe von Differenzobjekten aktuell halten können. In diesem vorletzten Abschnitt gehen wir nun etwas genauer auf die Differenzobjekte ein und erläutern, wie wir diese generisch modelliert haben. Abbildung 5 zeigt ein Klassendiagramm der dabei beteiligten Interfaces.

Objektdifferenzen werden in Form von *IObjectDelta*-Instanzen veröffentlicht und auf Objekte angewendet. Von *IObjectDelta* gibt es drei Ausprägungen:

- *IObjectAdded*: ein neues Objekt ist erzeugt worden;
- *IObjectDeleted*: ein bestehendes Objekt ist gelöscht worden;
- *IObjectChanged*: die Properties eines Objektes sind verändert worden.

Änderungen an einem Objekt beziehen sich entweder auf Attribute oder Referenzen. In beiden Fällen wird mit einem von *IPropertyDelta* abgeleiteten Interface die Änderung beschrieben:

- *ISingleDelta*: Beschreibt Änderungen einer Variablen deren *many* Flag nicht gesetzt ist. Bei einem Attribut beinhaltet es den neuen Wert und bei einer Referenz die ID des neu referenzierten Datenobjektes. Der Typ des referenzierten Objektes ist auf dem *PropertyDescriptor* abrufbar.
- *IManyDelta*: Beschreibt Änderungen einer Variable deren *many* Flag gesetzt ist. Hierfür werden zwei disjunkte Mengen benötigt: Die Menge der hinzugefügten und die Menge der entfernten Elemente. Analog zum *ISingleDelta* beinhalten die beiden Mengen entweder gleich die Werte oder nur die IDs im Fall von Referenzen.

Mit diesem schlichten Modell lassen sich alle Änderungen abbilden, die ein einzelnes Objekt betreffen. Da in einer Transaktion typischerweise mehrere Objekte verändert werden, beinhaltet der publizierte *ModelChangedEvent* eine Menge solcher *IObjectDeltas*.

Fazit

In der vorgestellten Architektur arbeiteten die Clients auf dem automatisch aktuell gehaltenen Domänenmodell. *Refresh Buttons* und DTOs wird man in der realisierten PSI-Software entsprechend keine finden. Natürlich hat diese automatische Synchronisierung auch ihren Preis. Jeder Client wird über jede Objektänderung informiert und es liegt am Client zu prüfen, ob die Differenzmeldungen für ihn relevant sind. Ob dieser Ansatz für eine grosse Anzahl Clients skaliert, wird sich noch zeigen.

Links

[DBind] http://en.wikipedia.org/wiki/UI_data_binding
[DM] http://en.wikipedia.org/wiki/Domain_model
[DTO] http://en.wikipedia.org/wiki/Data_transfer_object
[EDB] http://en.wikipedia.org/wiki/UI_data_binding
[Fow] http://martinfowler.com/bliki/AnemicDomainModel.html
[JMS] http://en.wikipedia.org/wiki/Java_Message_Service
[JPA] http://en.wikipedia.org/wiki/Java_Persistence_API
[PD] http://en.wikipedia.org/wiki/Primitive_data_type
[PSI] http://p-therapie.web.psi.ch/
[PubSub] http://en.wikipedia.org/wiki/Publish/subscribe
[RCP] http://www.eclipse.org/rcp/
[RMI] http://en.wikipedia.org/wiki/Java_remote_method_invocation
[SpDM] http://www.springsource.org/dmserver