# ArgoMobile

Ein ArgoUML-PlugIn für die Modellierung von mobilen Systemen

Christian Kroiß
Ludwig-Maximilians-Universität München

# Inhaltsverzeichnis

# 1 Einleitung

Durch den technologischen Fortschritt der letzten Jahre haben mobile Systeme in der EDV sehr stark an Bedeutung gewonnen. Heute lassen sich bereits sehr komplexe Anwendungen realisieren, an die bis vor kurzem noch niemand gedacht hätte. Dadurch ist es notwendig geworden, bei der Planung und Implementierung von mobilen Systemen methodische und formelle Vorgehensweisen in ähnlichem Umfang anzuwenden, wie bei herkömmlichen Systemen. Allerdings müssen dabei zusätzliche Aspekte berücksichtigt werden. Dieses Dokument beschäftigt sich mit der Modellierung von Vorgängen mit der zusätzlichen Berücksichtigung der Örtlichkeit. In [1] wird ein UML – Profil beschrieben, bei dem Aktivitätsdiagramme um entsprechende Angaben erweitert werden. Grob gesagt kann mit dem Profil sowohl der Standort von Objekten und Aktionen ( d.h. wo eine Aktion stattfindet ), sowie die Änderung von Standorten durch Aktionen modelliert werden.

Im Rahmen dieser Arbeit wurde ArgoMobile entwickelt, ein PlugIn für den Open Source-UML-Editor ArgoUML. ArgoMobile ermöglicht es, die in [1] beschriebenen Erweiterungen anzuwenden und bildet den Schwerpunkt der folgenden Ausführungen. Im ersten Teil wird zunächst das Profil und die damit verbundenen Konzepte vorgestellt. Dabei geht es hauptsächlich darum, die Anforderungen an das PlugIn deutlich zu machen und Begriffe einzuführen, die im Folgenden verwendet werden. Für eine formellere und detailliertere Betrachtung des Profils sei auf [1] verwiesen. Der zweite Teil dieses Dokuments bietet zunächst eine Übersicht über das PlugIn und seine Verwendung. Darauf folgt eine Betrachtung von Architektur und Design von ArgoMobile. Da ArgoUML die Plattform bildet, kommen an dieser Stelle auch allgemeine Aspekte des Programmiermodells von ArgoUML zur Sprache. Alles im allem soll dieses Dokument sowohl ein ausführliches Verständnis für die Funktionsweise von ArgoAgile vermitteln, als auch die Grundlage für weitere Entwicklungen schaffen. Dementsprechend werden im letzten Teil einige Ideen zur Verbesserung sowie mögliche Einsatzmöglichkeiten beschrieben.

# 2 Ein UML – Profil zur Modellierung mobiler Systeme

Bei der Realisierung von mobilen Systemen müssen einige Aspekte betrachtet werden, die bei herkömmlichen Systemen keine Rolle spielen. Bei Aktionen kann es eine große Rolle spielen, an welchem Ort sie stattfinden. Parameter wie die Verfügbarkeit einer Netzwerkverbindung oder die Leistungsfähigkeit eines Gerätes können den Ablauf der Aktion wesentlich beeinflussen. So könnte zum Beispiel das Versenden einer E-Mail-Nachricht automatisch aufgeschoben werden, bis eine Verbindung über einen Mobilfunkkanal wieder verfügbar ist. Letztendlich ist eine Aktion immer gewissen Beschränkungen ( *Constraints* ) unterworfen, die davon abhängen können, an welchem Ort sie ausgeführt wird. Um im Laufe der Implementierung diese *Constraints* berücksichtigen zu können, ist es sinnvoll, die Information über den Ort im Modell festzuhalten. Außerdem können sich die Aufenthaltsorte von Akteuren und mobilen Geräten und somit die Konfiguration des Systems verändern. In [1] wird ein UML –

Profil vorgestellt, das Aktivitätsdiagramme um die Information des Ortes von Objekten und Aktionen erweitert und somit genau die oben genannten Forderungen bedient. In diesem Abschnitt werden zunächst die grundlegenden Konzepte vorgestellt, bevor die Abbildung auf das UML-Profil beschrieben wird.

## 2.1 Grundlegende Konzepte

Die nachfolgenden Begriffe bzw. Konzepte bilden die Basis für das Modellieren der oben beschriebenen Informationen. An dieser Stelle werden sie nur eingeführt. Eine genauere und formelle Darstellung gibt es in [1].

| | ***Beschreibung*** | ***Beispiel*** |
|---|---|---|
| Mobiles Objekt (*mobile object*) | Ein Objekt, dessen Standort durch eine Aktion verändert werden kann. Dies kann sowohl ein physisches Objekt sein, wie ein PDA oder ein Benutzer, als auch eine beliebige Daten-Entität, die zu einem bestimmten Zeitpunkt beispielsweise im Speicher eines mobilen Geräts residiert. | Passagier, PDA, E-Mail |
| Standort (*location*) | Sowohl mobile Objekte, als auch Aktionen haben *immer genau einen* Standort, der explizit angegeben werden kann. Bei jedem Standort kann dabei wiederum angegeben werden, in welchem übergeordneten Standort (*parent location*) er sich befindet. Ein Beispiel wäre eine Stadt A, die in Land B liegt. Auf der anderen Seite kann ein Standort einen oder mehrere Unter-Standorte (*sublocations*) enthalten. Dadurch kann sich eine beliebig komplexe Hierarchie ergeben, wobei jedoch keine Zyklen erlaubt sind, da sich ein Standort nicht in sich selbst befinden kann. | Flughafen |
| Mobiler Standort (*mobile location*) | Ein mobiler Standort (*mobile location*) kann seinen übergeordneten Standort ändern. Dadurch ändert sich auch die Standort-Hierarchie aller Objekte oder Standorte, die sich rekursiv im mobilen Standort befinden. Ein mobiler Standort ist also immer Standort und mobiles Objekt zugleich. | Flugzeug, Passagier (hat PDA), PDA (speichert E-Mail) |
| Move | Eine Move-Aktion verändert den Standort eines mobilen Objekts oder eines mobilen Standorts. | "Ins Flugzeug einsteigen" |
| Clone | Eine Clone-Aktion erzeugt eine Kopie eines mobilen Objekts und transferiert diese in einen anderen Standort. | "Passagierliste zum Bordcomputer transferieren" |

*Tabelle 1: Grundlegende Konzepte*

Ob ein Objekt als mobiles Objekt oder als mobiler Standort beschrieben wird, kann oft abhängig von den Anforderungen des Modells entschieden werden. Ein Passagier kann ein mobiles Objekt sein, falls in einem Modell nur der Transport des Passagiers eine Rolle spielt. Wird zusätzlich z.B. eine E-Mail betrachtet, die auf dem PDA eines Passagiers gespeichert ist, kann der Passagier als mobiler Standort modelliert werden.

## *2.2 Abbildung der Konzepte auf das UML-Metamodell*

Die eingeführten Konzepte müssen sozusagen in die UML "eingebaut" werden. Das erfolgt zunächst durch die Definition des Profils an sich, wie in [1] dargestellt wird. Dieses Kapitel beschreibt knapp die dort getroffenen Vereinbarungen und widmet sich anschließend der Frage, welche zusätzlichen Erweiterungen durchgeführt werden müssen, damit die Unterstützung des Profils in einem Editor realisiert werden kann.

### 2.2.1 Der UML Extension Mechanism

Die Grundlage für das Erstellen eines UML-Profils ist durch den UML Extension Mechanism ( siehe 2.6 in [2] ) gegeben. Ein UML Profil wird definiert, indem bestehende Modellelemente der UML an die Anforderungen des Profils angepasst werden. Dazu wird ein Modellelement zunächst mit einem so genannten *Stereotype* markiert. Dem *Stereotype* können *Constraints* und *Tag Definitions* zugeordnet werden. *Constraints* definieren die eigentliche Semantik mittels einer geeigneten Sprache wie OCL. Durch *Tag Definitions* können einem Modellelement beliebige *Tagged Values* zugeordnet werden. Diese *Tagged Values* können wiederum beliebige Werte enthalten und auch andere Modellelemente referenzieren. Dadurch wird es praktisch möglich, jede Art von Information zu modellieren, solange die ursprüngliche Semantik des Metamodells nicht verletzt wird.

### 2.2.2 Die Definition des Profils

*In [1] wurden die Stereotypes <<mobile>>, <<location>> und <<mobile location>>, sowie <<move>>* und *<<clone>>* eingeführt. Sie entsprechen den in 2.1 beschriebenen Konzepten. Dabei werden die ersten drei zunächst verwendet, um Klassen zu markieren, die mobile Objekte oder ( mobile ) Standorte modellieren. *<<move>>* und *<<clone>>* werden in Aktivitätsdiagrammen eingesetzt und auf *Action States* angewendet. Um die Standortänderung darzustellen, die eine Move- oder Clone-Aktion bewirkt, werden die *Action States* mit eingehenden und ausgehenden *Object Flow States* verknüpft, die mobile Objekte bzw. mobile Standorte repräsentieren. Diese Situation ist in Abbildung 1 dargestellt. Dort ist ein Ausschnitt aus einem Aktivitätsdiagramm in der zuständigkeitsbezogenen Ansicht ( *responsibility centered view )* zu sehen. Genaueres zur Darstellung findet man in 2.3.


*Abbildung 1: Standortänderung durch Move-Aktion*

Die Standortbeziehung wird in [1] im Wesentlichen durch *OCL-Constraints* für die markierten Klassen und ihre Instanzen beschrieben. Jede Klasse, die mit einem der *Stereotypes <<location>>*, *<<mobile>>* oder *<<mobile location>>* markiert ist, muss ein Attribut "*atLoc*" besitzen, dass eine Referenz zu einer Location-Klasse enthalten kann. Durch Constraints wird dabei sichergestellt, dass keine Zyklen in der Standortbeziehung bestehen.

### 2.2.3 Umsetzung in ArgoMobile

Um das beschriebene UML-Profil in einem UML-Editor zu verwenden müssen die

Standortangaben jedoch explizit mit den Mitteln des UML-Metamodells modelliert werden. ArgoMobile verwendet dazu einen Ansatz, bei dem hauptsächlich Wert auf eine einfache Umsetzung gelegt wurde. Dadurch ergeben sich leichte Abweichungen von der ursprünglichen Definition in [1]. Die Frage, ob das gerechtfertigt ist, soll auf eine Diskussion in 4.6 verschoben werden.

Ein *Object Flow State* im Aktivitätsdiagramm kann eine Momentaufnahme eines mobilen Objektes darstellen. Als Typ des *Object Flow State* wird dann ein *Classifier Role* gewählt, dessen Basis von einer Klasse gebildet wird, die mit dem *Stereotype <<mobile>>* markiert ist. Der Standort des mobilen Objekts wird modelliert, indem dem *Object Flow State* der *Tagged Value „atLoc“* hinzugefügt wird. *atLoc* enthält eine Referenz auf einen weitere Object Flow State, der analog so konfiguriert ist, um die Momentaufnahme eines ( mobilen ) Standorts zu repräsentieren. Auf die gleiche Weise kann dem *Object Flow State* des Standorts ein weiterer Standort zugeordnet werden, der dann zur *Parent Location* wird. Bei ActionStates wird der Standort ebenfalls durch den TaggedValue „atLoc“ definiert. Zur Veranschaulichung ist eine solche Situation in abgebildet.

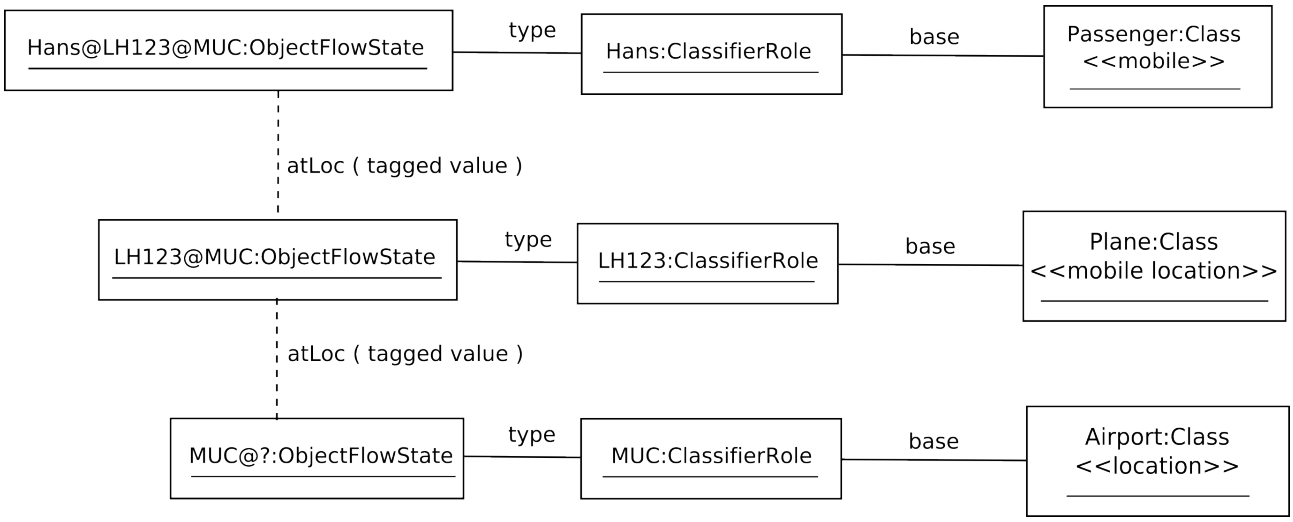



*Abbildung 2: Objektdiagramm - Hans @ LH123 @ MUC*

Wie man sieht, stellen entsprechen die *ClassifierRoles* Instanzen der Klassen, die mobile Objekte oder Standorte modellieren. Allerdings ist dabei die Instanz von ihrem Zustand getrennt, der ja erst durch die Assoziation mit dem *Object Flow State* dargestellt wird.

Aus Sicht der UML-Spezifikation entspricht der hier vorgestellte Ansatz streng genommen nicht ganz der vorgeschlagenen Semantik. Das liegt daran, dass *ClassifierRoles* eigentlich aus dem *Collaborations-Package* stammen und nicht für die Verwendung im Aktivitätsgraphen vorgesehen sind. Warum diese ( aus Spezifikationssicht fehlerhafte ) Implementierung bislang dennoch beibehalten wurde, ist im Rahmen der Diskussion in 4.6 dargestellt.

## *2.3 Darstellung in Aktivitätsdiagrammen*

Um in einem Aktivitätsdiagramm den Standort einer Aktion oder eines Objektes darzustellen, sieht das Profil zwei verschiedene Sichtweisen vor. Diese unterscheiden sich darin, ob der Standort eines Objekts textuell oder grafisch dargestellt wird. Dabei gibt es zu jedem Aktivitätsgraphen jeweils beide Ansichten, die in ArgoMobile

automatisch synchronisiert werden. Um die Unterschiede der beiden Darstellungsweisen sichtbar zu machen, wird in diesem Kapitel ein Beispiel verwendet, das aus [1] übernommen wurde. Es wird dabei folgender Vorgang beschrieben:

1. Der Passagier Hans besteigt am Flughafen München ( MUC ) das Flugzeug mit der Nummer LH123.
2. Das Flugzeug startet.
3. Während des Fluges erstellt Hans eine E-Mail ( auf seinem Smartphone, das allerdings nicht dargestellt wird ) und gibt den Befehl zum Senden der Nachricht. Diese kann allerdings erst übertragen werden, wenn das Flugzeug gelandet ist.
4. Das Flugzeug landet am Flughafen Charles de Gaulle ( CDG ) in Paris.
5. Hans verlässt das Flugzeug.

### 2.3.1 Responsibility Centered View

Bei der *Responsibility Centered View* ( zuständigkeitsbezogenen Ansicht ) wird die gesamte Standort-Information einfach zur Beschriftung des *ObjectFlowStates* hinzugefügt. Die Ansicht unterscheidet sich also nicht wesentlich von der konventioneller Aktivitätsdiagramme. Durch *Swimlanes* kann dargestellt werden, welcher Akteur für Aktionen verantwortlich ist[1]. Abbildung 3 zeigt unser Beispiel in der zuständigkeitsbezogenen Ansicht.

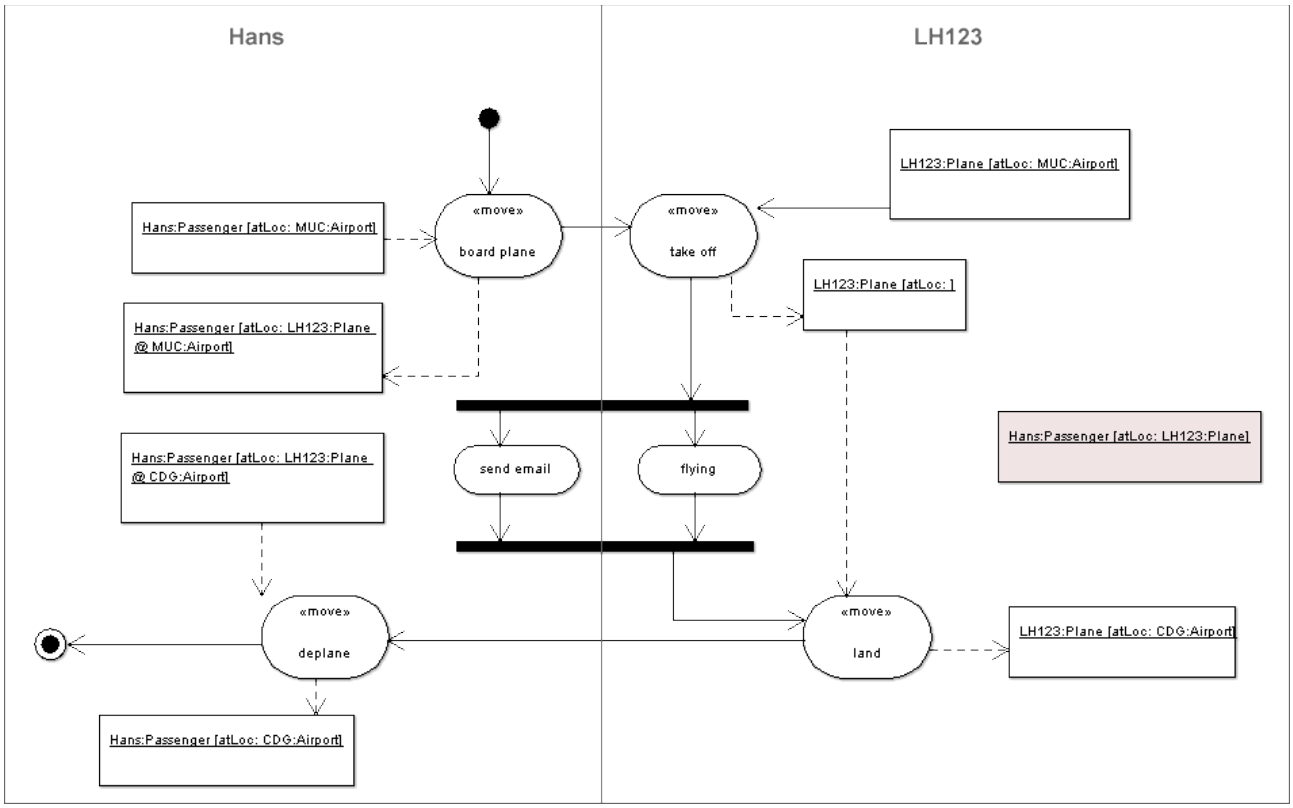



*Abbildung 3: Beispiel in Responsibility Centered View*

1 Swimlanes werden bisher von ArgoUML nicht unterstützt und müssen in einem Grafikprogramm hinzugefügt werden ( siehe auch 4.3 ).

Die Beschriftung der ObjectFlowStates hat den Aufbau N*ame:Klasse:[Standortangabe]*, wobei der Standort durch eine Kette von Standorten angegeben werden kann, die jeweils durch ein „@“ getrennt sind.

### 2.3.2 Location Centered View

Dagegen wird bei der *Location Centered View* die Standortbeziehung geometrisch dadurch dargestellt, dass die Figur eines mobilen Objekts, eines Standorts oder einer Aktion in der Figur eines Standorts vollständig enthalten sind. Wie oben erklärt, werden sowohl Standorte, als auch mobile Objekte im Diagramm durch *ObjectFlowStates* modelliert. Um das Einschließen von Figuren grafisch übersichtlich abbilden zu können, werden in der *Location Centered View* Standort-*ObjectFlowStates* zweigeteilt dargestellt, mit einer abgegrenzten Zeile für den Namen und die Klasse der Instanz, sowie einem Container-Bereich, in dem die Figuren der im Standort enthaltenen Elemente liegen. Bei einem reinen mobilen Objekt reicht einfache Darstellung, die nur Name und Klasse der Instanz darstellt. Abbildung 4 zeigt den selben Aktivitätsgraphen wie Abbildung 3 in standortbezogener Ansicht.

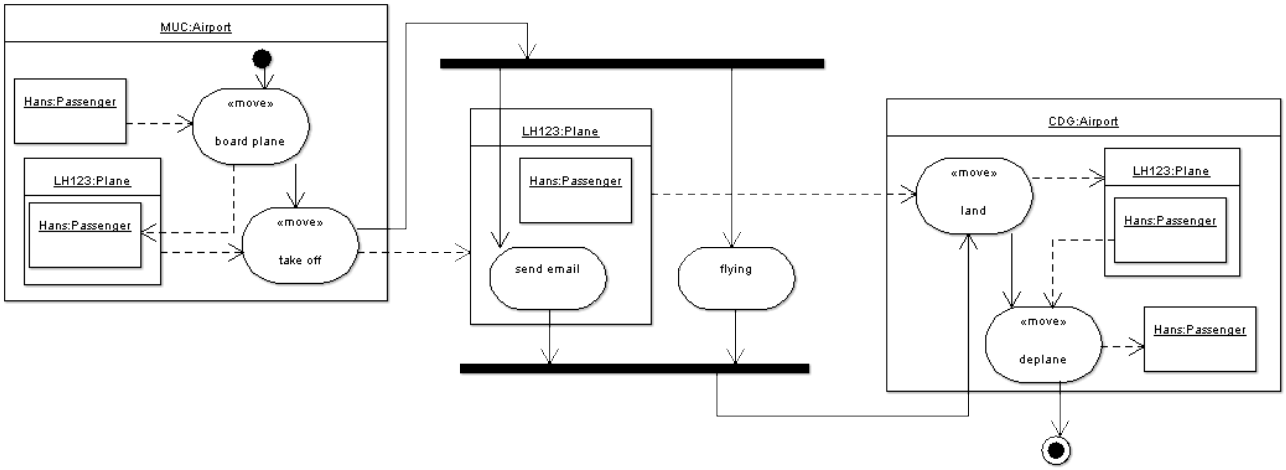



*Abbildung 4: Beispiel in Location Centered View*

Beim Vergleich der Diagramme fällt auf, dass nur in der *Location Centered View* der Standort von Aktionen ersichtlich ist. Auf der anderen Seite geht aus Abbildung 4 nicht wirklich eindeutig hervor, welcher Akteur für eine Aktion verantwortlich ist. Das verdeutlicht den Sinn für die Aufteilung und den engen Zusammenhang zwischen den beiden Ansichten eines Aktivitätsgraphen.

# 3 Das ArgoMobile PlugIn

Nachdem die notwendigen Grundlagen eingeführt sind, wird in diesem Kapitel ArgoMobile vorgestellt. Den Anfang bildet eine Beschreibung der einzelnen Bestandteile des PlugIns und deren Bedienung. Darauf aufbauend wird dann in 3.2 die Implementierung von ArgoMobile betrachtet. Dazu ist es notwendig, sich zumindest kurz der Architektur von ArgoUML zu widmen. Eine genaue Betrachtung würde jedoch den Rahmen sprengen und daher muss für detailliertere Informationen auf die Dokumentation des ArgoUML-Projektes verwiesen werden.

## *3.1 Bedienung*

Für die Beschreibung der einzelnen Funktionen wird in diesem Kapitel wieder das Beispiel aus 2.3 aufgegriffen. Das Vorgehen und die Reihenfolge der Schritte ist dabei typisch für ein Projekt in ArgoMobile.

### 3.1.1 Erstellen des Klassendiagramms

Gestartet wird mit einem leeren Projekt in ArgoUML. Der erste notwendige Schritt ist es, die Klassen zu definieren, die die Basis für die zu modellierenden mobilen Objekte bzw. Standorte bilden. Dazu wird zunächst über den Menüpunkt **Generieren ➤ ArgoMobile ➤ ArgoMobile Klassendiagramm** ein neues ArgoMobile-Klassendiagramm erstellt. Dabei werden gleichzeitig die vom Profil benötigten Stereotypes zum Projekt hinzugefügt. Ein ArgoMobile-Klassendiagramm unterscheidet sich von von einem Standard-Klassendiagramm nur dadurch, dass der Editor einige Funktionen anbietet, der häufige Schritte abkürzt. Die Klassen können in der in ArgoUML üblichen Weise zum Klassendiagramm hinzugefügt und mit dem gewünschten Stereotype versehen werden. Alternativ dazu gibt es im Editor des ArgoMobile-Klassendiagramms drei Schaltflächen in der Toolbar mit denen Klassen erzeugt werden, die bereits mit den Stereotypes mobile, location und mobile location markiert sind. In Abbildung 5 ist ein Screenshot von ArgoUML zu sehen, nachdem ein einfaches Klassendiagramm für das hier verwendete Beispiel erstellt wurde.

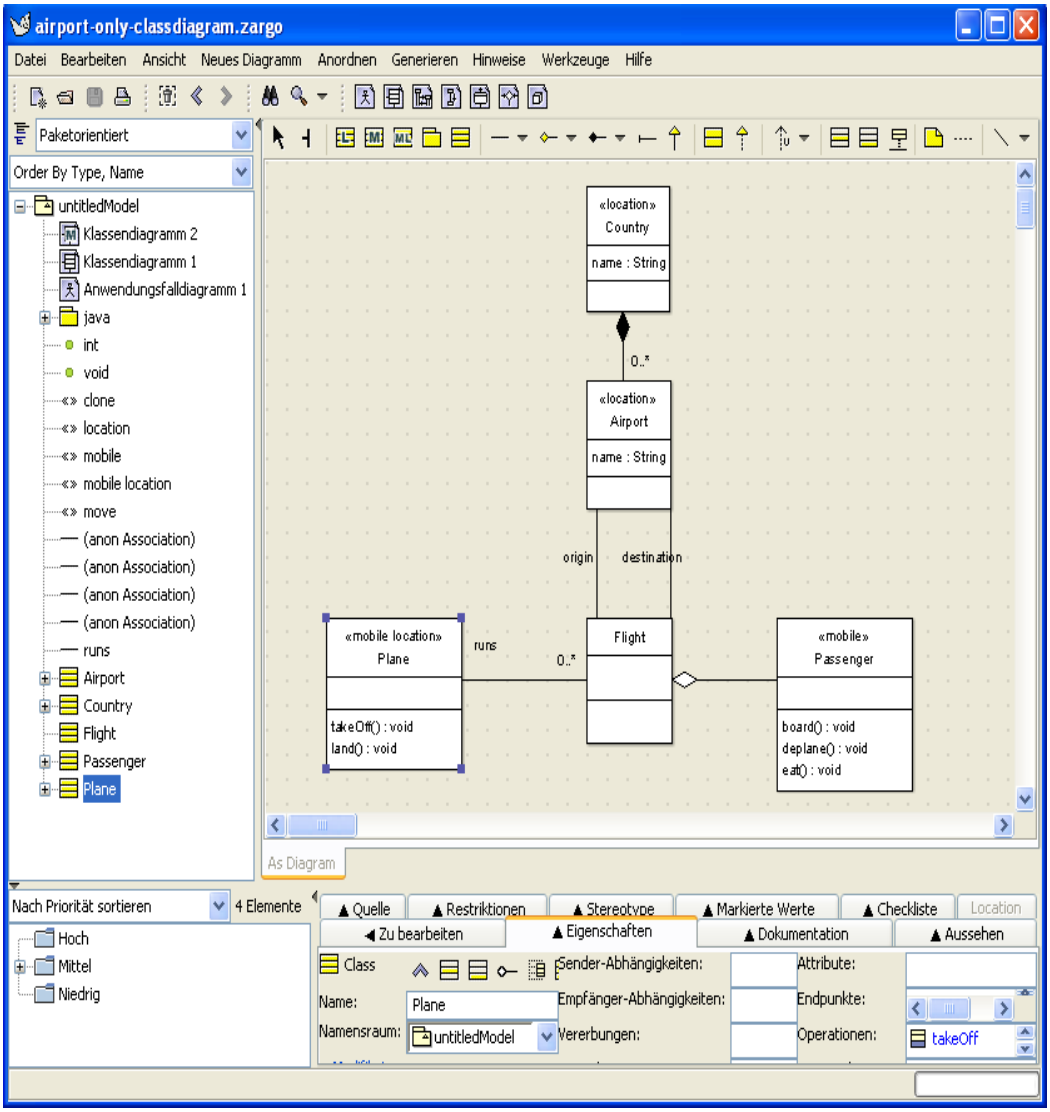



*Abbildung 5: ArgoMobile-Klassendiagramm*

Für das weitere Vorgehen wäre es ausreichend, nur die mit den angegebenen *Stereotypes* markierten Klassen zu erzeugen. Die Attribute, Operationen und Assoziationen im angegebenen Klassendiagramm werden von ArgoMobile nicht speziell behandelt[2].

### 3.1.2 Erstellen der Instanzen

Bevor mobile Objekte oder Standorte in einem Aktivitätsdiagramm modelliert werden können, müssen erst Instanzen der entsprechenden Klassen ins Projekt eingefügt werden. Diese werden, wie in  2.2.3 erklärt, im Modell von ArgoAgile durch *ClassifierRoles* repräsentiert. Dadurch ist es möglich, die Instanzen unabhängig von einem Aktivitätsgraphen einzuführen, also auch wenn noch kein Aktivitätsdiagramm existiert. Dazu markiert man eine Klasse im ArgoMobile-Klassendiagramm und wählt in ihrem Kontext-Menü den Befehl **Erstelle Instanz**. Daraufhin öffnet sich ein einfacher Eingabe-Dialog, indem der Name des konkreten Standorts, bzw. des konkreten mobilen Objekts eingegeben werden kann. Die erstellten Instanzen werden sofort ins Projekt eingefügt. Für unser Beispiel ergibt sich folgendes Bild im ArgoUML- Projekt-

2 Siehe dazu auch 4.4.

Browser:

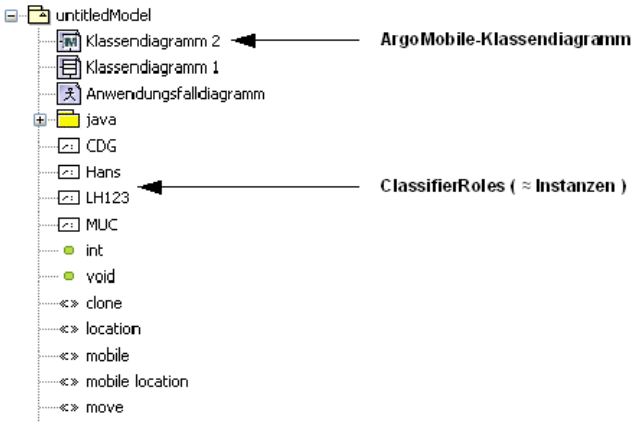



*Abbildung 6: Räumliche Instanzen im Projekt-Browser*

Eine komfortablere Alternative zum Erstellen und Bearbeiten von räumlichen Objekten bietet jedoch der *Instanzeditor* von ArgoMobile, der im Menü unter **Werkzeuge➤ ArgoMobile➤Bearbeite räumliche Instanzen** gefunden werden kann.

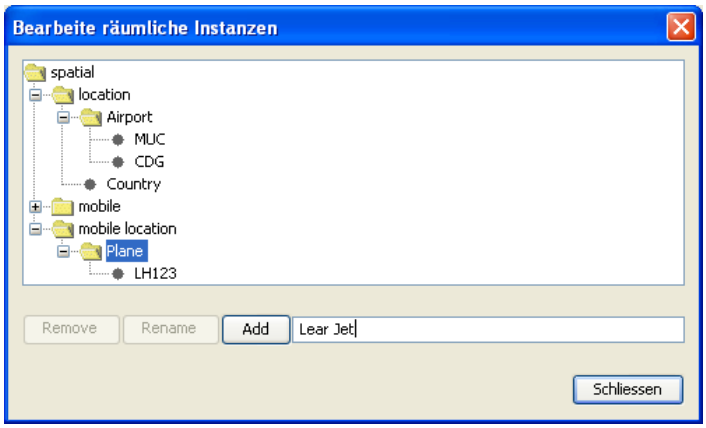



*Abbildung 7: Instanzeditor*

Der Instanzeditor ist in Abbildung 7 abgebildet. Er enthält im oberen Teil eine Baumansicht mit der Struktur Stereotype→Klasse→ClassifierRole. Darunter befinden sich Knöpfe zum Hinzufügen, Löschen und Umbenennen von Instanzen, sowie ein Textfeld. Damit eine Instanz erstellt werden kann, muss in der Baumansicht die entsprechende Klasse selektiert sein und das Textfeld muss den Namen der Instanz enthalten. Dabei darf der Name nicht bereits von einer Instanz derselben Klasse verwendet werden. Gelöscht werden kann eine *ClassifierRole* nur dann, wenn sie in keinem Aktivitätsdiagramm verwendet wird. Die Schaltfläche „Löschen“ ist daher gegebenenfalls deaktiviert. Das Umbenennen einer Instanz ist dagegen jederzeit möglich. Allerdings sollte man berücksichtigen, dass sich ein längerer Name auf das Layout eines bestehenden Aktivitätsdiagramms auswirken kann, da die Figuren der *ObjectFlowStates* eventuell ihre Größe ändern.

### 3.1.3 Erstellen der Aktivitätsdiagramme

Sobald die benötigten Klassen im Modell vorhanden und mit den richtigen Stereotypes markiert sind, können nun gemäß 2.3 ein Aktivitätsgraph und die beiden

dazugehörigen Ansichten erstellt werden. Dazu selektiert man im Projekt-Browser eine Klasse aus, die eines der räumlichen Stereotypes trägt und wählt den Menüpunkt **Neues Diagramm➤ArgoMobile➤MobileUML-Aktivitätsdiagramm-Paar**. Dadurch ein neuer Aktivitätsgraph erstellt und der ausgewählten Klasse zugeordnet. Der Aktivitätsgraph enthält zwei Aktivitätsdiagramme, eines für die *Location Centered View* und eines für die *Responsibility Centered View.* Die beiden Diagramme sind dabei tatsächlich *Ansichten desselben* Aktivitätsgraphen. Änderungen im Modell, die in einer Ansicht vorgenommen werden, übertragen sich automatisch auf die andere Ansicht. Das gilt jedoch nicht für Änderungen am Layout wie das Verschieben von Objekten. Das Editieren der beiden neuen Diagrammtypen unterscheidet sich nicht wesentlich von der Arbeit mit herkömmlichen Aktivitätsdiagrammen. Die Besonderheiten kommen in den beiden folgenden Abschnitten zur Sprache.

*ActionStates* können einfach über das Kontext-Menü mit den Stereotypes <<*move*>> oder <<*clone*>> versehen werden, was, anders als bei normalen Aktivitätsdiagrammen in ArgoUML bei normalen  . Anders als in normalen Aktivitätsdiagrammen wird der Stereotype auch für *ActionStates* dargestellt. Ein *ObjectFlowState*, der ein mobiles Objekt oder einen Standort darstellen soll, wird zunächst wie gewohnt zu einem Diagramm hinzugefügt. Anschließend wird im Karteireiter „Eigenschaften“ die gewünschte Instanz ( also die *ClassifierRole* ) als Typ gewählt. Der *ObjectFlowState* stellt jetzt eine Momentaufnahme der Instanz da. Die Darstellung hängt laut in den beiden Ansichten wurde bereits in 2.3 dargestellt. In den folgenden Abschnitten wird nun erklärt, wie die Standortangaben für die Objekte bzw. Aktionen hinzugefügt und bearbeitet werden können. Vorher soll jedoch zur betrachtet werden, wie sich das eben beschriebene beim Beispielprojekt in ArgoUML darstellt. Auf dem Screenshot in Abbildung 8 ist  der vollständige Aktivitätsgraph des Beispielprojekts dargestellt, wobei die *Location Centered View* aktiv ist. Man erkennt, dass es im Unterschied zu normalen Aktivitätsdiagrammen, im Projekt zwei Diagramme für einen Aktivitätsgraphen gibt. Die Oberfläche des Diagramm-Fensters unterscheidet sich dagegen gar nicht von der herkömmlichen. Das ist genau so auch in der *Responsibility Centered View* der Fall, weswegen sie hier nicht extra gezeigt wird.

### *Location Centered View*

Beim Editieren der Aktivitätsdiagramme ist es ratsam, sich zunächst auf die *Location Centered View* zu konzentrieren, da dort die Grundstruktur des Ablaufs wesentlich leichter festgelegt werden kann. In 2.3 wurde beschrieben, dass ein Standort in der *Location Centered View* durch einen *ObjectFlowState* dargestellt wird, dessen Figur die Figuren anderer Objekte oder Aktionen enthalten kann. Dementsprechend kann eine Standortangabe erzeugt werden, indem die Figur eines *ObjectFlowStates* oder eines *ActionStates* auf der Arbeitsfläche in die Figur eines Standort-*ObjectFlowState* geschoben wird. Die Figur des Standorts wird nun zum *Container*, was bedeutet, dass die beiden Figuren ab jetzt zusammen verschoben werden können. Auf die gleiche Weise kann auch ein die Figur eines ( mobilen ) Standorts in die eines anderen verschoben werden. Dadurch können praktisch beliebig verschachtelte Standort-Angaben entstehen. Auch lässt sich so der Standort von *ActionStates* angeben.

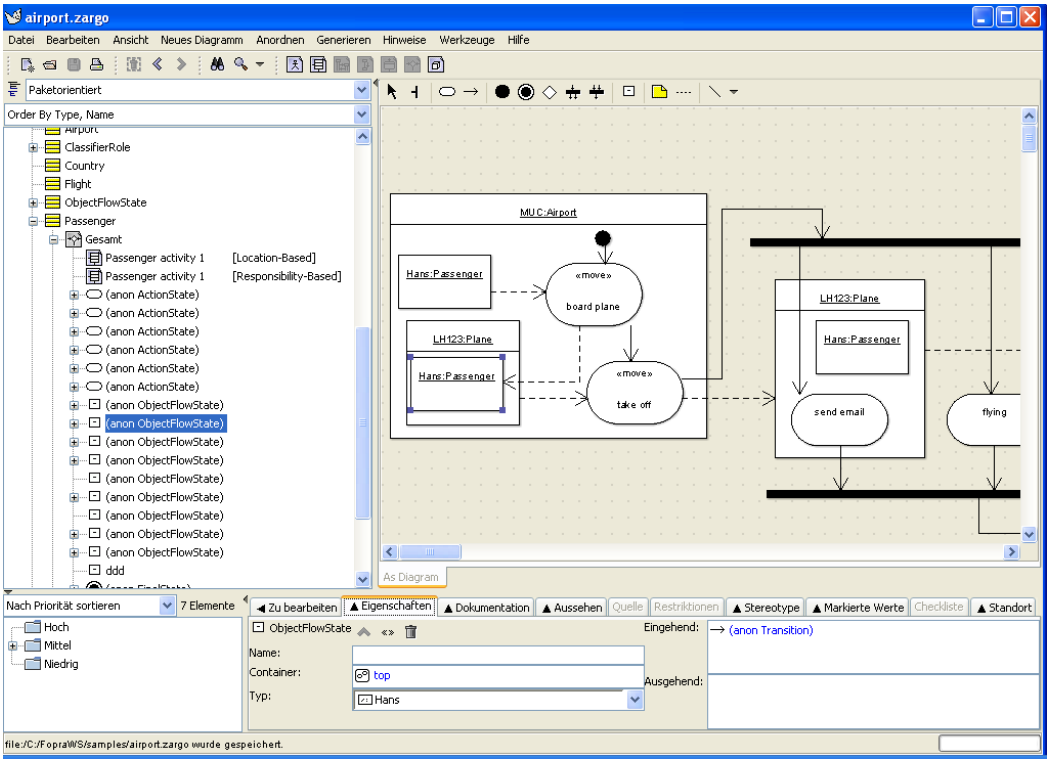



*Abbildung 8: Screenshot Location Centered View*

### *Synchronisierung*

Um die beiden Ansichten konsistent zu halten, gibt es in ArgoMobile einen Synchronisations-Mechanismus, der Änderungen am Modell in die jeweils inaktive Ansicht überträgt. Das betrifft das Hinzufügen und das Löschen von Modellelementen und Transitionen, nicht jedoch Änderungen am Layout, wie das Verschieben einer Figur. Beim automatischen Platzieren von Objekten in der inaktiven Ansicht werden in der jetzigen Version einfach die Koordinaten aus der aktiven Ansicht übernommen. Falls dadurch eine Überlappung entsteht, wird die neue Figur einfach an die nächste freie Stelle verschoben. Dadurch kann die inaktive Ansicht schon nach relativ wenigen Änderungen unübersichtlich werden. Das Layout der jeweils anderen Ansicht sollte also in kurzen Abständen korrigiert werden, da es sonst auch für den Autor schwierig werden kann, sich im Diagramm zurechtzufinden.

### *Responsibility Centered View*

Wie schon gesagt wird in der zuständigkeitsbezogenen Ansicht der Standort eines mobilen Objekts nicht durch eine eigene Figur dargestellt. Deshalb wird die Figur des ObjectFlowStates gelöscht, sobald sein Typ auf eine *ClassifierRole* einer reinen Standort-Klasse ( mit <<*location*>> markiert ) gesetzt wird. Bei mobilen Standorten hängt es von ihrer Rolle im Modell ab, ob sie dargestellt werden oder nicht. ArgoMobile entfernt den *ObjectFlowState* eines mobile Standorts genau dann aus dem Diagramm, wenn er mit keinem *ActionState* verknüpft ist und der Standort keine mobilen Objekte bzw. Standorte enthält[3].

3 Dieses automatische Verhalten kann problematisch sein. Siehe dazu 4.2

### 3.1.4 Standort-Editor

In der *Responsibility Centered View* ist es nicht möglich, eine Standortangabe zu editieren. Dazu kann jedoch der Standort-Editor benutzt werden. Dieser befindet sich auf dem Karteireiter „Standort“ in der Leiste am unteren Fensterrand. Der Karteireiter lässt sich erst dann aufklappen, wenn ein *ObjectFlowState* angewählt wird, dessen Typ ein (mobiler) Standort oder ein mobiles Objekt ist. In Abbildung 9 ist der Standort-Editor für einen Fall aus dem Beispielprojekt zu sehen. Es wurde dabei ein ObjectFlowState „Hans“ ausgewählt mit der Konfiguration „hält sich im Flugzeug LH123 auf, das auf dem Flughafen MUC steht“.

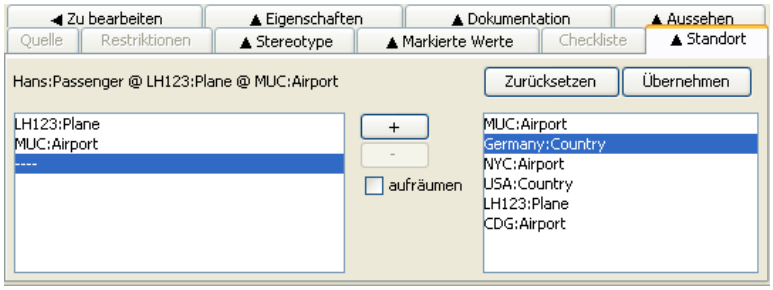



*Abbildung 9: Standort-Editor*

Die Liste im linken Teil der Ansicht zeigt die einzelnen Glieder der Standort-Kette. Auf der rechten Seite sind die verfügbaren Standort-Instanzen aufgelistet. Dazwischen gibt es zwei Schaltflächen, die mit einem + bzw. - beschriftet sind. Der Standort des mobilen Objekts kann editiert werden, indem Standort-Instanzen aus der linken Liste entfernt oder zur linken Liste hinzugefügt werden. Mit „+“ wird die Standort-Instanz, die in der rechten Liste markiert ist, an der Stelle über dem in der linken Liste markierten Eintrag eingefügt. Drücken von „-“ entfernt den markierte Eintrag der linken Liste. Die Schaltfläche „Übernehmen“ macht die Änderungen wirksam. Dadurch werden alle fehlenden *ObjectFlowStates* für die einzelnen Standorte erzeugt und zu den Ansichten hinzugefügt. In der *Location Centered View* erzeugt ArgoMobile dabei eine korrekte und möglichst übersichtliche geometrische Anordnung. Das beinhaltet auch, dass Figuren eventuell vergrößert oder verschoben werden. Dabei wird versucht, das bestehende Layout so wenig wie möglich zu verändern. Eine genauere Erklärung folgt in 3.2.8.

### 3.1.5 Hinweise

Ein wichtiges Feature von ArgoUML sind die so genannten *Design Critics* oder Hinweise. Während der Arbeit an einem Projekt werden dabei periodisch im Hintergrund verschiedene Aspekte des Designs überprüft. Wird dabei ein Problem entdeckt, so erscheint ein entsprechender Eintrag in der Hinweise-Ansicht in der linken unteren Ecke des Fensters. Sobald das Problem behoben wurde, verschwindet er wieder. Durch das Selektieren eines Eintrags kann sich der Benutzer auf dem Karteireiter „Zu bearbeiten“ eine nähere Beschreibung des Hinweises anzeigen lassen. Außerdem wird gegebenenfalls das betroffene Modellelement im Diagramm hervorgehoben. Bei einigen Hinweisen gibt es zusätzlich einen *Wizard* auf dem „Zu Bearbeiten“-Karteireiter, der beim Beheben des Problems hilft. Beispiele für Hinweise, die ArgoUML standardmäßig anbietet sind:

- „Definieren Sie eine konkrete (Unter-)Klasse“ ( wird bei abstrakten Klassen angezeigt, von der keine konkrete Klasse abgeleitet wurde ).

- „Definieren Sie Methoden für die Klasse XY“
- „Doppelt vorkommende Aggregation“ ( wird angezeigt, wenn beide Funktionen einer Assoziation vom Typ Aggregation oder Komposition sind ).

Wie man sieht gibt es dabei sowohl Hinweise, die auf Fehler im Design hinweisen wie das das dritte Beispiel, wie auch solche, die den Benutzer einfach an einen nächsten Schritt erinnern sollen. Folgerichtig wird jedem Hinweis eine der Prioritäten „Hoch“, „Mittel“ oder „Niedrig“ zugeordnet.

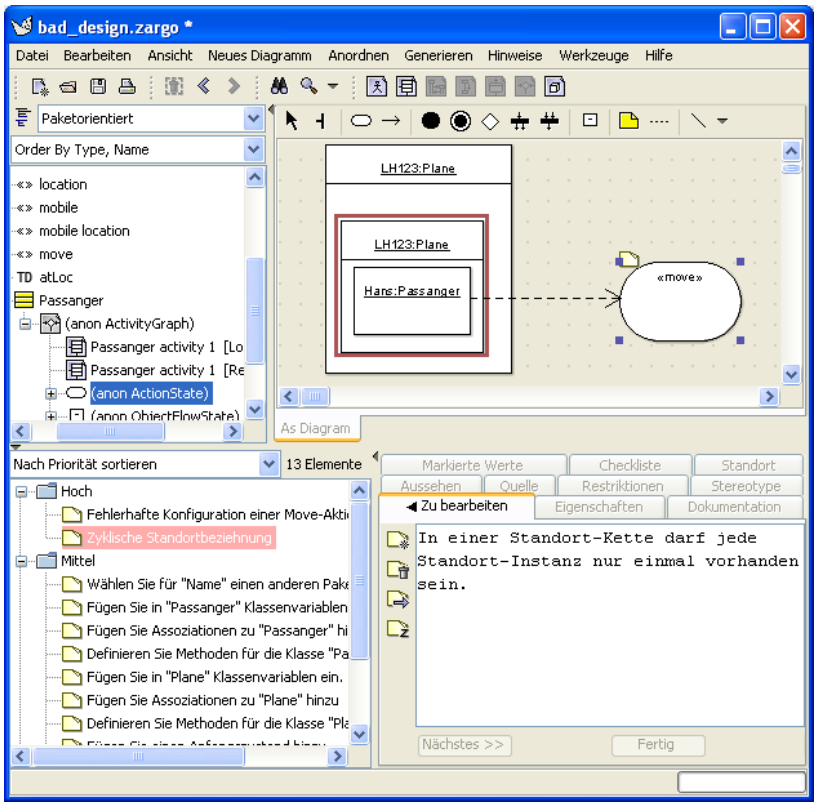


*Abbildung 10: Hinweise*

In Abbildung 10 ist links unten die Hinweis-Liste zu sehen. Ein Eintrag mit der Beschriftung „Zyklische Standortbeziehung“ ist selektiert. Dieser Design Critic weist darauf hin, dass im oben angezeigten Diagramm der obere *ObjectFlowState* „LH123“ in einem *ObjectFlowState* des gleichen Typs enthalten ist. Dies stellt einen schwerwiegenden Fehler da, da sich ein Standort definitionsgemäß nicht in sich selbst aufhalten kann. Rechts neben der Listenansicht ist der Karteireiter „Zu bearbeiten“ geöffnet, der einen Kurzen Erklärungstext enthält. Im Diagramm ist die Figur des betroffenen oberen ObjectFlowState durch ein Rotes Rechteck markiert.

Über den Menüpunkt **Hinweise➤Hinweise anpassen** kann konfiguriert werden, welche Regeln in die Überprüfung miteinbezogen werden und welche Priorität ihnen zugeordnet wird. Bei der Arbeit mit ArgoMobile kann es nützlich sein, bestimmte Hinweise auszuschalten. Zum Beispiel gibt es den Hinweis *„Zustandsübergänge zu XY hinzufügen“*. Ist dieser aktiv, wird auch für die *ObjectFlowStates* von Standorten bemängelt, dass sie nicht mit Aktionen verknüpft sind. Dies ist jedoch für nicht mobile Standorte der Normalfall.

Neben dem oben vorgestellten Hinweis „Zyklische Standortbeziehung“ enthält ArgoMobile noch einige andere, die speziell auf das verwendete UML-Profil eingehen.

Dabei basieren die verwendeten Regeln zum großen Teil auf die Constraints, die in [1] eingeführt wurden. Um sie verwenden zu können, müssen die ArgoMobile-Hinweisklassen erst im Dialog **Hinweise anpassen** aktiviert werden. Ihre Wirkungsweise wird in den folgenden Abschnitten beschrieben.

### *Zyklische Standortangabe*

**Quelle:** ObjectFlowState, das einen (mobilen) Standort referenziert.

Die strengste Bedingung für die Konsistenz von Modellen nach dem vorgestellten Profil ist, dass es keine zyklisch verschachtelten Standort-Ketten geben kann. Das heißt im Klartext, dass ein Standort nicht in sich selbst enthalten sein darf. Ist dies doch der Fall, wird die Standortangabe schlichtweg ungültig. Der Hinweis „Zyklische Standortangabe“ überprüft das Einhalten dieser Regel für alle *ObjectFlowStates*, die mobile Objekte oder (mobile) Standorte repräsentieren. Ein Problem wird dann gefunden, wenn die Standortangabe einen *ObjectFlowState* enthält, der den selben *Typ* hat.

### *Move-Aktion/Clone-Aktion ohne mobile Objekte*

**Quelle:** *ActionState* mit *Stereotype* *<<move>>* oder *<<clone>>*

Eine Move- oder Clone-Aktion verändert den Standort einer räumlichen Instanz. Daher macht ein mit *<<move>>* oder *<<clone>>* markierter *ActionState* nur dann Sinn, wenn er mit mindestens einem *ObjectFlowState* verbunden ist, der ein mobiles Objekt bzw. einen mobilen Standort repräsentiert.

### *Keine Entsprechung zu ein-/ausgehendem ObjectFlowState bei Move-Aktion*

**Quelle:** *ObjectFlowState*, der ein mobiles Objekt oder einen mobilen Standort referenziert.

Ist ein *ObjectFlowState*, der ein mobiles Objekt oder einen mobilen Standort repräsentiert, auf der eingehenden Seite mit einer Move-Aktion verbunden, dann muss auf der ausgehenden Seite der Aktion ein *ObjectFlowState* mit der Selben *ClassifierRole* als Typ vorhanden sein, der also die selbe räumliche Instanz repräsentiert. Dasselbe gilt für den umgekehrten Fall, wenn der entsprechende eingehende *ObjectFlowState* fehlt.

### *Keine Entsprechung zu ein-/ausgehendem ObjectFlowState bei Clone-Aktion*

**Quelle:** *ObjectFlowState*, der ein mobiles Objekt oder einen mobilen Standort referenziert.

Anders als bei der Move-Aktion muss bei der Clone-Aktion ein *ObjectFlowState* als Pendant vorhanden sein, der zwar dieselbe Klasse, jedoch nicht dieselbe Instanz repräsentiert. Wieder gilt die Gegenrichtung genauso. Zur Veranschaulichung ist in Tabelle 2 ein Beispiel für eine Clone-Aktion gegeben: die Datei „Passenger Log“ wird vom „File Server“ auf den „On-Board Computer“ kopiert. Die Clone-Aktion dabei gibt an, dass beim Kopieren eine *andere Instanz* der Datei auf den Bordcomputer verschoben wird. Die Datei auf dem File Server bleibt erhalten.

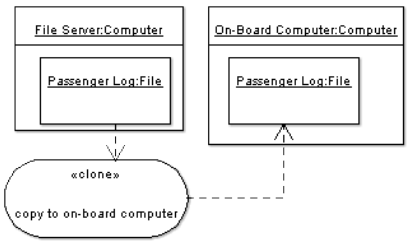
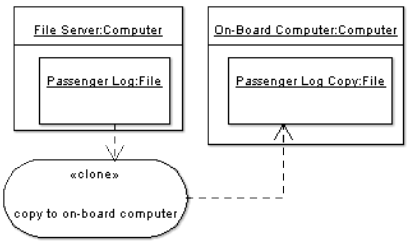
| ***Fehlerhafte Clone-Aktion*** | ***Gültige Clone-Aktion*** |
| --- | --- |
|  |  |

*Tabelle 2: Fehlerhafte Clone-Aktion*

#### *Keine Änderung des Standorts durch Move-Aktion*

**Quelle:** *ObjectFlowState*, der ein mobiles Objekt oder einen mobilen Standort repräsentiert.

Bei einer Move-Aktion sollten sich der Standort eines eingehenden mobilen Objekts oder Standorts ändern. Das bedeutet, dass sich die Standortangabe des entsprechenden ObjectFlowStates auf der ausgehenden Seite unterscheiden muss.

Bei einer Clone-Aktion wird die eben beschriebene Tatsache nicht als Problem angesehen, da es unter Umständen beabsichtigt ist, bloß eine Kopie des eingehenden Objektes zu erstellen.

#### *Keine Standortangabe für mobiles Objekt*

**Quelle:** *ObjectFlowState*, das ein mobiles Objekt referenziert.

Bei einem mobilen Objekt sollte immer ein Standort angegeben werden. Andernfalls sind die räumlichen Auswirkung der Aktionen im Diagramm nicht vollständig ersichtlich.

#### *Layout überprüfen*

**Quelle:** Eine Ansicht eines MobileUML-Aktivitätsgraphen

Wird ein Modellelement in einer Ansicht eines MobileUML-Aktivitätsgraphen hinzugefügt oder gelöscht, dann wird diese Änderung bei der Synchronisierung auch in der jeweils anderen Perspektive sichtbar gemacht. Wie in 3.1.3 beschrieben, muss dabei die Platzierung meist von Hand korrigiert werden. Das gleiche Problem ergibt sich beim automatischen Einfügen von Figuren durch den Standort-Editor ( siehe 3.1.4). Der Hinweis „Layout überprüfen“ erinnert den Benutzer daran, dass das Layout einer Ansicht kontrolliert werden muss. Er wird bei jeder automatischen Änderung an einem Diagramm aktiv und wird entfernt, sobald der Benutzer das Diagramm aktiviert, also anzeigt. Da der Hinweis mit dem Diagramm verknüpft ist, reicht es, auf seinen Eintrag in der Hinweis-Liste zu klicken, um zu dem Diagramm zu springen.

## *3.2 Implementierung*

Die Bedienung des PlugIns ArgoMobile sollte bis hierhin ausreichend beschrieben sein. Darauf aufbauend bietet dieses Kapitel einen Überblick über seine Implementierung. Dazu muss zunächst zumindest in Ansätzen die Architektur von ArgoUML vorgestellt werden. Darauf folgt eine Betrachtung der wichtigsten Aspekte bei der Implementierung von ArgoMobile. Im Vordergrund Stehen dabei grundlegende

Problemstellungen und Ansätze, die sich aus den Anforderungen an das PlugIn ergeben. Auf die konkrete Realisierung der aufgeführten Punkte kann oft nicht detailliert eingegangen werden, da sich an vielen Stellen durch die Beschaffenheit von ArgoUML sehr komplexe Lösungswege ergeben. Auch sind einige Punkte noch keineswegs zufriedenstellend gelöst, worauf in Kapitel 4 näher eingegangen wird.

### 3.2.1 Die Architektur von ArgoUML

Im ArgoUML Cookbook [3] werden 20 Subsysteme von ArgoUML aufgezählt, die in die Kategorien „Low Level“, „Model“, „View and Control“ und „Loadable Subsystems“ eingeteilt werden. Für unsere Betrachtungen spielen dabei nur einige wenige eine Rolle, die im folgenden beschrieben werden.

#### *Modell*

Das Modell von ArgoUML muss in der Lage sein, das UML-Metamodell auf Java-Klassen abzubilden. Die Basis dafür bietet die Technologie *Java Metadata Interface (JMI)* [4], die wiederum auf der *MetaObject Facility (MOF)* [5] aufbaut. Dabei beschreibt *MOF* allgemein eine Methode, um Metamodelle zu erstellen und zu speichern. *JMI* ist eine Spezifikation, die im Wesentlichen die Abbildung von *MOF* auf Java beschreibt. In den früheren Versionen von ArgoUML kam als Implementation von *JMI* die Bibliothek *NSUML* [6] zum Einsatz. Diese wurde jedoch mittlerweile durch das modernere *Metadata Repository (MDR) [7]* abgelöst, das UML 1.4 unterstützt. Generell besteht der Kern des Modells aus einer Sammlung von Interfaces, die gemäß der *JMI-* und *MOF-* Spezifikationen anhand des UML-Metamodells generiert wurden.

Im Rahmen der Umstellung auf *MDR* wurde Wert darauf gelegt, eine möglichst lose Kopplung zwischen Modell und Applikation zu schaffen, um die Implementation gegebenenfalls austauschen zu können. Dazu wurden strukturelle Entwurfsmuster wie *Strategy* und *AbstractFactory* verwendet, um eine Abstraktion von der konkreten Modell-Implementation zu erzielen. Als Beispiel für diesen Aufbau ist in Abbildung 11 ein sehr vereinfachter Ausschnitt aus dem Modell-Subsystem zu sehen.

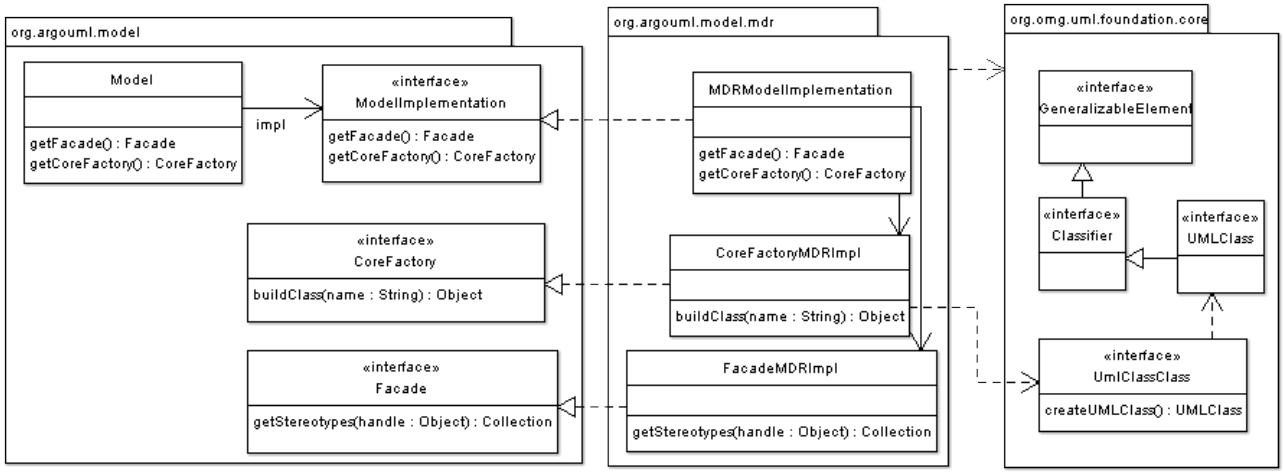



*Abbildung 11: Modell*

Insgesamt stellt die Klasse `Model` den einzigen Zugriffspunk für die gesamte Applikation da. Das Erstellen, Manipulieren und Analysieren von Modellelementen

geschieht ausschließlich durch *Factory-* und *Helper-Interfaces,* wie z.B. `CoreFactory` im obigen Diagramm sowie die über `Facade`.

### *Diagramme*

Für die Darstellung der UML-Diagramme verwendet ArgoUML das *Graphical Editing Framework (GEF)*. Generell wird dabei ein Diagramm als Graph behandelt. Für jeden Knoten und jede Kante existiert ein Eintrag im so genannten *GraphModel*. Im Fall von ArgoUML unter verwendung von *MDR* sind das Instanzen der oben erwähnten Metamodell-Interfaces wie `UMLClass`. Durch einen *Renderer* wird für jedes Modellelement eine Figur erzeugt, gegeben durch eine Unterklasse von `Fig`. Diese wird auf dem *Layer* des Diagramms positioniert.

### *Event-Handling*

In ArgoUML gibt es einen weitreichenden Mechanismus, der die Subsysteme und Module über Ereignisse informiert. ArgoMobile verwendet im Wesentlichen drei Arten von Ereignissen:

- `TargetEvents` zeigen an, wenn sich die aktuelle Auswahl ändert. Dabei können sowohl Elemente im Diagramm, als auch in der Projekt-Ansicht ausgewählt werden. Um `TargetEvents` zu erhalten, muss ein `TargetListener` im `TargetManager` registriert werden. Der `TargetManager` ist als *Singleton* implementiert und bietet zusätzlich Methoden an, um auf die aktuelle Auswahl zuzugreifen.
- Modell-Ereignisse werden bei Änderungen des Modells ausgelöst. Die Weiterleitung erfolgt über ein weiteres *Singleton*, die Klasse `ModelEventPump`. Dabei werden, entsprechend der oben beschriebenen Abstraktion von einer konkreten Realisierung des Modells, die implementations-spezifischen Ereignisse in `PropertyChangeEvents` umgesetzt. Beim Registrieren eines `PropertyChangeListener` in der `ModelEventPump` kann angegeben werden, ob Ereignisse für ein Bestimmtes Modellelement abgefangen werden sollen, oder für alle Modellelemente einer bestimmten Klasse. Außerdem kann eingestellt werden, dass eine Benachrichtigung nur erfolgen soll, wenn ein spezielles Attribut eines Modellelements geändert wurde[4].
- *GraphModelEvents* werden beim Hinzufügen oder Löschen Knoten und Kanten im GraphModel eines Diagramms ausgelöst. Diese Art von Ereignissen werden hauptsächlich direkt in den Diagramm-Klassen bearbeitet.

### *PlugIns*

ArgoUML bietet einen einfachen PlugIn-Mechanismus an, der von ArgoMobile verwendet wird. Ein Plugin muss in eine JAR-Datei verpackt werden, deren Manifest die Spezifikation „ArgoUML Dynamic Load Module“ enthält. Unten ist das Manifest von ArgoMobile angegeben.

```
Name: de/lmu/agile/argoagile/ArgoMobilePlugin.class
Extension-name: module.argomobile.plugin
Specification-Title: ArgoUML Dynamic Load Module
Specification-Version: 0.9.4
Specification-Vendor: University of California
```

4 Es können dabei alle Attribute aus dem Metamodell angegeben werden. Die Namen entsprechen dabei im Wesentlichen den Namen in der UML-Spezifikation.

Auf diese Weise können dabei mehrere Klassen angegeben werden, die das Interface `ArgoModule` oder ein davon abgeleitetes implementieren. Im Fall von ArgoMobile ist es das Interface `PluggableMenu`, das von der Klasse `ArgoMobilePlugin` implementiert wird. Dadurch können Einträge zum Hauptmenü hinzugefügt werden, die den Einstieg für die Funktionalität des PlugIns bieten. Der Module-Loader von ArgoUML durchsucht den Classpath nach JAR-Dateien, die die beschriebenen Bedingungen erfüllen und initialisiert die PlugIns.

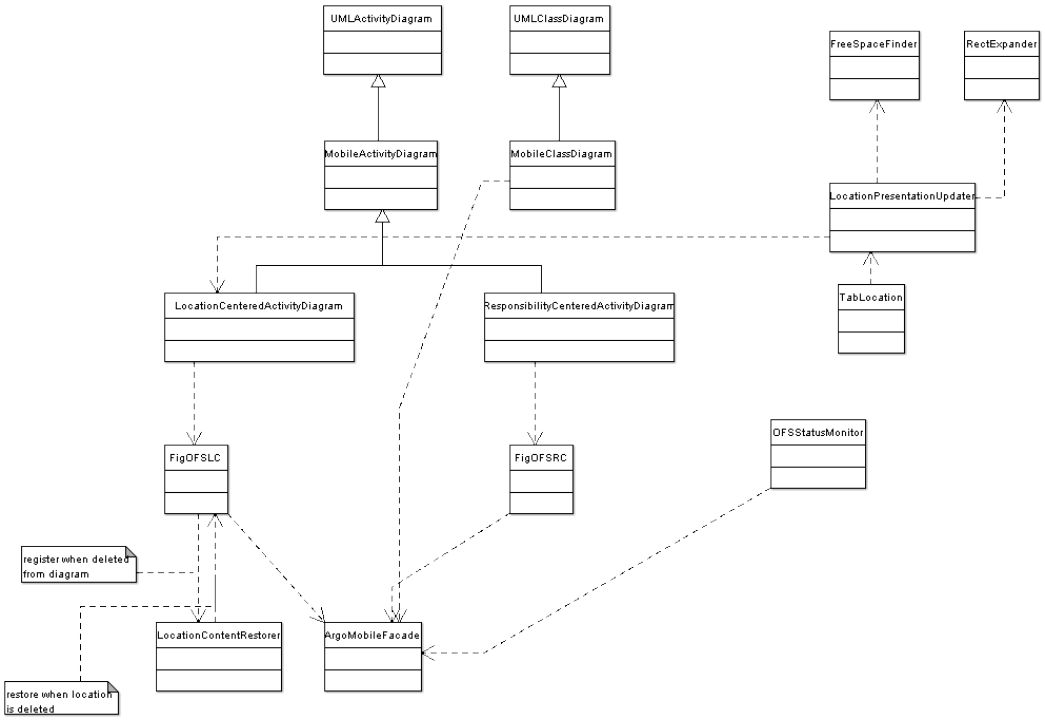
#### *Hinweise*

Der *Design Critics-Mechanismus* ist, wie schon erwähnt wurde, ein mächtiges Modul von ArgoUML. Zusätzlich zum reinen Anzeigen eines Problems bietet er noch weitere Möglichkeiten, die automatische Analyse in den Design-Prozess einzubinden. Dazu gehören zum Beispiel so genannte *Wizards*, die den Benutzer interaktiv beim Beheben von Problemen unterstützen. Die einzelnen Hinweis-Klassen werden dabei als Ableitungen von `Critic` realisiert und sind hauptsächlich verantwortlich für die Überprüfung der jeweiligen Regel. Dadurch ist es sehr einfach möglich, neue Hinweise hinzufügen.

Der Kern des Moduls bildet die Singleton-Klasse `Agency`. Sie stellt einen *Thread* dar, der im Hintergrund die Überprüfung der Design-Regeln in den einzelnen Hinweis-Klassen steuert. Tatsächlich iteriert `Agency` bei jedem Durchlauf über alle Elemente des Projekts und ruft alle entsprechend registrierten *Critics* auf. Dabei werden sowohl Modellelemente, als auch Diagramme betrachtet.

### 3.2.2 Die Struktur von ArgoMobile

Nachdem die relevanten grundlegenden Bestandteile von ArgoUML vorgestellt wurden, sollen nun die wichtigsten ( bzw. interessantesten ) Aspekte bei der Umsetzung von ArgoMobile beleuchtet werden. Als erster Überblick dient dabei das folgende Diagramm.




### 3.2.3 Implementierung als PlugIn

Architektur und Design von ArgoMobile sind zunächst zu einem großen Teil durch die Anforderungen bestimmt, die sich aus dem PlugIn-Prinzip ergeben:

- ArgoMobile muss sich zu einer bestehenden Installation von ArgoUML hinzufügen lassen, die beliebige andere PlugIns enthält.
- Der bestehende Code von ArgoUML darf nicht verändert werden, da jede Änderung das Verhalten anderer PlugIns oder Module beeinflussen könnte. Aus dem gleichen Grund dürfen keine Module oder verwendete Bibliotheken ausgetauscht werden.

Die Integration in die Hauptanwendung ist beschränkt auf die Möglichkeiten, die durch den in *** vorgestellten Erweiterungsmechanismus von ArgoUML gegeben sind. In *** werden einige Probleme, die sich daraus ergeben, etwas näher betrachtet.

### 3.2.4 Modellzugriff

In ArgoMobile ist der Zugriff auf das Modell entsprechen dem *Facade*-Muster die in der Klasse `ArgoMobileFacade` realisiert. Dort existieren zum einen Methoden zum Erstellen der Modellelemente, die eher zur vereinfachten Handhabung dienen. Am wichtigsten ist ein Satz von Methoden, die das Modell in Hinsicht auf das UML-Profil von ArgoMobile analysieren. Beispiele sind

- `public boolean isMobileOFS(Object handle)`: überprüft, ob das Modellelement „handle“ ein ObjectFlowState ist, der ein mobiles Objekt repräsentiert.
- `public String getLocationString(Object handle)`: gibt die vollstädige Standortangabe eines ObjectFlowStates zurück ( z.B.

LH123:Plane@MUC:Airport ).

Das UML-Profil wurde prinzipiell exakt so Umgesetzt, wie es in 2.2.3 erklärt wurde. Um die Referenz auf ein *ObjectFlowState* im *TaggedValue* „atLoc“ zu realisieren, wird die so genannte *MOF-Id* des referenzierten Modellelements gespeichert. Die *MOF-Id* wird auch in der XMI-Repräsentation des Modells verwendet.

### 3.2.5 Aktivitätsdiagramme

Wie oben beschrieben gibt es in ArgoMobile zwei verschiedene Ansichten für Aktivitätsdiagramme, die in zwei verschiedenen Diagramm-Klassen umgesetzt wurden: `LocationCenteredActivityDiagram` und `ResponsibilityCenteredActivityDiagram`. Die beiden Diagramme werden immer zusammen erstellt und stellen die gleichen Abläufe dar. Die Unterschiede zwischen den Ansichten wurden oben erläutert.

Beide Diagramm-Klassen sind von `UMLActivityDiagram` abgeleitet. Die Grundfunktionalität ist dabei unverändert übernommen worden. Stattdessen ist die Unterstützung des Profils hauptsächlich in den beiden Klassen `FigOFSLC` und `FigOFSRC` implementiert.

Wie in 2.3 erklärt wurde, kann es vorkommen, dass *ObjectFlowStates* von mobilen Standorten automatisch aus der *Responsibility Centered View* entfernt bzw. wieder hergestellt werden. Die Aktualisierung der Ansicht wird dabei von der Klasse `OFSStatusMonitor` gesteuert. `OFSStatusMonitor` ist ein *Singleton*, und wird beim Laden des Plugins als *Listener* in der `ModelEventPump` registriert. Reagiert wird auf Änderungen der Properties *type*, *taggedValue*, *incoming* und *outgoing* und außerdem auf das Löschen eines Modellelements.

In der *Location Centered View* wird eine Standortangabe wie beschrieben dadurch erstellt, dass die Figur eines Objekts in der Figur eines Standort-*ObjectFlowStates* platziert wird. Dadurch wird beim verschobenen `Fig`-Objekt die Methode `setEnclosingFig()` aufgerufen. In den Klassen `FigStereotypedActionState` und `FigOFSLC` ist diese Methode überschrieben. Dort wird der *TaggedValue* "atLoc" beim Modellelement der verschobenen Figur auf die MOF-ID des Modellelements der Standort-Figur gesetzt. Auf diese Weise kann also wie der Standort von Aktionen oder Objekten definiert werden. Gleichzeitig wird auf Darstellungsebene die Figur des Standorts zum Container für die verschobene Figur. Ab diesem Zeitpunkt wird beim Verschieben der Standort-Figur auch die eingeschlossene Figur mitverschoben. Das ermöglicht es, beliebig komplexe Standort-Hierarchien zu erstellen.

### 3.2.6 Synchronisierung

Zur Synchronisierung der beiden Ansichten muss beim Einfügen und Löschen von Modellelementen in einem Diagramm die Änderung ins verknüpfte Diagramm mit der jeweils anderen Ansicht übertragen werden. Diese Funktionalität ist in den Diagramm-Klassen implementiert. Die Synchronisierung wird in den Methoden `nodeAdded()` und `nodeRemoved()` ausgelöst. Beim Hinzufügen eines Objekts muss eine neue Figur erzeugt und auf dem jeweiligen Diagramm platziert werden. Als Position für die neue Figur wird von der Position der entsprechenden Figur im verknüpften Diagramm ausgegangen und dann der nächste Freie Platz gesucht, bei dem keine Überlappung entsteht. An dieser Stelle könnten weitaus intelligentere Verfahren angewandt werden um die Handhabung zu optimieren ( siehe 4.1 ).

### 3.2.7 Wiederherstellung von Standort-Figuren nach dem Löschen

Beim Bearbeiten der Standort-Hierarchie eines Objekts kann es notwendig sein, einen Standort aus einer unteren Position der Hierarchie zu entfernen. Dabei ergibt sich jedoch ein Problem in der *Location Centered View*, da durch die Löschroutine in ArgoUML alle in einem Container enthaltenen Figuren gelöscht werden. Daher müssen die Figuren aller Objekte, die sich - auch rekursiv - in dem gelöschten Standort befinden, wiederhergestellt werden. Diese Aufgabe übernimmt die Klasse `LocationContentRestorer`. Auch sie ist als *Singleton* implementiert und wird beim Laden des PlugIns als *EventListener* bei der `ModelEventPump` registriert. Sobald ein Standort-*ObjectFlowState* aus dem Modell gelöscht wird, geht die Löschroutine von ArgoUML anfangend von der höchsten Figur in der Schachtelung der Reihe nach die eingeschlossen Figuren durch und löscht sie aus dem Diagramm. Dadurch wird die Methode `removeFromDiagram()` in der betroffenen `Fig`-Classe aufgerufen. Diese ist in der Klasse `FigOFSLC` überschrieben um die gelöschte Figur zusammen mit ihrer derzeitigen Position und Größe zu sichern. Diese Informationen werden dabei mit dem Standort-ObjectFlowState verknüpft, der durch `atLoc` vom Modellelement der Figur referenziert wird. Nachdem alle Figuren aus dem Diagramm entfernt sind, wird das eigentliche Löschen des Modellelements durchgeführt. Das löst ein `PropertyChangeEvent` aus, welches wiederum vom `LocationContentRestorer` behandelt wird. Dort werden jetzt rekursiv, angefangen mit dem gelöschten Standort-OFS, alle Figuren rekonstruiert, die dem Standort-OFS zugeordnet sind. Anschließend sind also alle Figuren außer der des gelöschten OFS im gleichen Zustand wie vor dem Löschen.

### 3.2.8 Automatische Aktualisierung der Location Centered View

In 3.1.4 wurde erläutert, dass bei Verwendung des Standort-Editors die standortbezogene Ansicht eines Aktivitätsgraphen automatisch aktualisiert werden muss. Diese Aufgabe wird von der Klasse `LocationPresentationUpdater` übernommen. Das grundsätzliche Ziel bei der Aktualisierung ist es, das bisherige Layout so wenig wie möglich zu verändern und dabei die Übersichtlichkeit des Diagramms zu bewahren. Das hauptsächliche Problem dabei ist, dass eventuell Figuren von Standorten vergrößert werden müssen, um die Figuren der neu erstellten Unter-Standorte aufnehmen zu können. Dabei kann es leicht vorkommen, dass an der ursprünglichen Position der Figur nicht genügend Platz ist, so dass sich Überlappungen mit anderen Figuren ergeben würden. Da Überlappungen auf jeden Fall vermieden werden sollen, müssen automatisch Änderungen am Layout des Diagramms vorgenommen werden. Man kann dabei folgende Fälle unterscheiden:

- Die vergrößerte Figur soll in eine Standort-Figur eingefügt werden
  - In der umgebenden Figur ist an einer anderen Position ausreichend Platz für die einzufügende Figur vorhanden. Dann wird diese Position zum Einfügen verwendet und die Layout-Änderung ist abgeschlossen.
  - In der umgebenden Figur kann kein freier Platz für die einzufügende Figur gefunden werden. In diesem Fall muss die umgebende Standort-Figur vergrößert werden. Dadurch können allerdings ihrerseits wieder Überlappungen mit der Umgebung entstehen. Daher muss die Größenänderung gegebenenfalls rekursiv nach unten fortgesetzt werden.
- Die vergrößerte Figur soll direkt auf dem Diagramm platziert werden. In diesem Fall ist es nicht sinnvoll, eine neue Position zu wählen, weil dadurch das bisherige Layout des Diagramms stark verändert würde. Stattdessen wird der

Mittelpunkt der vergrößerten Figur festgehalten und alle anderen Figuren des Diagramms von der platzierten Figur weg geschoben.

Beim Platzieren einer Figur in einer anderen Figur wird die Klasse `FreeSpaceFinder` verwendet. Dort ist ein Algorithmus implementiert, der einen freien Bereich gegebener Größe innerhalb einer Menge von Rechtecken findet. Dabei wird ein Sweep-Line-Algorithmus eingesetzt, der auf der Menge der Minkowski-Summen der Rechtecke operiert. Beim vergrößern von Figuren kommt die Klasse RectExpander zum Einsatz. Dort wird ebenfalls durch ein Sweep-Line-Verfahren die Erweiterung eines Rechtecks mit dem geringsten Flächenzuwachs berechnet, bei der ein gegebenes Rechteck am Rand eingefügt werden kann. Eine detaillierte Betrachtung dieser geometrischen Algorithmen soll hier nicht erfolgen. Sie wurden im Rahmen der Entwicklung von ArgoMobile erstellt, wobei keine Zeit für eine weiterführende Optimierung blieb. Auch deshalb wurde bei der Implementierung auf Unabhängigkeit von ArgoUML oder ArgoMobile geachtet, um es zu erleichtern, die Algorithmen auszutauschen.

### 3.2.9 Spezielle Hinweise

Die Design Critics in ArgoMobile nutzen momentan nur die Basisfunktionalität des Critics-Moduls und sind daher sehr einfach gehalten. Im Prinzip umfasst jede Critic-Klasse nur die Implementierung der Methode `predicate2()`, in der die eigentliche Überprüfung der jeweiligen Design-Regel stattfindet. Die momentan enthaltenen Critics, die in *** beschrieben wurden, beziehen alle außer `CrDiagramDirty` auf mobile Objekte und Standorte in Aktivitätsdiagrammen. Bei der Implementierung wurde darauf geachtet, nicht-triviale Modell-Zugriffe stets in die Klasse `ArgoAgileFacade` zu sammeln und so eine leicht Basis für die Entwicklung von zusätzlichen Design Critics herzustellen.

Bei den beiden Critics `CrWrongMoveActionConfiguration` und `CrWrongCloneActionConfiguration` zeigt sich eine Besonderheit im Vergleich zu den herkömmlichen Hinweisen in ArgoUML. Dort bezieht sich die Regel nämlich nicht nur auf ein einzelnes Modellelement, sondern auf die Zusammenstellung eines ActionStates mit durch Transitionen verbundenen *ObjectFlowStates*. Ein Problem wird beispielsweise gefunden, wenn es zu einem eingehenden *ObjectFlowStates* keinen entsprechenden ausgehenden *ObjectFlowStates* gibt ( siehe 3.1.5 ). In diesem Fall kann jedoch nur eines der beiden beteiligten Modellelemente, also entweder der *ObjectFlowStates oder* der ActionState, als Quelle des Problems dargestellt werden. in ArgoMobile wurde willkürlich der *ActionState* gewählt.

# 4 Probleme und Verbesserungsvorschläge

Zum dem Zeitpunkt, an dem dieses Dokument entsteht, befindet sich ArgoMobile noch in einem Prototypen-Stadium. In diesem Kapitel werden einige Probleme und Vorschläge zu Verbesserungen erläutert, für deren Lösung bzw. Umsetzung bisher keine Zeit blieb.

## *4.1 Probleme beim automatischen Layout*

ArgoMobile ermöglicht in der aktuellen Version das Erstellen von Klassendiagrammen

und Aktivitätsdiagrammen nach dem in [1] vorgestellten Profil. Prinzipiell lassen sich wie in 3.1.3 beschrieben beliebig verschachtelte Standortangaben erstellen und entweder direkt im Diagramm oder durch den Standort-Editor bearbeiten. Dabei erleichtert es der Synchronisations-Mechanismus, die beiden Ansichten konsistent zu halten. Gerade bei der automatischen Platzierung von Figuren durch den Standort-Editor ist noch Raum für Verbesserungen. Ziel ist es, bei automatischen Änderungen an den Diagrammen, die Übersichtlichkeit zu erhalten und möglichst die Wünsche des Benutzers bezüglich des Layouts zu beachten. Vor allem sollten bewusste Entscheidungen des Benutzers nicht verletzt werden. Ein Beispiel für solch eine bewusste Layout-Entscheidung wären Transitions-Kanten, die der Benutzer von Hand in einer bestimmten Weise ausgerichtet und geformt hat. In der momentanen Version von ArgoMobile kann es leicht vorkommen, dass die Endknoten der Kanten verschoben werden, um Platz für eine zu platzierende Figur zu schaffen. Eine Verbesserung wäre z.B. durch eine interaktive Lösung möglich, bei der der Benutzer selbst entscheiden kann, wohin überlappende Figuren verschoben werden.

Ein anderer Punkt ist die Platzierung von Figuren bei der Synchronisierung. Eine Figur, die vom Benutzer erstellt und dabei in einer Ansicht platziert wurde, muss automatisch in der anderen Ansicht positioniert werden. Bisher werden dafür einfach die gleichen absoluten Koordinaten verwendet. Sinnvoller wäre jedoch eine Positionierung, die sich an der Lage zu benachbarten Figuren orientiert.

## *4.2 Anzeige von Standorten in der zuständigkeitsbezogenen Ansicht*

Ein weiterer offener Punkt ergibt sich bei der Darstelllung von Standorten in der zuständigkeitsbezogenen Ansicht. In 2.3 wurde beschrieben, dass dort Standort-*ObjectFlowStates* genau dann dargestellt werden, wenn sie entweder kein Objekt beinhalten, oder mit einer Aktion verknüpft sind. Dieses Verhalten ist jedoch nicht in jedem Fall wünschenswert. Zum Beispiel sollte in Abbildung 3 folgenden Abbildung der markierte object flow state nicht dargestellt werden.

In diesem Fall wäre es besser, dem Benutzer die Möglichkeit zu geben, explizit zu definieren, ob ein object flow state in der zuständigkeitsbezogenen Sicht angezeigt werden soll. Diese Information muss dann zusammen mit dem Diagramm gespeichert werden. Die API von ArgoUML bietet jedoch keine Möglichkeit, bei der Persistierung der Präsentationsschicht, Informationen hinzuzufügen[5]. Ein Ausweg würde darin bestehen, einen TaggedValue zu verwenden, der diesen Status als boolschen Wert enthält und dem object flow state hinzugefügt wird.

## *4.3 Keine Unterstützung von Swimlanes*

ArgoUML unterstützt in der Aktuellen Version (0.20) keine Swinlanes in Aktivitätsdiagrammen. Diese sind jedoch im Mobile UML-Profil ein wichtiger Bestandteil der zuständigkeitsbezogenen Ansicht. Dem Issue-Tracking-System von ArgoUML ist zu entnehmen, dass an der Einführung dieses Features schon seit längerem gearbeitet wird[6]. Es ist jedoch noch nicht klar wie lange es dauern wird, bis es in ein Release aufgenommen wird. Bis dahin bleibt nur, die *Swimlanes* durch die Zeichentools von ArgoUML nachzubilden oder das exportierte Bild eines Diagramms in

5 ArgoUML verwendet zum Speichern der grafischen Informationen die Precision Graphics Markup Language (PGML). Siehe [8]

6 Siehe Issue #3110: http://argouml.tigris.org/issues/show_bug.cgi?id=3110

einer separaten Grafik-Anwendung zu bearbeiten.

Ein völlig anderer Ansatz wäre es, die Zuständigkeit ähnlich wie den Standort als O*bjectFlowState* darzustellen, der als Container O*bjectFlowState* und ActionStates aufnimmt. Auf Modellebene könnte diese Beziehung ebenso wie "atLoc" durch einen *TaggedValue* gespeichert werden. Der Aufwand für die Umsetzung ist als eher gering einzuschätzen, da letztendlich Teile aus der Implementierung der standortbezogenen Ansicht übernommen werden. Allerdings hätte dieses Vorgehen eher den Charakter eines „Workarounds“.

## *4.4 Berücksichtigung der Klassenstruktur*

Die Grundlage für die Modellierung mobiler Systeme besteht immer aus einem oder mehreren Klassendiagrammen, in denen die Klassen der mobilen Objekte bzw. Standorte definiert und mit den entsprechenden Stereotypes markiert werden. ArgoMobile verwendet die Klassen als Basis für die mobilen Objekte und Standorte. Die Struktur der Klassendiagramme wird bisher vom PlugIn nicht weiter betrachtet. Sie ließe sich jedoch verwenden, um eine genauere Prüfung der Aktivitätsdiagramme zu ermöglichen. Das verwendete UML-Profil schreibt nicht vor, dass Standort-Beziehungen im Klassendiagramm explizit angegeben werden müssen. Trotzdem lässt sich auch explizit durch Assoziationen modellieren, welche Klassen für den Standort eines mobilen Objekts bzw. eines anderen Standorts in Frage kommen. Diese Einschränkungen könnten dann durch Design Critics überprüft werden. Außerdem wäre es so möglich, den Benutzer beim Editieren zu unterstützen, indem im Standort-Editor nur solche Standort-Instanzen zur Auswahl gestellt werden, die gemäß Klassendiagramm für eine Position in der Standort-Hierarchie zulässig sind.

## *4.5 Offene Punkte bei der PlugIn-Unterstützung von ArgoUML*

Der API von ArgoUML ist anzumerken, dass es noch relativ wenige PlugIns gibt, die unabhängig von der Hauptanwendung entwickelt werden und dadurch keine Veränderungen an deren Code vornehmen dürfen. Für ein PlugIn wie ArgoMobile, das weitreichende Erweiterungen an Diagrammen und Oberfläche einführt, ergeben sich einige Einschränkungen. Dieser Abschnitt soll einige Anregungen für Verbesserungen liefern, die sich aus den Erfahrungen bei der Entwicklung von ArgoMobile ergeben haben:

- Der ModuleLoader von ArgoUML würde es prinzipiell ermöglichen, das JAR-Archiv eines PlugIns einfach in ein Verzeichnis "ext" im ArgoUML-Verzeichnis zu kopieren. Beim Start werden dann die im Manifest angegebenen Klassen per ClassLoader geladen und registriert. Das funktioniert allerdings nicht bei alle verwendeten Klassen. Z.B. wird die Klasse TabLocation, die den Standort-Editor enthält, nicht gefunden und beim Starten wird eine ClassNotFoundException geworfen. Als Ausweg bleibt nur, das JAR von ArgoMobile in den Classpath beim Starten von ArgoUML aufzunehmen, also das Start-Skript anzupassen.
- Im Projekt-Explorer könnte die Anzeige ( das Icon ) der Knoten in Abhängigkeit vom Stereotype des Modellelements bestimmt werden. Bisher können Icons nur für Klassen ( also de facto Diagramme ) registriert werden. In ähnlicher Weise könnten über Callback-Interfaces Spezielle Kommandos für das Kontext-Menü hinzugefügt werden. Konkret könnten so in ArgoMobile Instanzen für mobile Objekte bzw. Standorte über das Kontextmenü der entsprechenden Klasse erstellt werden.
- Beim Event-Handling auf wäre es hilfreich, wenn Komponenten bei bestimmten

Ereignissen bereits vor der Ausführung einer Aktion informiert würden und in den Ablauf der Aktion eingreifen könnten. Auf Modellebene werden ohnehin PropertyChange Events eingesetzt. Daher läge die Verwendung von constrained properties bzw. VetoableChangeListener nahe[7]. Hauptsächlich ist dabei das Löschen von Modellelementen oder Assoziationen interessant. So könnte z.B. vermieden werden, dass eine Klasse gelöscht wird, wenn sie im Projekt noch benötigt wird.

- Bisher gibt es auch noch keine klar definierte Möglichkeit für PlugIns, eigene (Meta-) Daten beim Speichern des Projekts einzubringen. Bei ArgoMobile könnte das interessant werden, wenn die Gedanken aus 4.1 und 4.2 übernommen werden. Wenn der Benutzer, wie dort beschrieben, Einstellungen zum Layout eines Diagramms trifft, dann sollten diese auch zusammen mit dem Diagramm gespeichert werden.

## 4.6 ClassifierRole oder ClassifierInState

Ein erster Prototyp von ArgoMobile entstand im Rahmen der Definition des Profils und basierte auf der UML-Version 0.16.1 ( siehe [9] ). Dabei wurde ein anderer Ansatz für die Abbildung des Profils auf das UML-Metamodell verwendet, der sich mehr als der jetzt verwendete an den Ausführungen in [1] orientiert. Zur Veranschaulichung des Unterschieds ist Unten ein Objektdiagramm abgebildet, das analogen Sachverhalt wie Abbildung 2 darstellt.

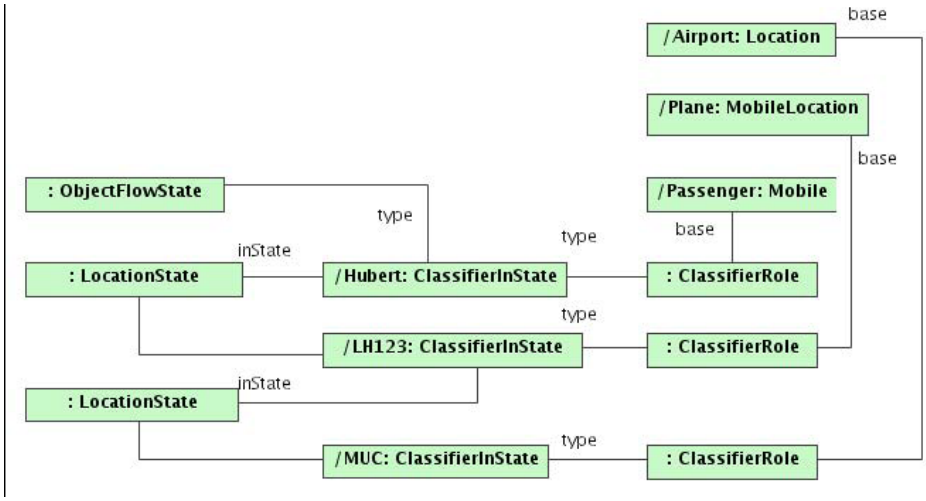



*Abbildung 12: Algternative Abbildung auf das UML-Metamodell*

Es ist zu erkennen, dass als Type für die *ObjectFlowStates* hier sozusagen *ClassifierInState* vorgeschaltet werden, statt direkt *ClassifierRoles* zu verwenden werden, die vergleichbar jeweils räumliche Instanzen repräsentieren. Die Standort-Beziehung wird hier nicht durch einen *TaggedValue* modelliert, sondern durch die Verknüpfung mit einer neu eingefügten Meta-Klasse: *LocationState.* Diese ist von *State* abgeleitet und kann eine weitere Verknüpfung zu einem *ClassifierInState* enthalten, um verschachtelte Standortangaben zu ermöglichen.

7 siehe Kapitel

Das Einführen der Metaklasse LocationState die Veränderung des Metamodell-Repositories von ArgoUML. Da bei der Neuauflage von ArgoMobile Wert auf die strikte Einhaltung des PlugIn-Prinzips gelegt wurde, fiel die Entscheidung also zugunsten des leichtgewichtigeren Ansatzes, der in 2.2.3 vorgestellt wurde.

Auf der anderen Seite entspricht die Verwendung von *ClassifierInState* der UML-Semantik besser als die direkte Verwendung von *ClassifierRoles* ( „ein Objekt ist in einem Zustand“ ). Zudem wurde beim Entwurf des Ursprünglichen Ansatzes daran gedacht, mobilen Objekten auch andere Zustände als den Standort zuzuweisen ( z.B. „Passagier ist hungrig“ ). Dies wäre durch die Semantik von *ClassifierInState* ideal abgebildet, da sich dieser mit beliebig vielen *States* assoziieren lässt. Allerdingst bietet sich die gleiche Möglichkeit auch für den aktuell verwendeten Ansatz durch die reine Verwendung von *TaggedValues.*

Ein viel versprechender Kompromiss zwischen der ursprünglich angedachten und der aktuell verwendeten Umsetzung wäre es also, wie oben beschrieben *ClassifierInStates* zu verwenden und statt der Einführung von LocationState den UML Extension Mechansim einzusetzen. Konkret müsste dafür LocationState durch einen mit Stereotype versehenen SimpleState ersetzt werden. Die Assoziation mit dem *ClassifierInState des* Standorts ließe sich wiederum durch einen *TaggedValue* realisieren. Das gleiche ließe sich auch für eventuelle andere Zustände ( „Hunger“ ) übertragen. Dieser Ansatz ließe sich mit vertretbarem Aufwand in ArgoMobile durchaus umsetzen, würde allerdings einiges an zusätzlicher Komplexität mit sich bringen.

Die offene Frage ist nun, ob sich der Aufwand für die Umstellung durch dir größere Nähe zur ursprünglichen UML-Semantik rechtfertigen lässt. Dabei ist auch daran zu denken, dass sich eine höhere Komplexität auch für andere Anwendungen bemerkbar macht, die die Modelle aus ArgoMobile, verwenden ( z.B. in Form von XMI ).

# 5 Ausblick

In den vorangegangen Kapiteln wurde beschrieben, wie sich mit ArgoMobile Klassen- und Aktivitätsdiagramme zu Strukturen bzw. Abläufen in mobilen Systemen erstellen lassen. Wie dargestellt lassen sich zusätzliche Informationen zur Örtlichkeit von Objekten und Aktionen modellieren und im Diagramm Darstellen. Dadurch kann man mobile Systeme besser durch UML-Diagramme visualisieren und so die Kommunikation innerhalb eines  Projekt-Teams verbessern. Auf der anderen Seite liegen die erstellten Modelle in wohldefinierter und standardisierter Form vor und ermöglichen so die Weiterverarbeitung und den Austausch mit anderen Applikationen. Model Driven Development ist als Methode die konsequente Weiterführung dieses Gedankens. Durch Einbeziehung der Informationen darüber, wo eine Aktion stattfindet, könnten bei der Code-Generierung Entscheidungen hinsichtlich technischer Details getroffen werden. Auf der anderen Seite ist es möglich, Informationen über Standorte und Standortänderungen in formalen Methoden der Prozessalgebra und der Systementwicklung zu verwenden. Diese Informationen könnten zumindest teilweise aus MobileUML - Aktivitätsdiagrammen gewonnen werden.

Zusammenfassend kann man sagen, dass das PlugIn ArgoMobile schon in seinem jetzigen Zustand durchaus eine Bereicherung bei der Entwicklung von mobilen Systemen sein kann. Das ArgoUML-Projekt ist nach wie vor sehr aktiv und der Editor wird mittlerweile auch in der professionellen Softwareentwicklung eingesetzt. Gerade durch das Zusammenwirken mit dem sehr verbreiteten MDA-Framework AndroMDA [10] und die Umstellung auf das von NetBeans entwickelte Model Data Repository

(MDR) als Modell [7], ist die Zukunft des Projekts wohl gesichert. Bei der zukünftigen Weiterentwicklung sowohl von ArgoMobile, als auch von ArgoUML selbst, sollte die Stabilität und die Skalierbarkeit im Vordergrund stehen. Denn gerade bei komplexen Projekten kann die UML ihre Stärken ausspielen.

# Literaturverzeichnis


[1] Extending activity diagrams tomodel mobile systems. 2003
[2] OMG Unified Modeling Language Specification v. 1.4. 2001
[3] Cookbook for Developers of ArgoUML. ( http://argouml-stats.tigris.org/documentation/defaulthtml/cookbook/ )
[4] Java Metadata Interface (JMI). ( http://java.sun.com/products/jmi/ )
[5] OMG's MetaObject Facility. ( http://www.omg.org/mof/ )
[6] Novosuft UML Library. ( http://nsuml.sourceforge.net/ )
[7] Metadata Repository. ( http://mdr.netbeans.org/ )
[8] Precision Graphics Markup Language (PGML). ( )
[9] . *D3.3: UML-based Prototyping Tool*
[10] AndroMDA. ( http://www.andromda.org/ )