Diplomarbeit

Enhancing an Open Source UML Editor by Context-Based Constraints for Components

Martin Skinner

12th December 2001

Fachgruppe Computerunterstützte Informationssysteme (CIS)
Institut für Softwaretechnik und Theoretische Informatik
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin
Betreuer: Felix Bübl
Gutachter: Prof. Dr. Herbert Weber, Dr. Ralf-Detlef Kutsche

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den Quellen wörtlich oder inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Abstract:

Before starting a detailed design, a specification model of the component-based system assists the software developer in early problem detection as soon as possible in the development process. The Component Constraint Language (CCL) developed by CIS at the Technical University Berlin enables the developer to add context-based constraints (CoCons) to a component specification model. This produces a model which goes beyond the simple description of the system's static structure. At this time, there is no tool to integrate the component specification model into the development process. The primary goal of this master's thesis was to design such a tool, thereby supporting the Continuous Software Engineering (CSE) philosophy.

The open source modelling tool ArgoUML serves as a starting point. ArgoUML is enhanced with a plugin mechanism, transforming the previously monolithic application into a component-based system, allowing pluggable components (plugins) to be developed independently from the application. This mechanism is then used to extend ArgoUML with a plugin supporting the CCL concepts.

Zusammenfassung:

Noch vor der Erstellung eines detaillierten Entwurfs hilft ein Spezifikationsmodell eines komponenten-basierten Systems dabei, Probleme so früh im Entwicklungsprozeß wie möglich zu entdecken. Die Sprache CCL ('Component Constraint Language') wurde bei CIS, Technische Universität Berlin, entwickelt und erlaubt den Entwickler 'Context-based Constraints' dem Spezifikationsmodell hinzuzufügen. Dadurch entsteht ein Modell, das über die Beschreibung der statischen Struktur des Systems hinausgeht. Zur Zeit existiert allerdings kein Werkzeug, das das Komponentenspezifikationsmodell in den Entwicklungsprozeß integriert. Ziel dieser Diplomarbeit war der Entwurf eines solchen Werkzeugs, um die Philosophie des Continuous Software Engineering (CSE) zu unterstützten.

Das Open-Source Modellierungswerkzeug ArgoUML dient als Ausgangspunkt. Mit einem Plugin-Mechanismus wird ArgoUML von einer monolithischen Applikation in einem komponenenten-basierten System umgewandelt. Dadurch können austauchbare Komponenten (plugins) unabhängig von der Applikation entwickelt werden. Dieses Mechanismus wird dann verwendet um ArgoUML mittels eines Plugins um die CCL-Konzepte zu erweitertern.

This document is available online at http://www.cocons.org

Contents

I. Base Technologies and Methods 9

1. Software Components and Context-Based Constraints 10
1.1. The Component Constraint Language CCL 11
1.2. Component Specification and the Unified Modeling Language 12
1.3. Enhanced Global Business Type Diagrams 12
1.4. Enhanced Component Specification Diagrams 13
1.5. Enhanced Interface Specification Diagrams 13

2. Integrating the Component Constraint Language in UML 16
2.1. The Unified Modelling Language 16
2.2. Context Properties in UML 17
2.3. Context-based Constraints in UML 18

3. Software Design Patterns 19
3.1. Observer Pattern 19
3.2. Adapter Pattern 19
3.3. Singelton Pattern 21
3.4. Abstract Factory and Factory Method Patterns 21
3.5. Composite Pattern 21
3.6. Model View Controller 23

4. XML – The eXtensible Markup Language 25
4.1. XML and Metadata 25

5. Unit Testing 27
5.1. The Java Unit Testing Framework JUnit 28

II. Analysis of ArgoUML 30

6. The UML Metamodel Library NSUML 31
6.1. Primitives 32
6.2. Enumerations 32
6.3. Datatypes 34

6.4. Elements . . . 34
6.5. Accessing and modifying metaattributes . . . 35
6.6. Accessing and modifying metaassociations . . . 35
6.7. NSUML reflective API . . . 37
6.8. NSUML event notification . . . 38
6.9. NSUML undo/redo support . . . 39
6.10. Model Persistance . . . 40

7. The Graph Editing Framework Library GEF 42
7.1. GEF View Architecture . . . 42
7.2. GEF Controller Architecture . . . 42
7.2.1. GEF command classes . . . 44
7.2.2. GEF Editing Modes . . . 44
7.2.3. GEF Selection classes . . . 46
7.3. Diagram Persistance . . . 47

8. ArgoUML Architecture 49
8.1. The UML Diagrams . . . 49
8.2. The Details Pane . . . 53
8.3. The Navigator Pane . . . 54
8.4. The Design Critics . . . 56
8.5. The ArgoUML project . . . 57

III. The CCL Modelling Tool 59

9. Extending the ArgoUML Modelling Tool 60
9.1. Adding a Plugin Mechanism . . . 60
9.2. Avoiding source dependency . . . 62
9.3. Avoiding distribution dependency . . . 63
9.4. Plugin dependencies . . . 66
9.5. Limits . . . 66

10. The CoCons Physical and Object Models 67
10.1. Extending the NSML Library . . . 67
10.2. The CoCons CCL Java Library . . . 71
10.3. CoCons Model Persistance . . . 72
10.3.1. Extending the UML-Document Type Definition . . . 72
10.3.2. Writing XMI documents . . . 75
10.4. Object Model Test Cases . . . 75
10.4.1. Testing metaattributes . . . 75
10.4.2. Testing metaassociations . . . 76
10.4.3. Testing persistance . . . 77

11. The CCL Editor Diagrams **79**
11.1. The Figure Classes 80
11.1.1. The Context-based Constraint Figure 83
11.1.2. The Context Condition Subfigure 84
11.1.3. The Context Elements Subfigure 87
11.1.4. The Context Property Figure 87
11.1.5. The Component Spec and Interface Type Figure 88
11.1.6. The Interface Dependency Figure 90
11.2. The constraint diagram 90
11.2.1. The Diagram class 90
11.2.2. The Graph Model 92
11.2.3. The Diagram Renderer 94
11.3. The business type diagram 94
11.3.1. The Diagram class 95
11.3.2. The Graph Model 96
11.3.3. The Diagram Renderer 98
11.4. The component specification diagram 98
11.4.1. The Diagram class 98
11.4.2. The Graph Model 99
11.4.3. The Diagram Renderer 99
11.5. The interface specification diagram 99
11.5.1. The Diagram class 100
11.5.2. The Graph Model 100
11.5.3. The Diagram Renderer 101
11.6. The Diagram Actions 101
11.6.1. Creating Stereotyped Nodes 101
11.6.2. Creating Stereotyped Edges 101
11.6.3. Adding Context Properties 102
11.7. Integrating the diagrams in ArgoUML 102

12. Extending the ArgoUML Workspace **104**
12.1. Extending the Details Pane 104
12.1.1. The Tagged Values Tab 106
12.1.2. The Stereotype Property Panel 106
12.1.3. The Context-based Constraint Property Panel 107
12.1.4. The Context Condition Property Panel 108
12.1.5. The Tag Definition Property Panel 109
12.1.6. The ContextProperyTag Property Panel 109
12.2. Extending the Navigator Pane 110
12.3. Adding new Design Critics 111
12.4. Plugin Initialization 112

13. Conclusion **115**

A. Necessary Source Modifications **116**
A.1. Bugfix in package ru.novosoft.uml.undo 116
A.2. Modifications to the ArgoUML sources 116
A.2.1. The plugin package 117
A.2.2. Accessing the Application Menu Bar 118
A.2.3. Extendable Property Panels 118
A.2.4. The extended NSUML library 118

Introduction

Changing requirements is a major challange to software development. The component approach needs to address this issue – by designing for change. Properly done, components become replaceable with limited impact on the rest on the system.

Before starting detailed design, a specification model of the component-based system assists the software developer in early problem detection as soon as possible in the development process. With the use of the Component Constraint Language (CCL), the developer can enrich his specification model with context-based constraints (CoCons), producing a component specification model which goes beyond the simple description of the systems static structure.

At this time, there is no tool to integrate the component specification model into the development process. The goal of this work was to design such an tool. The resulting design document should lay the groundwork for the development of a software modelling tool which supports the continuous software engineering (CSE) perspective including graphical representation and editing of the component specification model in the visual language CCL and support for verification of CoCons during modifications of the model. In order to avoid 're-inventing the wheel', the open source modelling tool ArgoUML was selected as a starting point. It contains support for many concepts every modelling tools requires.

This work is divided into three main parts. The first part gives a short introduction to the base technologies and methods the reader should be familiar with in order to fully understand this work.

An analysis of the ArgoUML architecture is a prerequisite to extending the ArgoUML application. A detailed, in-depth description of the ArgoUML internals would go beyond the scope of this work, so the second part focuses on the high-level view of the architecture and goes into a more detailed perspective only when necessary.

The third part is a detailed design document describing how ArgoUML can be modifed to support pluggable components ('plugins'), and how a plugin supporting the new modelling concepts introduced in CCL can be implemented.

Part I.

Base Technologies and Methods

1. Software Components and Context-Based Constraints

Software components can be considered as building blocks from which complex systems are constructed. Components share some of the fundamental principles of object technology:

Unification of data and function each component consists of data values (which form its state) and the functions that access this data.

Encapsulation the client needs no knowledge of the component's implementation, only its specification.

Identity each component is unique identifiable. Two different components could have the same state (which means they could be considered *equal in value*) but still have their own identities.

But components are not simply objects – some differences include:

Component Infrastructure in order for components to be able to interact, there must be some basic standard on which inter-component communication is built. Some real-world examples are COM+, Enterprise Java Beans (EJB) and CORBA.

Component Interfaces build on the concept of encapsulation by adding another level of indirection: A client does not depend on a component, but on a component's interface. This means that a component can be replaced by another component offering a 'compatible interface' without having to change the clients.

Interface Specification the notion of a 'compatible interface' is dictated by the interface specification. Depending on the infrastructure, an exported interface could be considered compatible if the services offered are a superset of the services requested.

Component Specification since clients access components only through their interfaces, the specification of a component is reduced to the interfaces it offers (exports) and the interfaces it uses (imports).

Component Architecture the structural relationships between components and their behavioral dependencies.

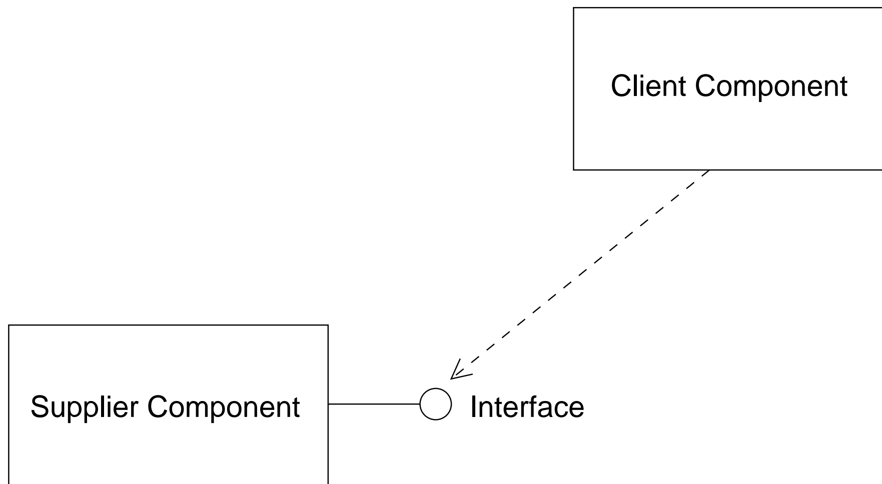


Figure 1.1.: A simple component architecture

This new level of indirection makes components replaceable – and underlines the different goals of component-based and object-oriented approaches: while object-oriented designs promise the *reusablity* of its subsystems, component-based architectures promise the *replaceability* of individual components. The internals of a component may very well be *implemented* with an object-oriented language. Most component infrastructures encapsulate the implementation so well as to allow different components to be implemented in different languages – but component implementation is beyond the scope of the work.

1.1. The Component Constraint Language CCL

The specification of components and interfaces can exists on many levels: some formal, some informal. The simplest form of formal specification is the purely syntactical specification: do the function name and parameters the client use match the signature of the function offered? The most flexible form is the specification in a natural language: although sometimes unavoidable, this can lead to ambiguity.

Constraints lie somewhere in the middle. An example of a constraint is 'a customer must have a name'. A constraint is usually expressed in a formal language as an invariant which must be fulfilled whenever the system is in a stable state. The above example, expressed in a formal constraint language could be '`Customer.name->notEmpty`'. By using a formal language it becomes possible to automatically check whether all constraints are fulfilled.

This constraint applies to a single, directly specified model element: `Customer`. The component constraint language (CCL) adds a new kind of constraint where it is possible to specify model elements *indirectly*. The basic idea is to add *context-properties* to model elements which hold meta-information about the model element. A *context-based constraint* (CoCon) uses these context properties to identify to which model elements it

applies. It then becomes possible to specify 'all model elements with the context-property `PersonalData` are unreadable by the component `WebServer`'.

1.2. Component Specification and the Unified Modeling Language

Although the Unified Modelling Language (UML) includes the notion of 'component', it is not the same as the component handled here. The UML definition is:

> A component type represents a distributable piece of implementation of a system, including software code (source, binary, or executable) but also including business documents, etc. in a human system. [...] A component instance represents a run-time implementation unit and may be used to show implementation units that have identity at run-time, including their location on nodes. [UML1.3, 3.97.1]

Clearly, this is focused on the *distribution* and *deployment* of components and has nothing to do with the *specification* of components, which is the focus of this work. UML was designed to describe object-oriented analysis and design, and doesn't describe component-based systems very well. One possibility of expressing the specification of components in a component architecture with UML can be found in [Cheesman+2001]. This 'UML Components' approach is used in this work, although other approaches are certainly possible.

In this approach, static structure diagrams (informally known as class diagrams) are used to model component specification. Stereotypes are used to indicate that the classes represent specification-level entities – these are specification models, not implementation models. The following sections describe the various specification models used in [Cheesman+2001] and enhance them with the context-properties introduced in [Bübl2001-CCL].

1.3. Enhanced Global Business Type Diagrams

Business type diagrams [Bübl2001-CCL, section 2.2] show the business types which are in the domain of the system. Business types represent the data objects which are exchanged between the components in a system.

Enhanced business type digrams are modeled using a subset of the UML static structure diagrams enhanced with context properties from the CoCons metamodel. Business types are represented by the UML `Class` metaclass but have less detail – there are no methods and the attributes are not yet fully defined. To set them apart from normal implementation classes, they always have a stereotype such as `<<type>>` or `<<info type>>`. Each business type can have context properties – represented by rectangles with crossed corners.

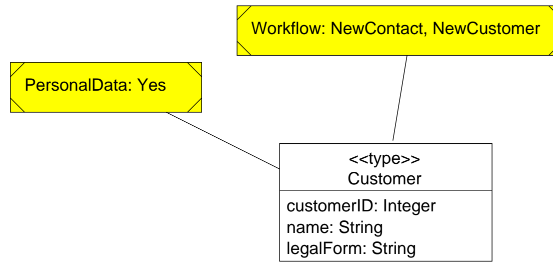


Figure 1.2.: Enhanced global business type diagram

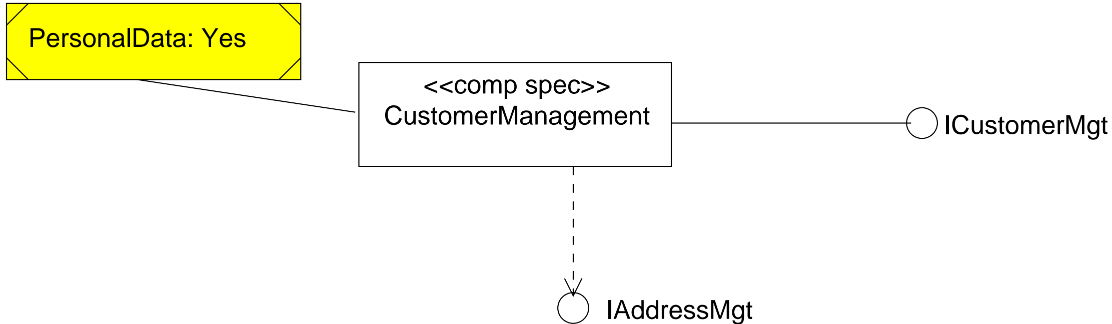


Figure 1.3.: Enhanced Component Specification Diagram

1.4. Enhanced Component Specification Diagrams

Component specification diagrams show what interfaces components offer (export) and what they assume are available (import). This diagram type is not the same as the UML component *deployment* diagram which is geared towards implementation and deployment of components. The focus of this diagram is the *specification* of components. For this reason, they not modeled using the UML `Component` metaclass, but with the `Class` metaclass having the stereotype `<<comp spec>>`. As with business types diagrams, components specifications may also have context properties.

The interfaces are also represented by the `Class` metaclass, but have the stereotype `<<interface type>>`. In the diagram they are depicted with the usual 'lollipop' notation.

1.5. Enhanced Interface Specification Diagrams

These diagram further refine the interfaces used in component specification diagrams. They specify which methods are part of the interface and what business types (or infomation types[1]) are required by the methods. Note that the `<<interface type>>` classes

[1] according to [Cheesman+2001, 3.8.2], information types are 'views' on a business type, containing only the information a certain interface cares about

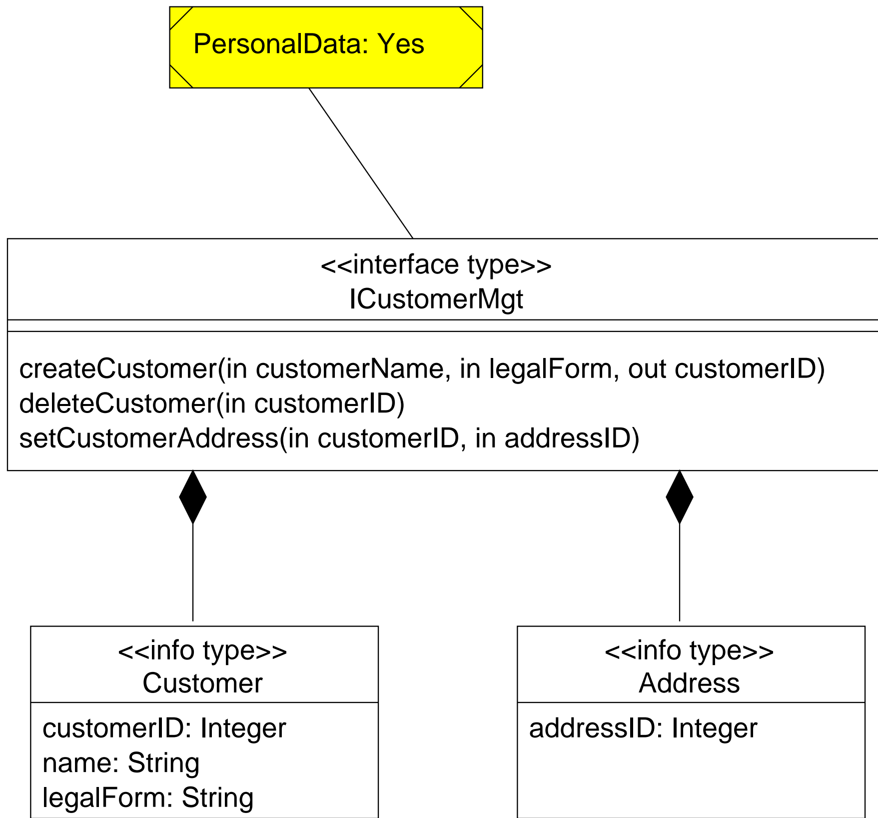


Figure 1.4.: Enhanced interface specification diagram

in this diagram are the same model elements as the those in the enhanced component specification diagram – they are simply rendered differently and in more detail.

2. Integrating the Component Constraint Language in UML

The Component Constraint Language is a graphical as well as a textual notatation for describing context-based constraints. Although CCL is an independent langauge, it is possible to combine it with other modelling languages. This chapter describes how CCL can be integrated in the Unified Modelling Language.

2.1. The Unified Modelling Language

> The Unified Modeling Language (UML) provides system architects working on object analysis and design with one consistent language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling.[UML1.3, introduction]

The Object Management Group (OMG) describes a four-level metadata architecture [XMI1.1, section 1.2.1] known as the Meta Object Facility (MOF). This is the OMG's standard for defining, representing and managing metadata. The MOF-Model (the meta-metamodel) resides on the level with the highest abstraction. This describes the basic building blocks from which metamodels are constructed.

UML is a M2-metamodel – it can be described with MOF-model and can describe user level (M1) models. UML defines metaclasses such as classes, associations, states and transitions which the system architect uses to describe his system.

UML 1.3 is the most widely accepted and supported version, and although version 1.4 of the specification has been released in September 2001, most of the available modelling

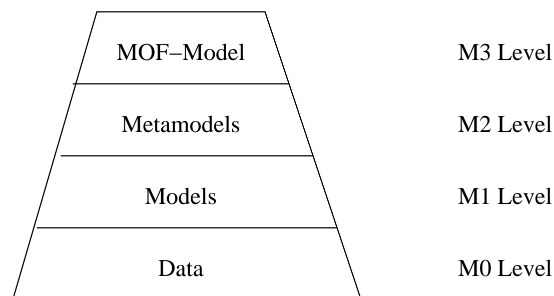


Figure 2.1.: MOF 4-Level Metadata Architecture

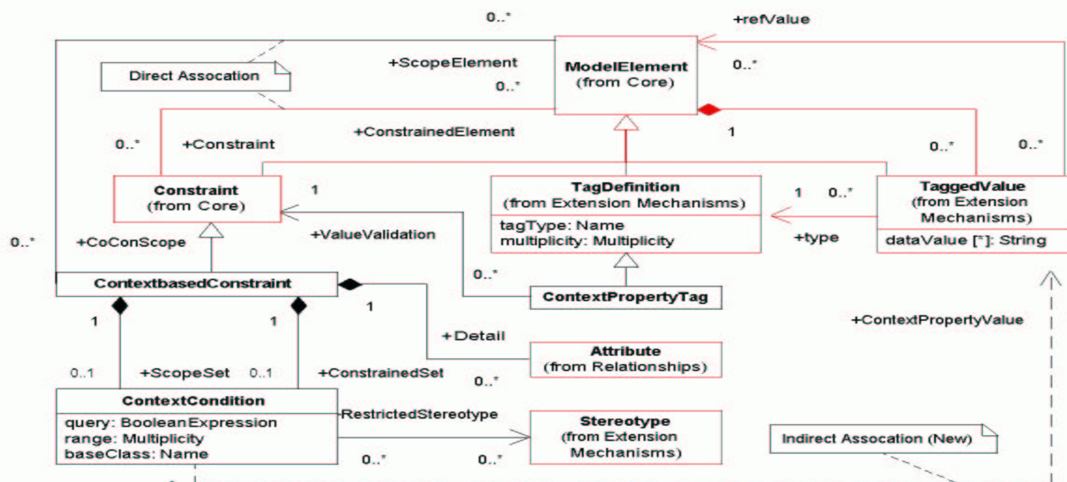


Figure 2.2.: Simplified CCL metamodel

tools do not yet support this version.

Not all of the CCL concepts can be described with the UML metamodel. For this reason, the UML 1.4 metamodel has been extended with the elements shown in figure 2.2. The following sections describe the semantics of these new elements.

2.2. Context Properties in UML

As described in 1.1, model elements can be enriched with context properites. When using UML, context properties are modeled using UML 1.4 tagged values. As with all tagged values, context properites are defined by their `type` – which must be a context property tag. Since `ContextPropertyTag` is derived from `TagDefinition`, each context property tag must be owned by a stereotype[1]. This means that the stereotype of a model element determines what context properties the model element may have. The context property tag also has a *value validation*, a constraint which further defines what combination of context properties are valid. The following example will illustrate these relations:

1. The stereotype <<comp spec>> has the base class ‘class’, so it can only be used on classes.

[1]Although the UML 1.4 metamodel allows tag definitions not to have a stereotype, the UML 1.4 specification explictly discourages this. Tag definitions not bound to a stereotype are only permitted for UML 1.3 compatibility reasons.

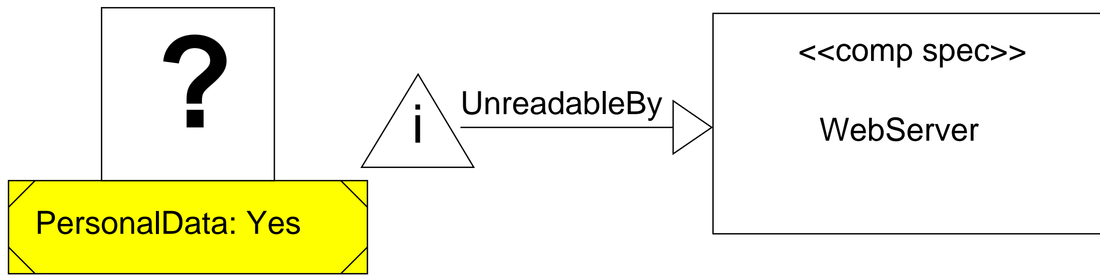


Figure 2.3.: Context-based constraint diagram

2. This stereotype has a defined tag (a context property tag) with name='Tier'
3. This context property tag has the tagType='{"Business Process", "Internet", "unknown"}' and multiplicity='*' (a component may be in more than one tiers).
4. This context property tag also has a value validation which specifies that the value 'unknown' may not be used in combination with the other possible values (if the tier is unknown, no other value may be specified).

All of this specifies that classes with the stereotype <<comp spec>> (1) *may* have a context property 'Tier' (2). This context property can have any number of values out of {"Business Process", "Internet", "unknown"} (3), but if the value 'unknown' is used, it must be used alone (4).

2.3. Context-based Constraints in UML

CCL introduces a new type of constraint, the context-based constraint (CoCon). Each CoCon consists of constrained elements, scoped elements and a constraint body. The constrained elements and the scope elements may either be specified directly (through a reference) or indirectly (through a context condition).

Figure 2.3 shows a CoCon which specifies that any model element with the context property 'Personal Data' with the value 'Yes' may not be readable by the class 'WebServer'. This example illustrates the power of CoCons: the constrainted elements are specified indirecty by the context condition 'Personal Data: Yes'.

Context-based constraints have both a textual and a graphical notation. The details of these notations can be found in [Bübl2001-CCL]. A new type of diagram, the *constraint diagram*, shows the context-based constraints in their graphical form.

3. Software Design Patterns

> All well-structured object-oriented architectures are full of patterns. Indeed, one of the ways that I measure the quality of an object-oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects. Focusing on such mechanisms during a system's development can yield an architecture that is smaller, simpler, and far more understandable than if these pattern are ignored.
>
> Grady Booch [Gamma+95, Forward]

Is this aspect, ArgoUML is no exception, making extensive use of various design patterns. The following sections give short descriptions of some of the design patterns which have be used in ArgoUML.

The diagrams are not identical to those in [Gamma+95] – they are now in UML notation and reflect certain aspects of the Java language (e.g. interfaces).

3.1. Observer Pattern

The observer pattern [Gamma+95, p. 293] is one of the most useful design patterns. It allows one object (the *subject*, also known as *observable*) to notify other objects (the *observers*, also called *listeners*) without making assumptions about who these objects are or how many exist. Through this pattern, there is an abstract coupling between subject and observer – the subject has a set of observers, each realizing a simple interface. The observers need no knowledge of each other and the subject is not dependent on the concrete observers.

The observers have to register themselves with the subject – from this point on (or at least until they are unregistered) they receive notification whenever the state of the subject changes. Once being notified, the observers may act by inspecting the subject and reacting accordingly.

3.2. Adapter Pattern

The adapter pattern [Gamma+95, p. 139] can be used to convert the interface of a class (or an entire class library) into another interface the clients expect. This allows classes to work together even if they have incompatible interfaces.

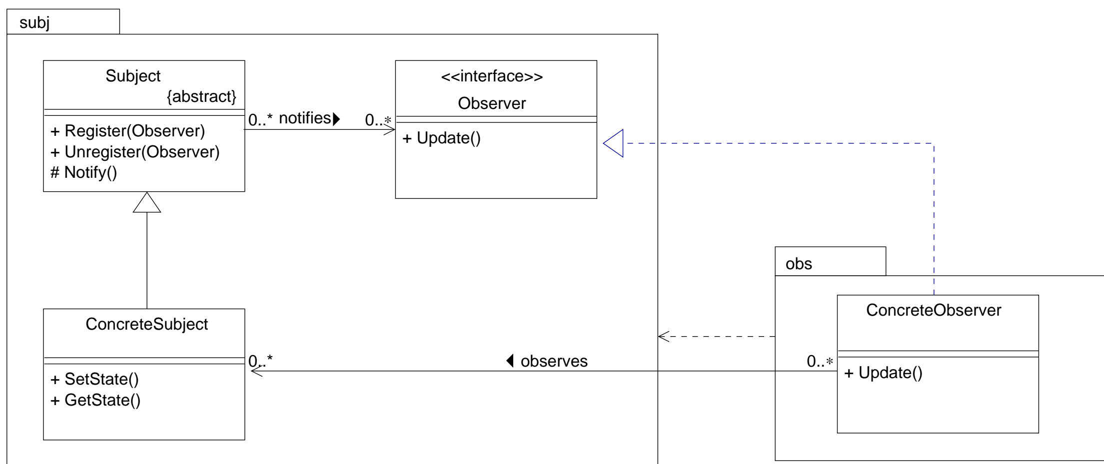


Figure 3.1.: Observer pattern

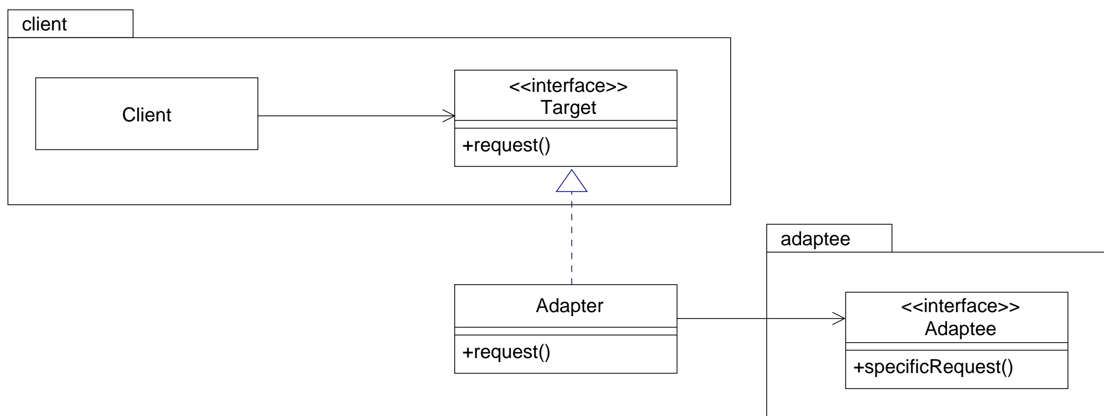


Figure 3.2.: Adapter pattern

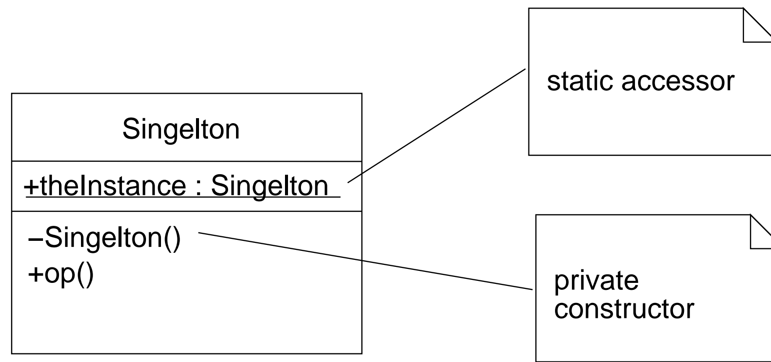


Figure 3.3.: Singelton pattern

Clients can thus be used with unforseen classes – avoiding depencendies between client and adapted class (*Adaptee*). The *Adaptor* translates and forwards calls from the client to the Adaptee.

3.3. Singelton Pattern

Sometimes it is important that a class has only one instance. The singelton pattern [Gamma+95, page 127] is a way to ensure that the class is instatiated only once and all client access this instance from a well-known access point.

This is achieved by declaring the constructor as private, preventing any other class from creating instances. The only access to this class is through a static member which creates the single instance upon initialization.

3.4. Abstract Factory and Factory Method Patterns

These creational patterns allow objects to be created without their concrete classes having to be specified.

A framework could, for example, use interfaces to specify a class hierarchy. The framework can create new products by using the abstract factory and access these products through their interfaces.

The application implements the factory methods to create the concrete products. Although the framework is able to create new products, it needs no knowledge of the application-specific classes.

3.5. Composite Pattern

The composite pattern [Gamma+95, p. 163] can be used to represent part-whole hierarchies. Clients can treat individual objects and compositions uniformly.

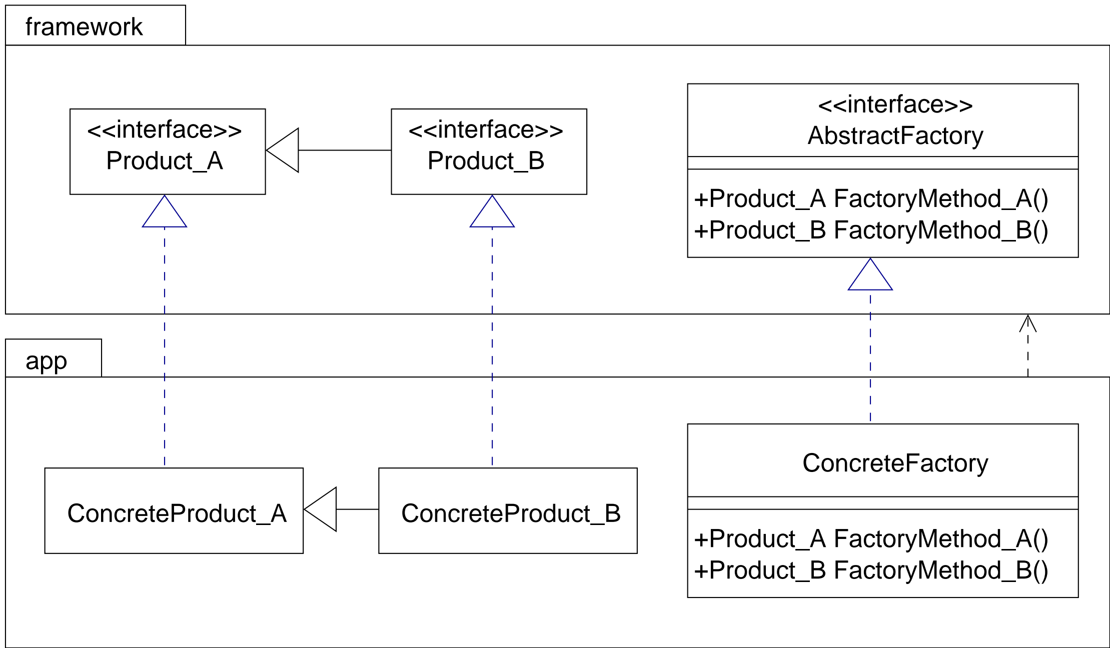


Figure 3.4.: Abstract factory and factory method patterns

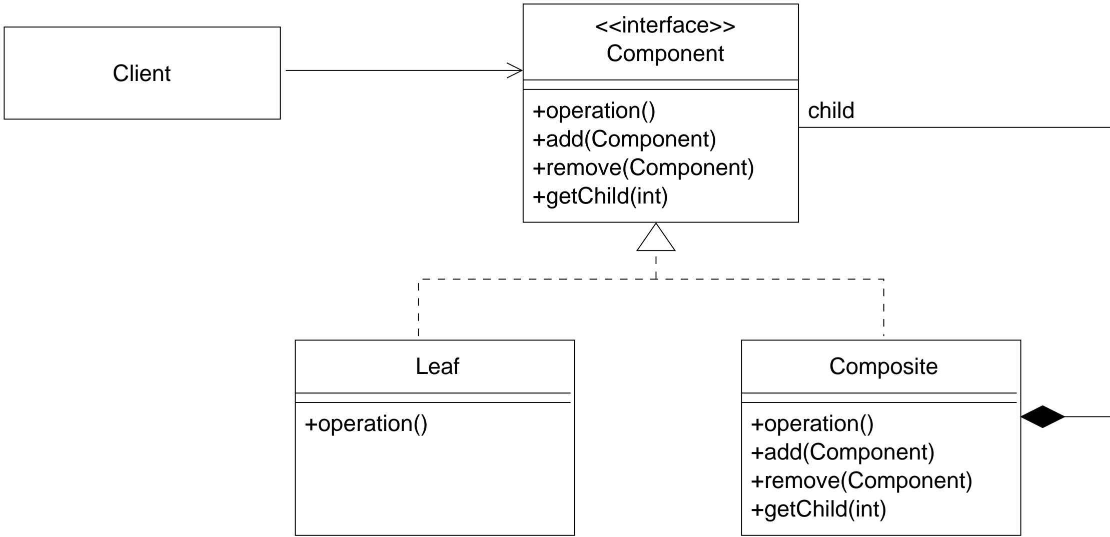


Figure 3.5.: Composite pattern

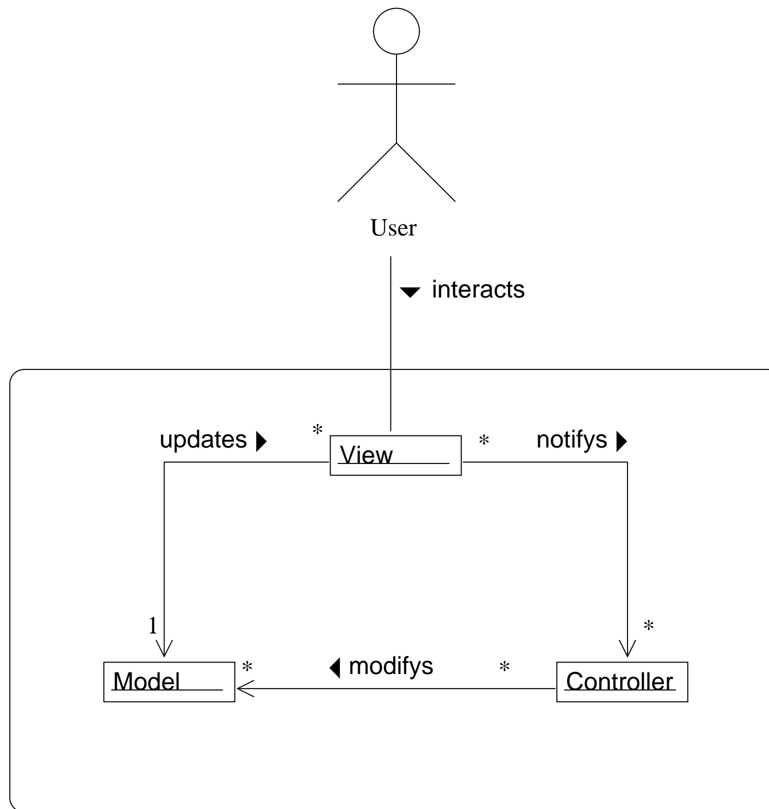


Figure 3.6.: Model-View-Controller pattern

The interface `Component` represents both single and composite objects. Clients use the `Component` interface to interact with objects in the composite structure. If the recipient is a leaf, the call is processed directly. If the recipient is a composite, it recursivly forwards the call to all its children. If there is some functionality common to both leaves and compostites, it may be advantageous to implement `Component` as an abstract class.

3.6. Model View Controller

The Model-View-Controller (MVC) pattern decouples the visual representation of graphs from their data stucture. In the MVC pattern, the user interface (view and controller) is seperated from the data to be displayed and manipulated (model). A single model may have multiple views associated with it. In the case of a spreadsheet, the model may be a two dimensional array of values. A bar chart, a pie chart and a table with rows and cells are three views visualising the model in three different ways.

Views are associated with controllers that modify the model as necessary when the user interacts with the view. The controller then calls the model's mutator methods to

modify the model. The model then notifys *all* views of the change (using the observer pattern) so they always reflect the current state of the model.

4. XML – The eXtensible Markup Language

The eXtensbile Markup Language (XML) is the language of choice when describing and encoding structured data. XML is not a 'fixed' format like HTML (which is a predefined markup language), but can be considered a 'meta-language' which describes and specifies customized markup languages (*XML applications*, sometimes referred to as *document types*). These definitions are specified by Document Type Definition (DTD) files. There are already thousends of XML applications, some examples are:

XHTML is a modified form of the Hypertext Markup Language (HTML). The modifications are necessary since some of the HTML syntax elements do not conform to the XML syntax rules

XMI is a set of document types used to describe models. One example of an XMI-DTD is the UML-DTD.

MOF is part of the XMI specification and is used for describing metamodels.

SVG is a format for scalable 2D vector graphics

4.1. XML and Metadata

According to the XML Metadata Specification [XMI1.1], any MOF level 2 metamodel (such as UML) can be encoded in XML with the MOF-DTD. XMI also specifies how to generate a XMI-DTD from a MOF document. The resulting DTD describes how a model (M1) should be structured when encoding in XMI. One example of this is the UML-DTD, which is included in the XMI specification.

In XMI-DTDs each class, attribute and role in the metamodel is represented by its own *XML-element*, for example: the attribute `name` of metaclass `ModelElement` is specified as:

```
<!ELEMENT Foundation.Core.ModelElement.name
  (#PCDATA | XMI.reference)*
>
```

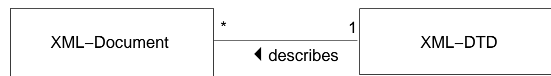


Figure 4.1.: Relationship between XML and DTD documents

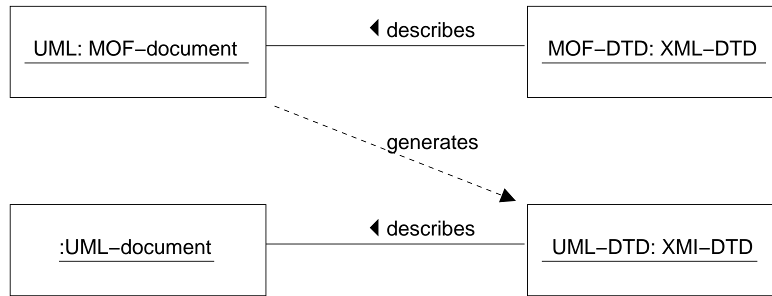


Figure 4.2.: Relationship between MOF and XMI

This defines a new XML-element named 'Foundation.Core.ModelElement.name', which contains parsed character data (i.e. text) or an XML-element named 'XMI.reference'.

In an XML document, elements have a starting tag, their contents, and a closing tag. An XMI document conforming to this DTD could contain:

```
<Foundation.Core.ModelElement.name>
Customer
</Foundation.Core.ModelElement.name>
```

In some cases, the DTD makes use of *XMI-attributes*:

```
<!ELEMENT Foundation.Core.ModelElement.visibility
  EMPTY
>
<!ATTLIST Foundation.Core.ModelElement.visibility
  xmi.value ( public | private | protected )
  #REQUIRED
>
```

This specifies that the XML-element should have no contents (EMPTY), but has one XMI-attribute named 'xmi.value' which should have either 'public', 'private' or 'protected' as a value. XML attributes are included in the opening tag. In the XMI-Document, this would be:

```
<Foundation.Core.ModelElement.visibility
xmi.value="protected" >
</Foundation.Core.ModelElement.visibility>
```

Since this XML-element is empty, XML allows the opening and closing tag to be combined. In this shorthand notation, the above example would be:

```
<Foundation.Core.ModelElement.visibility
xmi.value="protected" />
```

5. Unit Testing

Extreme Programming (XP) is a relativly new and quite controversial method for developing software. It consists mainly of twelve programming practices which reinforce each other – the shortcommings of the individual practices are compensated by the strengths of the others. According to [Beck2000, p. 69]:

> 'Any one practice doesn't stand well on its own (with the possible exception of testing). They require the other practices to keep them in balance'

XP relies on automated test routines to detect side-effects that can occur when refactoring code. *Functional tests* are written by (or with the help of) the customer to test the requirements of the system, and *unit tests* are written by the programmer to satisfy the programmer that the software does what he thinks it should do. Some of the criteria of unit tests are:

- Test cases must be fully automated, requiring no user interaction.
- Each test case must be isolated, having no dependencies on other tests.
- The test cases should be kept simple, avoiding the necessity of debugging the unit test.
- Tests must be boolean, returning either 'tested passed' or 'test failed'
- The unit tests should be called often – as much as once per build cycle. Since the unit tests are fully automated, it takes almost no effort to execute them.
- If a change in code causes a test to fail, the top priority is to fix the code so that all unit tests pass. A test suite where 99% of the tests pass is still a failure.
- Unit tests which test what a method should do should be written first – then the method should be implemented so that it passes the test.
- Whenever a bug is discovered which is not caught by a test, a new test should be created which isolates the problem. Only then should the code be corrected to pass the test.

Unit tests are traditionally[1] used to test application logic, not user interfaces – although it is possible to test user interfaces a well [Wake2000].

[1]it may not be an old tradition, but it is a tradition nonetheless.

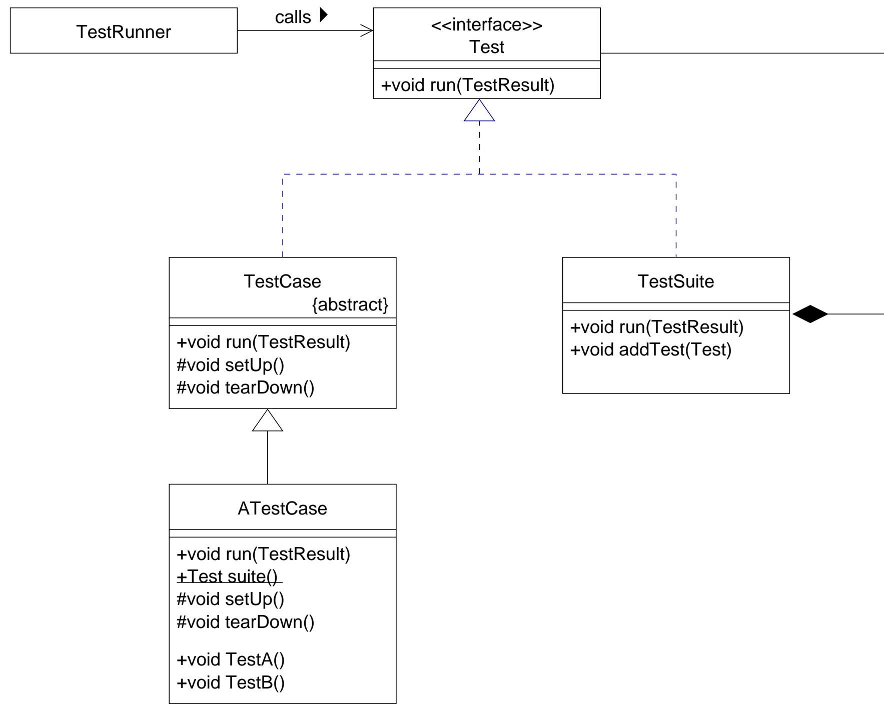


Figure 5.1.: JUnit uses the composite pattern

Unit testing can been seen as a pragmatic compromise between a formal proof of correctness and not testing at all. The goal is to create tests which are likely to fail – and then to write the code which passes the test.

5.1. The Java Unit Testing Framework JUnit

JUnit is a simple, yet flexible unit testing framework written for, and in, java. With this framework, the developer can concentrate on writing the test cases in a straightforword manner – the framework takes care of automatically running the tests and giving immediate feedback to the developer.

The core of the test case architecture is based in the composite pattern (section 3.5). This allows hierarchies of test suites to built during development of the application.

The developer places the concrete test methods in subclasses of `TestCase`. The framework (or more specifically, the `TestRunner` class) expects a static method `suite()` which returns a `TestSuite` instance containing the individual test cases:

```
public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(new ATestCase("testA"));
  suite.addTest(new ATestCase("testB"));
  return suite;
}
```

It is recommended practice to create a test suite for each package in the application. The test suites in higher-level packages should include all the test suites in their subpackages:

```
public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(new ATestCase("testA"));
  suite.addTest(new SubpackageTest.suite());
  return suite;
}
```

The first call to `addTest` adds a single `TestCase` to the suite, the second call adds an entire `TestSuite`. This allows the developer to create a hierarchy of test suites, parallel to the hierarchy of packages.

Part II.

Analysis of ArgoUML

6. The UML Metamodel Library NSUML

The main domain of ArgoUML is, of course, UML models. ArgoUML uses the Novosoft UML API (NSUML), a java implementation of the Unified Modeling Language 1.3 physical specification [UML1.3, Chapter 6]. This specification simplifies the UML 1.3 metamodel – it contains only bidirectional associations, the association classes have been removed, and is more closely aligned to XMI. The following changes were made to the UML 1.3 metamodel:

- spaces in package names changed to underline (e.g. `use_cases`)
- unnamed association ends are named (e.g. `Comment / ModelElement`)
- enumeration literals added as attributes of the enumeration classes for enumeration data types (e.g. `VisibilityKind`: `PRIVATE, PROTECTED` and `PUBLIC`)
- enumeration literal `sorted` added to data type `OrderingKind`
- inheritance link from `Message` to `ModelElement` added[1]
- association class `ElementOwnership` (between `Namespace` and `ModelElement`) removed. Attributes `visibility` and `isSpecification` moved to `ModelElement`
- association class `ElementResidence` changed to class `ElementResidence`, association between `Component` and `ModelElement` changed accordingly[2]
- association class `ElementImport` changed to class `ElementImport`, association between `Package` and `ModelElement` changed accordingly
- association class `TemplateParameter` changed to class `TemplateParameter`, association between `ModelElement` and `ModelElement` changed accordingly

The *Novosoft UML metamodel* follows the UML physical metamodel very closely. Several change were necessary to allow the metamodel to be mapped to the Java language – most of the changes resolve naming conflicts:

1. Renamed `partition` to `partition1` in association `contents (ModelElement) / partition (Partition)`

[1] according to [UML1.3, figure 2-17] `Message` is already a subclass of `ModelElement` in the metamodel

[2] according to [UML1.3, figure 2-8], the association class is named `Element`. In the following text, as well as in [UML1.3, figure 2-29], it is referred to as `ElementOwnership`, so figure 2-8 is probably incorrect.

2. Renamed `collaboration` to `collaboration1` in association `constraingElement (ModelElement) / collaboration (Collaboration)`
3. Renamed `classifierRole` to `classifierRole1` in association `availableContexts (ModelElement) / collaboration (Collaboration)`
4. Renamed `elementImport` to `elementImport2` in association `modelElement (ModelElelemt) / elementImport (ElementImport)`
5. Changed multiplicity of binding end of association `argument (ModelElement) / binding (Binding)` from 0..1 to *, allowing a `ModelElement` to be an argument in several templates[3]

The Novosoft UML API contains four differerent groups classes representing the UML types and metaclasses:

Primitives are UML data types with the stereotype `<<primitive>>`. These are mapped directly to java classes.

Enumerations are UML data types with the stereotype `<<enumeration>>`.

Datatypes are UML data types with no stereotypes.

Elements are UML metaclasses constituent of a UML model

The Novosoft UML API also contains auxiliary classes which provide event notification support and undo/redo support.

6.1. Primitives

NSUML maps UML data types with the stereotype `<<primitive>>` directly to java types and objects[4].

6.2. Enumerations

NSUML realizes UML data types with the stereotype `<<enumeration>>` as final classes with only private constructors. The names of the classes correspond to the UML name, prefixed with a capital M.

For each enumeration literal, public static final instances are created on initalization. There will be exactly one (immutable) instance for each literal for the entire lifetime of

[3]Without this workaround, the templates bindings `set<int>` and `list<int>` would not be possible, `int` being an argument in more than one binding. This will no longer be an issue in UML 1.4 [UML1.4draft, Chapter 2] - the UML metamodel will include an additional metaclass `TemplateArgument`, removing this restriction.

[4]The UML data type `Boolean` is actually an enumeration ([UML1.3, figure 2-11]). NSUML treats it as a primitive

UML primitives	Java Type
Boolean	boolean
Name	String
Integer	int
UnlimitedInteger	int
LocationReference	String
Geometry	String

Table 6.1.: UML primitives and NSUML types

UML enumeration	NSUML Class
AggregationKind	MAggregationKind
CallConcurrencyKind	MCallConcurrencyKind
ChangeableKind	MChangeableKind
MessageDirectorKind	MMessageDirectorKind
OperationDirectionKind	MOperationDirectionKind
OrderingKind	MOrderingKind
ParameterDirectionKind	MParameterDirectionKind
PseudostateKind	MPseudostateKind
ScopeKind	MScopeKind
VisibilityKind	MVisibility

Table 6.2.: UML enumerations and NSUML classes

UML data type	NSUML class
Expression	MExpression
ActionExpression	MActionExpression
ArgListsExpression	MArgListsExpression
BooleanExpression	MBooleanExpression
IterationExpression	MIterationExpression
MappingExpression	MMappingExpression
ProcedureExpression	MProcedureExpression
TimeExpression	MTimeExpression
TypeExpression	MTypeExpression
Multiplicity	MMultiplicity
MultiplicityRange	MMultiplicityRange

Table 6.3.: UML datatypes and NSUML classes

UML element	NSUML interface	NSUML class
Package	MPackage	MPackageImpl
Class	MClass	MClassImpl
Attribute	MAttribute	MAttributeImpl
...	...	...

Table 6.4.: UML elements, NSUML interfaces and classes

the system[5], ensuring that identity comparisons and equality comparisions will always have the same result.

6.3. Datatypes

Datatypes have no stereotype. Each datatype is mapped to exactly one NSUML Java class

The class `MMultiplicity` has four predefined instances for the most common UML multiplicities: `M0_1`, `M1_1`, `M0_N` and `M1_N`. These instances are defined as static class members (accessable as `MMultiplicity.M0_1`, etc.).

6.4. Elements

UML elements are structured in packages (foundation, core, behavior, etc). Each element is mapped to exactly one NSUML interface and one NSUML class.

The NSUML class `MBaseImpl` is the abstract superclass of all element classes – likewise, all NSUML interfaces are derived from the interface `MBase`.

[5]this is similar to the singelton pattern (3.3) where a class has exactly one instance

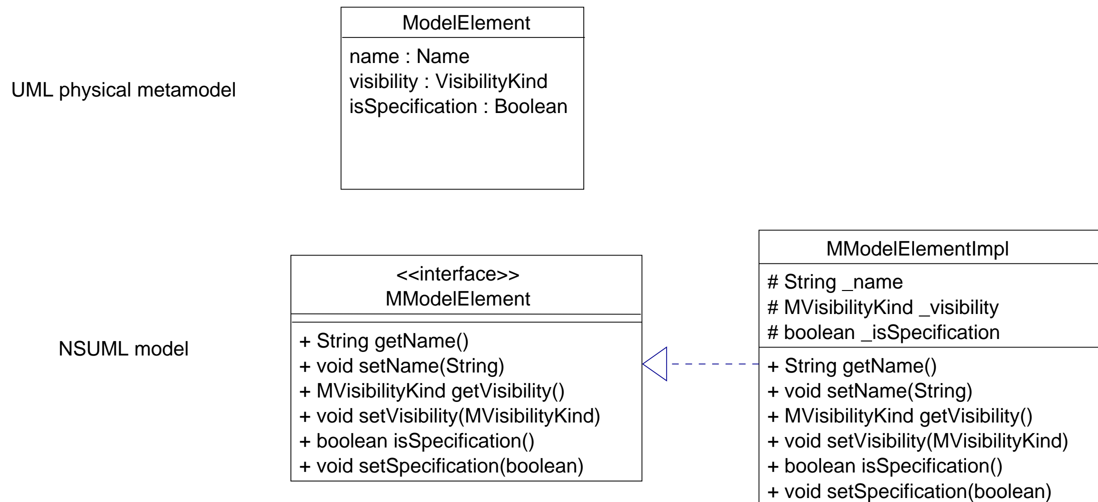


Figure 6.1.: UML metaattributes and their corresponding NSUML methods

An application should always refer to an object by its interface – never directly by its class:

```
MBase cls0 = new MClassImpl();      // ok
MClass cls1 = new MClassImpl();     // also ok
MClassImpl cls2 = new MClassImpl(); // avoid this!
```

6.5. Accessing and modifying metaattributes

Element metaclasses contain *metaattributes*, which can be one of the primitive, enumeration or datatype classes.

Each metaattribute is mapped to a member variable and should only be accessed by the corresponding inspector (`get`*`Attribute`* or `is`*`Attribute`*) and mutator (prefix `set`*`Attribute`*) methods.

6.6. Accessing and modifying metaassociations

Two element metaclasses can be linked to each other with *metaassociations*. Every metaassociation has a *association role* at each end. Depending of the multiplicity of the role, different methods are used to access the association

Reference roles have a multiplicity of 0..1 or 1. They are accessed similarily to attributes with `get`*`Rolename`* and `set`*`Rolename`* methods.

Bag roles are unordered with a multiplicity different from 0..1 and 1. The `get`*`Roles`* method returns a collection of all the references in the role. Note that this collection is a copy of the internal collection and can not be changed. With `set`*`Roles`* all

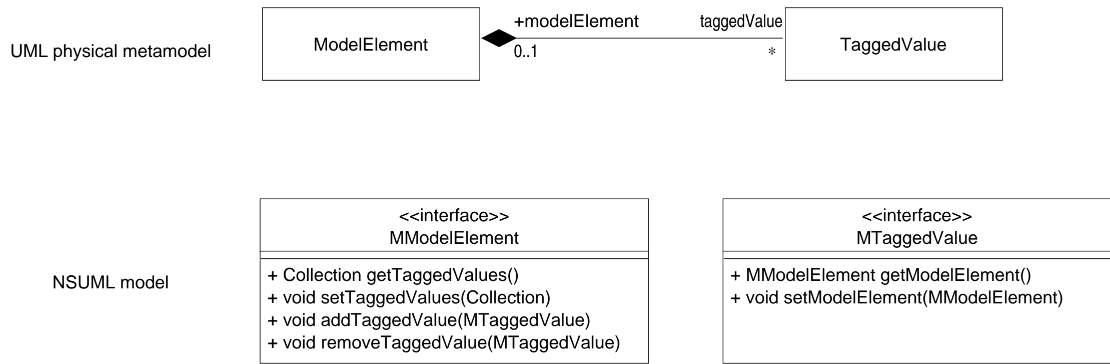


Figure 6.2.: UML metaassociations and their corresponding NSUML methods

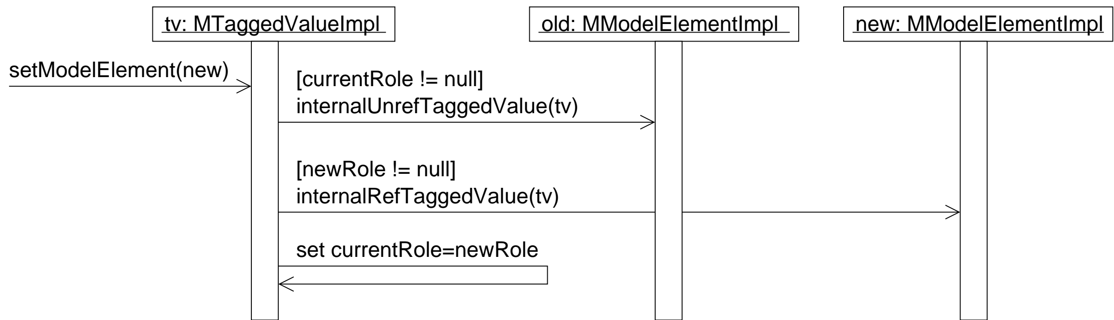


Figure 6.3.: Setting an assocation from the reference role end

the roles can be set at once. The `add`*Role* and `remove`*Roll* methods allow adding and removing single associations.

List Roles are ordered with a multiplicity different from 0..1 and 1. In addition to the methods bag roles offer, they also define `add`*Role*`(int,` *OppositeRole*`)`, `re-move`*Role*`(int)`, `set`*Role*`(int,` *OppositeRole*`)` and `get`*Role*`(int)` methods to allow accessing the roles at specific positions in the list.

A link between two objects can be added to removed with a single method call to *either* side. The status of the opposite object is implicitly updated:

```
MTaggedValue tv;
MModelElement me;
tv = new MTaggedValueImpl;
me = new MModelElementImpl;
tv.setModelElement(me);   // (1) adds an association
me.removeTaggedValue(tv); // (2) removes it
```

In this example, line (1) not only modifies the status of `tv`, but also of `me`.

When modifying a reference role (with `set`*Role*), the object first checks whether a link already exists. If this is the case, it tells the existing opposite end to remove the link

<<interface>>
MBase

+ Object reflectiveGetValue(String)
+ void reflectiveSetValue(String, Object)
+ void reflectiveAddValue(String, Object)
+ void reflectiveRemoveValue(String, Object)
+ Object reflectiveGetValue(String, int)
+ void reflectiveSetValue(String, int, Object)
+ void reflectiveAddValue(String, int, Object)
+ void reflectiveRemoveValue(String, int, Object)

Figure 6.4.: NSUML reflective API methods

reflective method	valid for feature
Object reflectiveGetValue(String)	all
void reflectiveSetValue(String, Object)	all
void reflectiveAddValue(String, Object)	bags and lists
void reflectiveRemoveValue(String, Object)	bags and lists
Object reflectiveGetValue(String, int)	only lists
void reflectiveSetValue(String, int, Object)	only lists
void reflectiveAddValue(String, int, Object)	only lists
void reflectiveRemoveValue(String, int, Object)	only lists

Table 6.5.: NSUML reflective methods

with a call to `internalUnref`*`Role`*. If the call to `set`*`Role`* is meant to create a link (as opposed to clearing it by passing null as a parameter), it then tells the (new) opposite end to create the link with `internalRef`*`Role`*.

When setting a complete bag role or list role the difference between the old set of links and new set of links is determined. Any links which are in the old set but not in the new set are removed – resulting in (possibly) several calls to `internalUnref`*`Role`* to update the state at the opposite end of the association. Then any new links are added (again calling `internalRef`*`Role`* to update the opposite end).

6.7. NSUML reflective API

NSUML offers a orthogonal methods for accessing *features* (metaattributes and metaassociations) by name. These reflective methods are slower than the direct methods, but are often easier to use.

Depending on the type feature, not all reflective methods may be valid:

Each implementation class overrides the appropriate reflective methods. If the feature parameter is known to the class, it reacts to the invocation by setting or getting the feature. If the feature is unknown, the call is passed to the superclass for further handling. If, after traversing the hierarchy, no superclass recognized the feature, an `IllegalArgumentException` is throw by `MBaseImpl`.

event generated	MEventListener method called
ELEMENT_REMOVED	removed
ELEMENT_RECOVERED	recovered
ATTRIBUTE_SET	propertySet
REFERENCE_SET	propertySet
BAG_ROLE_SET	propertySet
BAG_ROLE_ADDED	roleAdded
BAG_ROLE_REMOVED	roleRemoved
LIST_ROLE_SET	propertySet
LIST_ROLE_ADDED	roleAdded
LIST_ROLE_ITEM_SET	listRoleItemSet
LIST_ROLE_REMOVED	roleRemoved

Table 6.6.: NSUML generated events

6.8. NSUML event notification

NSUML is able to generate events whenever the model is modified. This allows the application to react to any changes in the model.

To receive events, the application define a class realizing the `MEventListener`[6] interface. `MBase` defines methods for registering and unregistering observers. `MEventListener` must be registered with each element to be observed.

Depending on what mutator method was called, different events are sent to different observer methods:

These events are also generated by the respective reflective API methods.

The generation of events is controlled by the `MFactoryImpl` class. Initially, no events are send to the observers. By calling `MFactoryImpl.setEventPolicy` event generation may be enabled.

Whenever a mutator method (whether for attributes or roles) changes the state of an object, it generates an event by one of the followng methods:

- `fireAttrSet`, `fireBagSet`, `fireListSet` when a set* mutator is called
- `fireBagAdd`, `fireListAdd` when an add* mutator is called
- `fireBagRemove`, `fireListRemove` when a remove* mutator is called
- `fireListItemSet`, when setting a single link of a list role directly

This does not immediately notify any registered listeners – the events are collected at first. Every mutator method begins with `operationStarted()` and ends with `operationFinished()`. In this way, the `MFactoryImpl` can keep track of the operation nesting

[6]this conforms to the observer interface in the observer pattern (3.1)

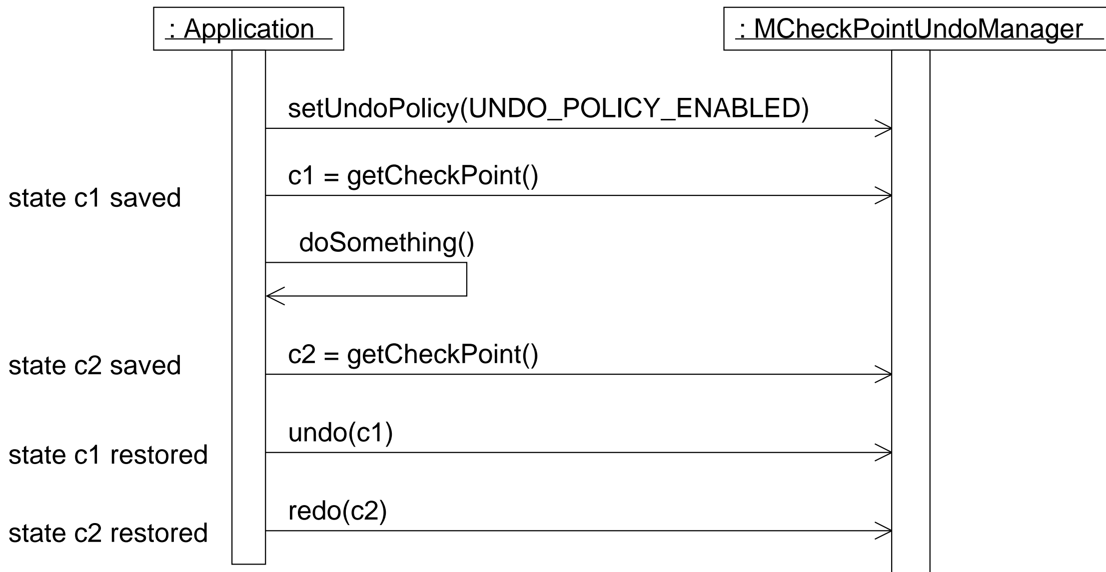


Figure 6.5.: NSUML undo/redo support

depth. When the depth = 0 (all operations have finished) all collected events are send to the listeners.

6.9. NSUML undo/redo support

NSUML offers undo/redo support through the use of *checkpoints*. A call to `MCheckPointUndoManager.getCheckpoint()` creates a `MCheckpoint` instance, and the current state is stored on a stack. A subsequent call to `undo (MCheckpoint)` restores the model to its previous state. The undo operation may be revoked with a call to `redo (MCheckpoint)`.

Whenever a mutator is called, the operation necessary to restore the original state of the object (compensation operation) must be saved onto an "undo"-stack. This is achieved by use of the `method` object (package `java.lang.reflect`), allowing methods to by handled as objects.

The compensation methods of each class are stored as `private static` members variables:

Attributes `_`*`attribute`*`_setMethod`

Reference Roles `_`*`role`*`_setMethod`

Bag Roles `_`*`role`*`_setMethod`, `_`*`role`*`_addMethod` and `_`*`role`*`_removeMethod`

List Roles `_`*`role`*`_listSetMethod` in addition to the members specified for Bag Roles

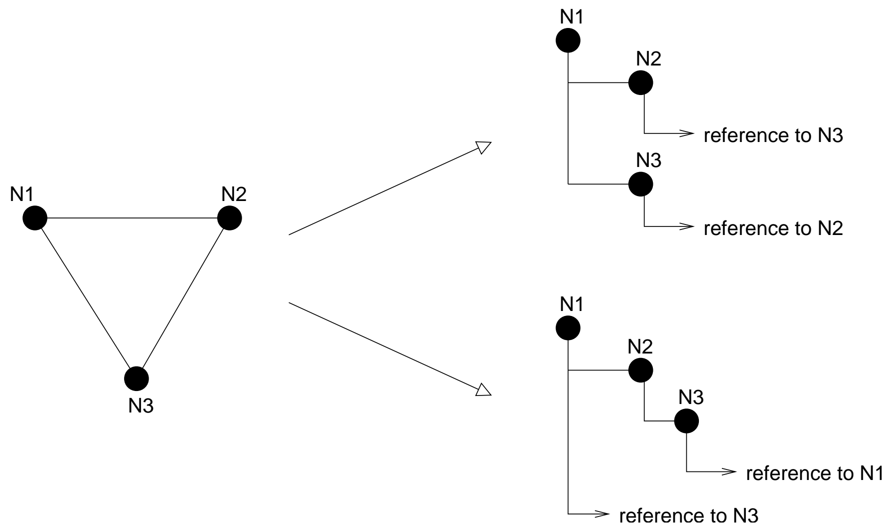


Figure 6.6.: A graph may have many equivalent tree representations

Each mutator method then logs the appropriate compensation method and parameters necessary to undo the operation:

```
public class TaggedValueImpl {
  public void setModelElement(MModelElement new)
  {
    // ...
    logRefSet(_modelElement_setMethod, cur, new);
    cur = new;
    // ...
  }
```

The call to `logRefSet` saves the compensation operator onto the stack. The undo manager can use this stack to undo or redo the operation as required.

6.10. Model Persistance

The ability to save and restore the user's model is a necessary function of any modelling tool. NSUML supports model persistance closely aligned to the OMG's XML Metadata Exchange (XMI) specification.

The class `ru.novosoft.uml.xmi.XMIWriter` (derived from `java.io.Printwriter`) is responsible for encoding the UML model as an XMI text stream. XMI documents (and all XML documents) consist of elements which may contain further elements – forming a treelike structure. Since the model is a graph and not necessarily a tree, simply recursivly traversing the model and serialising each (model) element could result in one element being serialized more than once. This would then result in troublesome duplicates when the model is restored from the XMI document.

To avoid this problem, the `XMIWriter` must transform the graph into a semantically equivalent tree. The method `XMIWriter.gen` first assigns each element a temporary, unique id (`xmiid`). The first time an element is serialized, this id, as well as the complete state of the element, is written to the stream. If the same element is encountered again while traversing the graph, only a reference to that element (its id) is written. The hash map `processedElements` keeps track of which elements have already been serialized. Note that depending on how the model is traversed, different, semantically equivalent trees may result.[7]

Parsing the XMI document and recreating the model is done by the `ru.novosoft.uml.xmi.XMIReader` class. The XML tree can simply be traversed and each element is created and its state restored as required. The ids are again stored in a hash map, so any ids occuring later in the document can be dereferenced to recreate the entire model.

[7]This must be considered when creating the test cases. Comparing a model with a pre-computed tree may result in a false negative if the traversing algorithem is changed. A better approach would be to transform the model to a tree and then transform the tree back to a model. If both models are equivalent, the test is passed.

7. The Graph Editing Framework Library GEF

The Graph Editing Framework (GEF) is a library of java classes which support the visual representation and modification of connected graphs. It is based on the MVC pattern (section 3.6) and contains support for both the view and the controller, and specifies a set of interfaces to access the model. ArgoUML uses this framework to display the UML diagrams.

7.1. GEF View Architecture

In MVC the view is responsible for visualising the model to the user. GEF assumes the model is a connected graph, although the class hierarchy could be extended to support other models. A connected graph consists of nodes with ports, and edges may connects ports with each other.

The class `LayerManager` and its array of `Layers` form the view support in GEF. Layers are like clear sheets of plastic, each containing part of a drawing. Layed on top of each other, they make the overall drawing. They can be hidden, locked or grayed out individualy. `LayerDiagram` is a layer which contains a collection of view elements (class `Fig`).

`LayerPerspective` extends on this by assuming that the model is a connected graph. Each view element represents a node or an edge in the model. The class `LayerPerspective` accesses this model through several interfaces: `GraphModel` supports traversing the graph; `GraphNodeRenderer` and `GraphEdgeRenderer` are class factories for creating displayable `Fig` instances from the model's nodes and edges.

7.2. GEF Controller Architecture

Usually, the user interacts with the application by clicking on buttons, activating tools in toolbars or selecting menu items in popdown menus. In Java/Swing, menu items and buttons are represented by with objects realizing the `Action` interface. Whenever a user selects a menu item oder a toolbar button, the Swing library invokes the `Action.actionPerformed` method. Usually applications make use of the default action implementation, `AbstractAction`, and derive their own actions from this class.

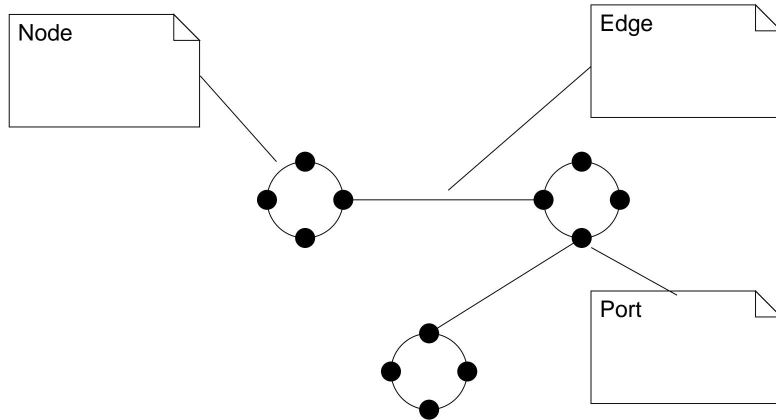


Figure 7.1.: GEF assumes the model is a connected graph

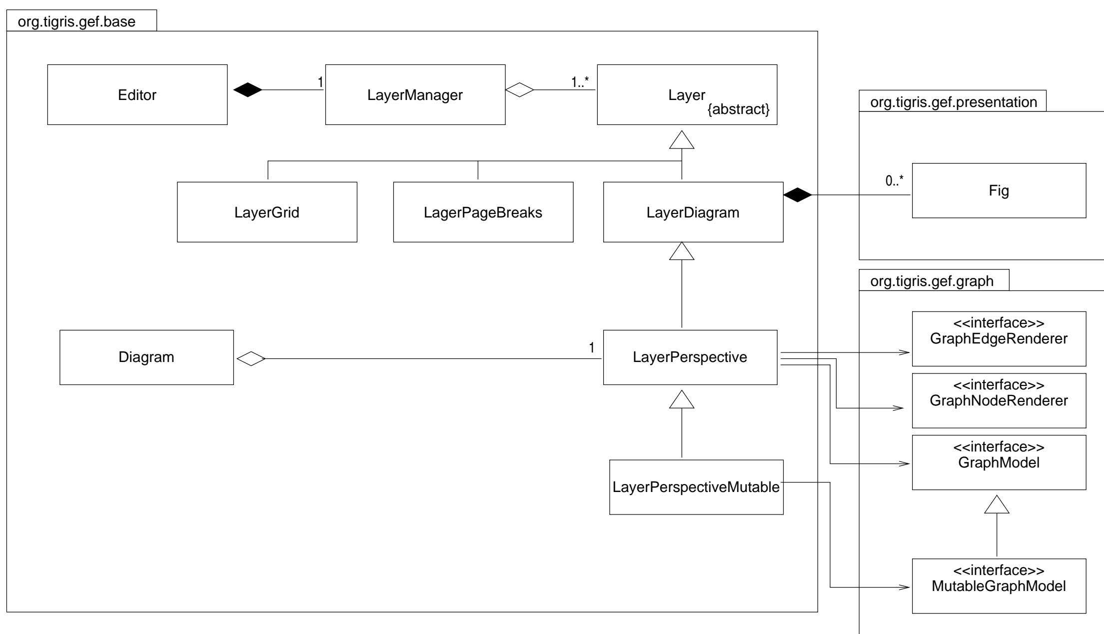


Figure 7.2.: GEF editor architecture

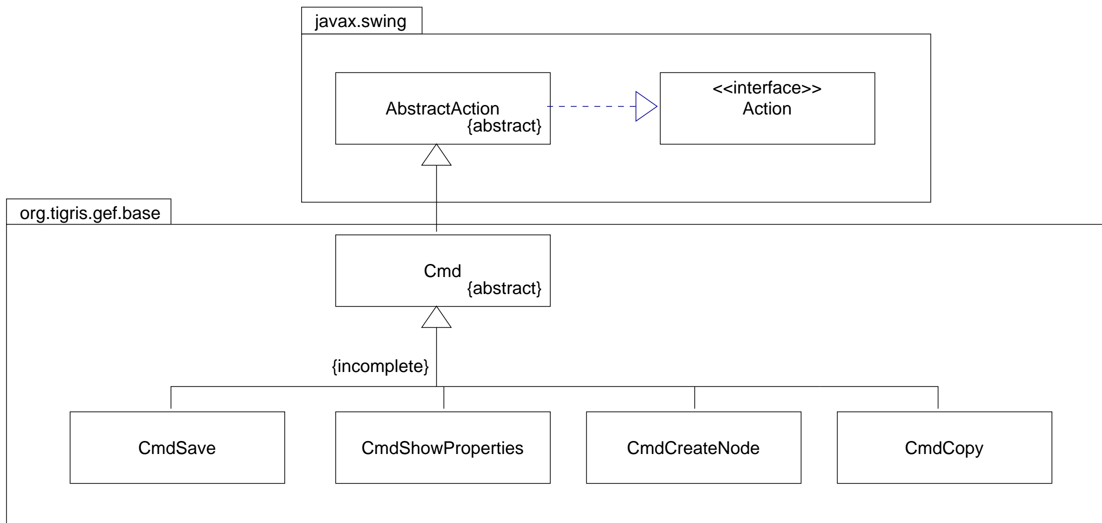


Figure 7.3.: Partial Cmd class hierarchy

7.2.1. GEF command classes

The GEF class Cmd is an abstract superclass for all editor commands, adding command arguments, support for "undo" and an application-global command registry. GEF contains over 40 Cmd subclasses, supplying support for load/save commands, cut and paste commands and commands to add and delete nodes. An application can, of course, add new Cmd-subclasses as required.

7.2.2. GEF Editing Modes

In order to support visual editing, the editor must keep track of which mode (or modes) it is in. How the editor reacts, for example, when the user clicks on the diagram depends on whether the user previously triggered CmdCreateNode to add a new figure to the diagram (ModePlace) or is selecting a figure in order to move it (ModeSelect followed by ModeModify). The class ModeManager keeps track of which modes are currently active[1]. Whenever the editor receives an event (e.g. the user clicks on a figure) the top mode on the stack gets a chance to react to the event.

If the mode does not consume the event the next modes in the stack get a chance to process the event, from top to bottom. This puts all behavior associated with a particular mode into one object. Since all the behaviors are encapsulated in classes realizing the FigModifyingMode interface, new application-specific behaviors can easily be added.

Initialy, the ModeManager's stack contains three modes: ModeDragScroll, ModeSelect and ModePopup. ModeDragScroll allows the user to scroll the editor by clicking

[1]Note that this architecture is based on the state pattern [Gamma+95], allowing an object (in this case, the editor) to alter its behavoir when its internal state changes.

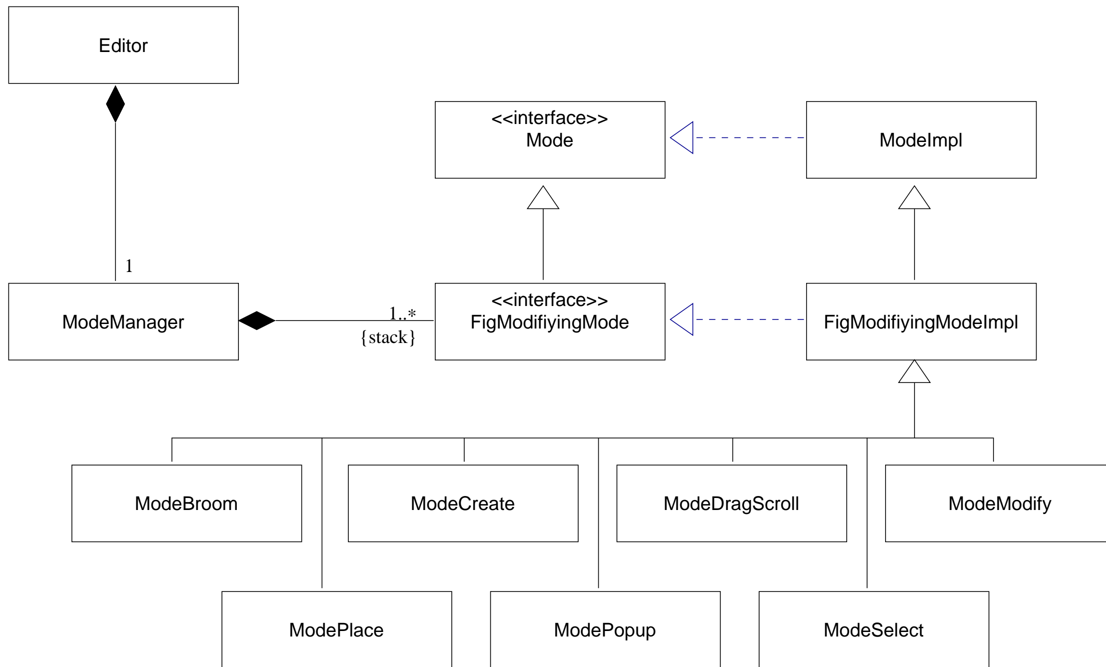


Figure 7.4.: GEF controller architecture

and dragging with the middle mouse button. `ModeSelect` interprets user input as selecting one or more Figs. `ModePopup` catches right mouse button events and shows a popup menu.

If the user drags an existing figure to a new position, the editor goes through two state transitions. First, the user presses the left mouse button while the mouse cursor is over an existing figure, generating a `MouseEvent`. The `ModeManager` forwards the event to each mode in its stack, from top to bottom. `ModeDragScroll` ignores the event, since it only reacts to middle mouse buttons. `ModeSelect` then checks whether the cursor is over an existing figure, notifies the `SelectionManager` (section 7.2.3) that a new figure has been selected, then adds `ModeModify` to the stack before consuming the event – preventing any other modes from processing the left-button-down event. When the user moves the mouse (remember: the mouse button is pressed, so he is actually dragging), `ModeModify` continuously modifies the position of the selected figs. By finally releasing the mouse button, `ModeModify` is removed from the stack, and the editor returns to its normal state.

When adding a new node, the editor has to not only modify the diagram – a new node must also be added to model. The class `LayerPerspectiveMutable` is able pass these modifications to the model through the interface `GraphModelMutable`. Modes which wish to modify the model can get a reference to the current `GraphModelMutable` instance and can then access the models mutators.

When the user activates the appropriate “add node” command, `ModePlace` is pushed onto the stack. As soon as the mouse button is pressed somewhere in the diagram,

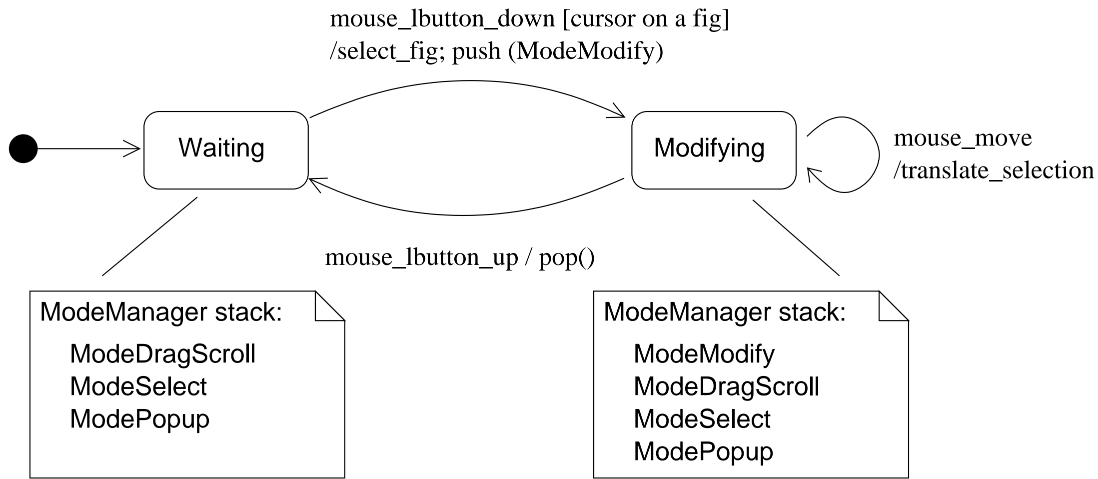


Figure 7.5.: Moving a figure to a new position

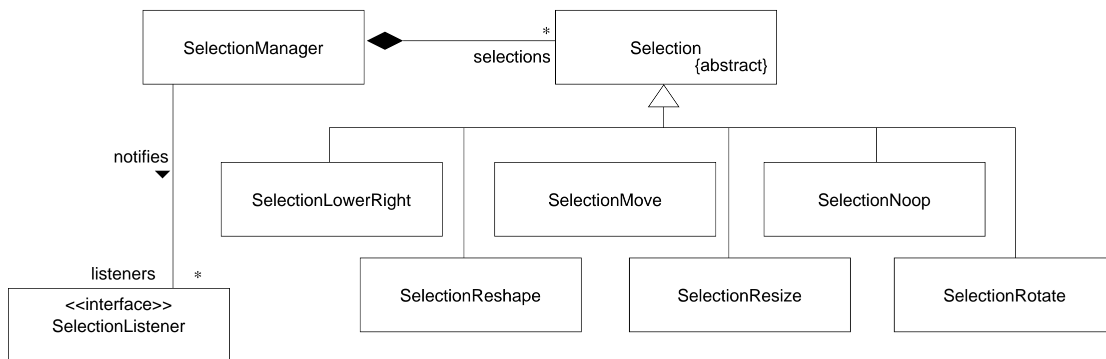


Figure 7.6.: GEF selection classes

`ModePlace` creates the respective `Node` instance as well as a corresponding `Fig`. Once the mouse button is released, the newly created node is added to the model by calling `MutableGraphModel.addNode`.

7.2.3. GEF Selection classes

The set of figures which are currently selected is handled by the `SelectionManager`. Whenever the selection changes, this class also notifies all registered `SelectionListeners` (observer pattern, section 3.1) of the change. This could allow another window to update its contents to stay synchronized with the diagram. The `SelectionManager` also asks the figure whether it has a `Selection` object. This object contains the behavoir of a figure in its selected state.

`Selection` objects handle the display of handles or whatever graphics indicate that something is selected, and they process events to manipulate the selected figure. GEF offers several `Selection` subclasses, which can be extended by the application.

SelectionMove allows the user to move the figure by dragging it, but not to resize it

SelectionNoop does not allow the user do manipulate the figure.

SelectionReshape draws one handle over each point in the figure. This allows the user to reshape the figure by moving the individual points.

SelectionResize shows the user the figure's bounding box with handles allowing the user to resize the figure.

SelectionLowerRight is similar to the SelectionResize class, but only allows resizing by dragging the lower right corner of the boudning box.

SelectionRotate allows the user to rotate the figure.

7.3. Diagram Persistance

In addition to saving and restoring the underlying model (which does not fall under the responsibility of GEF), an application will probably need to save and restore the diagrams – otherwise, the information about what model elements appear in which diagram (at what geometric coordinates) will be lost.

The GEF library currently (version 0.9.x) stores its diagrams as Precision Graphical Markup Language (PGML) documents. PGML is an XML application designed to respresent 2D scalable graphics. GEF's graphic primitives are very similar to the PGML elements with few exceptions:

text objects GEF defines a rectangle with left, right or centered text inside; PGML defines an origin and always writes the text left to right from that point.

groups of shapes GEF uses the `FigGroup` class to represent a group of shapes (composite pattern, section 3.5). The nodes in the graph model are represented by subclasses of `FigNode` which subclasses from `FigGroup`. These subclasses could be more than simply groups of shapes, with their own behavior for resizing and computing text (should the text grow when the shape is expanded or should the line-breaks be recomputed?) PGML does not support this will simply scale the text along with the shape.

Additionaly, every edge and node in a PGML document has a reference to the model element it represents.

The PGML documents are generated by the class `OCLExpander`. A TEE-document (Templates with Embedded Expressions) contains templates for all the java classes used in the diagrams, and can contain an OCL-like expression[2]. The `OCLExpander` evaluates these expression using the current context of the java class being processed, so if the TEE document contains:

[2] only a very small subset of OCL expressions in the form `self.attribute.attribute...` are supported

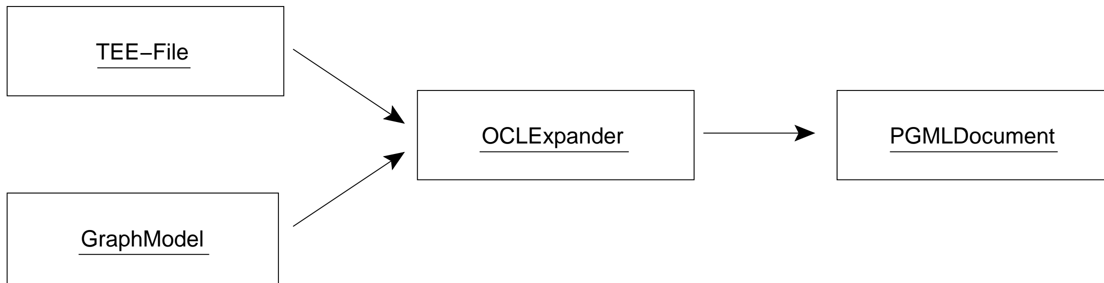


Figure 7.7.: GEF generation of PGML files

```
<template class="org.tigris.gef.presentation.FigEdge">
  classname="<ocl>self.class.name<ocl>"
</template>
```

then any FigEdge objects will be encoded as:

```
classname="org.tigris.gef.presentation.FigEdge"
```

in the output document.

The next major version of GEF will replace PGML with SVG (Scalable Vector Graphics), another XML application, but the template-based approach will probably remain.

8. ArgoUML Architecture

The ArgoUML application consists of three main packages (each containing many subpackages): the Novosoft UML metamodel library (NSUML), the Graph Editing Framework (GEF), and of course, ArgoUML itself. NSUML contains all the classes needed to represent and manipulate UML 1.3 models, GEF is responsible for visualising these UML models as diagrams, and ArgoUML ties all this together and adds the application logic.

The Model-View-Controller architecture is essential for any UML modelling tool. Each project usually contains exactly one UML (user level) model, but contains many views on the model. The various UML diagrams, the form-based property panels, even the navigator pane (which shows the model in a tree-like structure) are all simply different views which visualize the model from different perspectives and in different ways.

In figure 8.2, it should be clear that all three views are views on the same UML model. 'P2' in each diagram represent the same model element, namely the UML package P2.

Figure 8.3 shows the layout of the ArgoUML user interface, which consists of several panels. The *navigator pane* displays the project in one of several perspectives as a tree-like structure, enabling easy navigation. The *multieditor pane* is the main pane of the application containing a diagram-specific toolbar, and of course, the GEF editor pane displaying the active UML diagram. The *details pane* shows the attributes of the currently selected model element. The *todo pane* displays a list design issues which need the user's attention. The *application menu bar* and *status bar* perform the obvious tasks.

The main ArgoUML window is the singelton[1] `ProjectBrowser`, containing classes for each of the windows panes.

8.1. The UML Diagrams

ArgoUML uses the GEF library for the various UML diagram types. As detailed in section 7.1, GEF assumes that the data it visualizes is structured as a graph consisting of nodes and edges.

GEF uses the `GraphModel` interface to access the model. The ArgoUML classes which implement this interface must map the UML metamodel to GEF's node-and-edge structure. In this section, the UML use-case diagram (one of the simpler diagrams) will serve as an example to illustrate the ArgoUML diagram design.

[1] the singelton pattern [Gamma+95] is not used here, so it would be *possible* to create additional instances, although this is not recommended

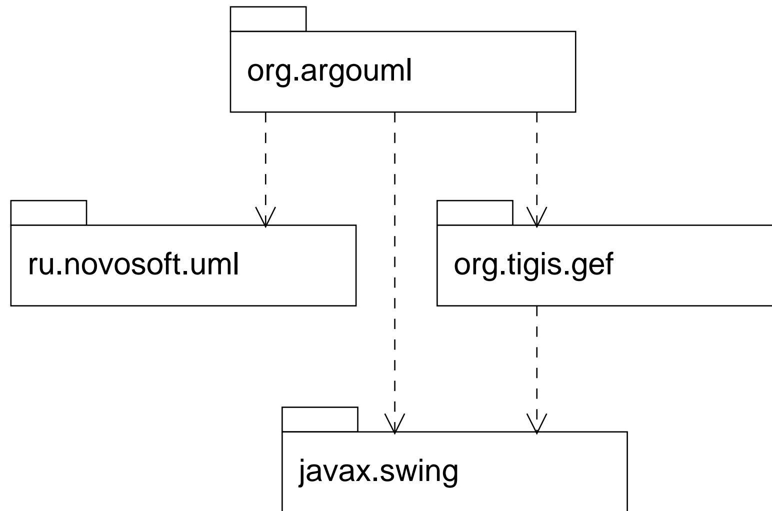


Figure 8.1.: high-level view of the ArgoUML architecture

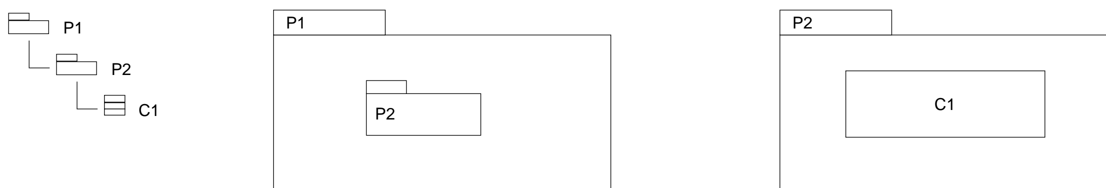


Figure 8.2.: Three views of the same model

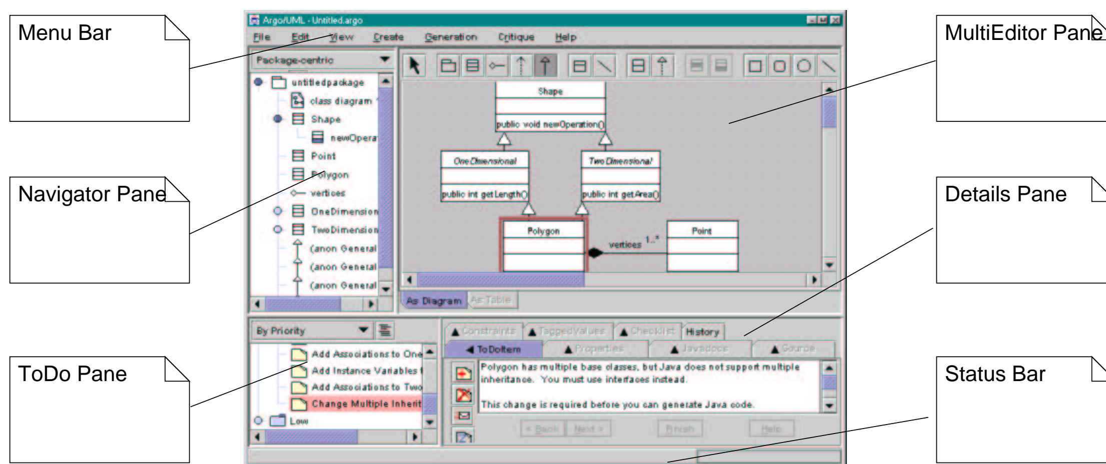


Figure 8.3.: ArgoUML user interface

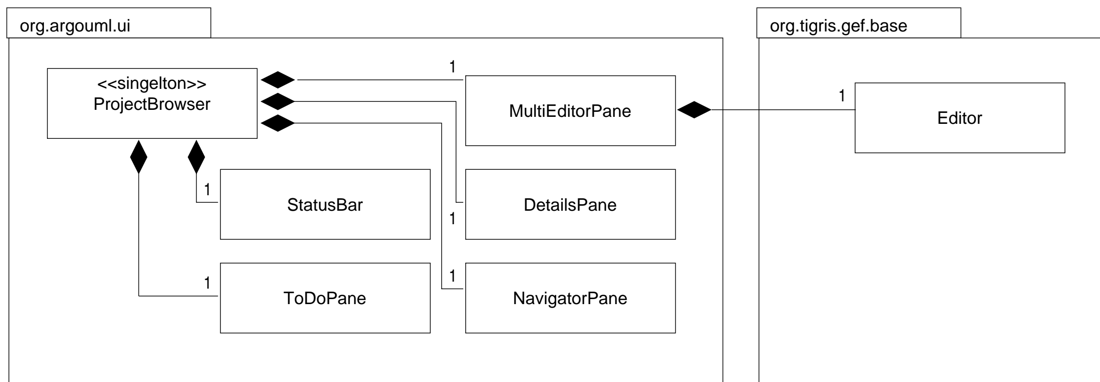


Figure 8.4.: Major ArgoUML user interface classes

UML metaclass	figure class	graph element
MActor	FigActor	node
MUseCase	FigUseCase	node
MGeneralization	FigGeneralization	edge
MDependency	FigDependency	edge

Table 8.1.: UML use-case diagram figures and model elements

A use-case diagram can contain the following model elements: actors, use-cases, dependencies and generalizations. The `UseCaseDiagramGraphModel` must map these metaclasses to GEF figure classes.

As shown in table 8.1, actors and use-cases are considered nodes, since they can exists independently. Generalizations and dependencies are edges which connect the nodes. These figure classes represent the metaclasses graphically in a specific diagram.

Implementing the various UML diagrams (static structure, use-case, sequence, etc.) is suprisingly straightforward. Due to the supporting framework, only a few classes for each diagram type are necessary. Each UML diagram type requires:

1. a class derived from `Diagram`. This class should contain member objects for each command (usually `Cmd`-Subclasses, section 7.2) the user may trigger. ArgoUML supplies the classes `ArgoDiagram` which uses the observer pattern (section 3.1) to add listener registration support allowing the diagram to react to changes in the model, as well as the class `UMLDiagram` which adds support for a UML model (interface `MNamespace` as opposed to the generic `MutableGraphModel`) and commands common to all UML diagrams.

2. a displayable `Fig` subclass for each UML metamodel element which can be inserted in the diagram.

3. a factory class (section 3.4) realizing the `GraphEdgeRenderer` and `GraphNodeRenderer` interfaces which can create the necessary `Fig` instances on demand.

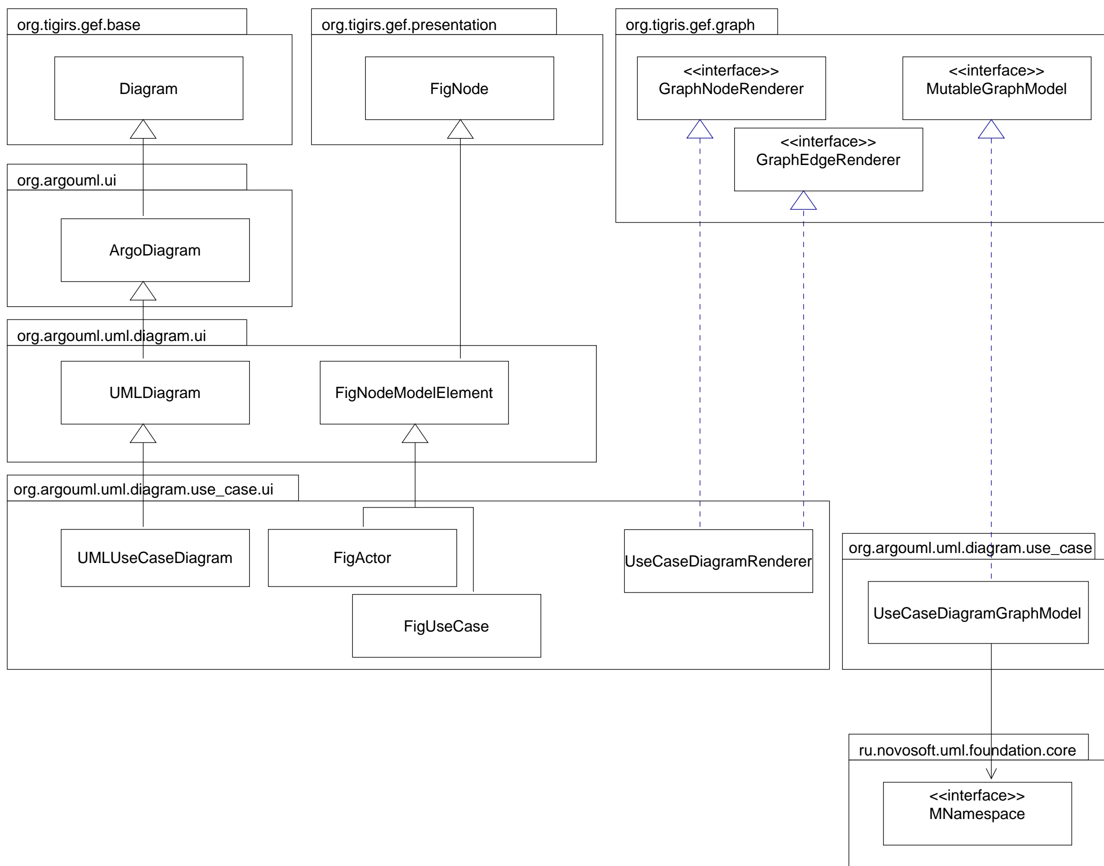


Figure 8.5.: Use-case diagram main classes

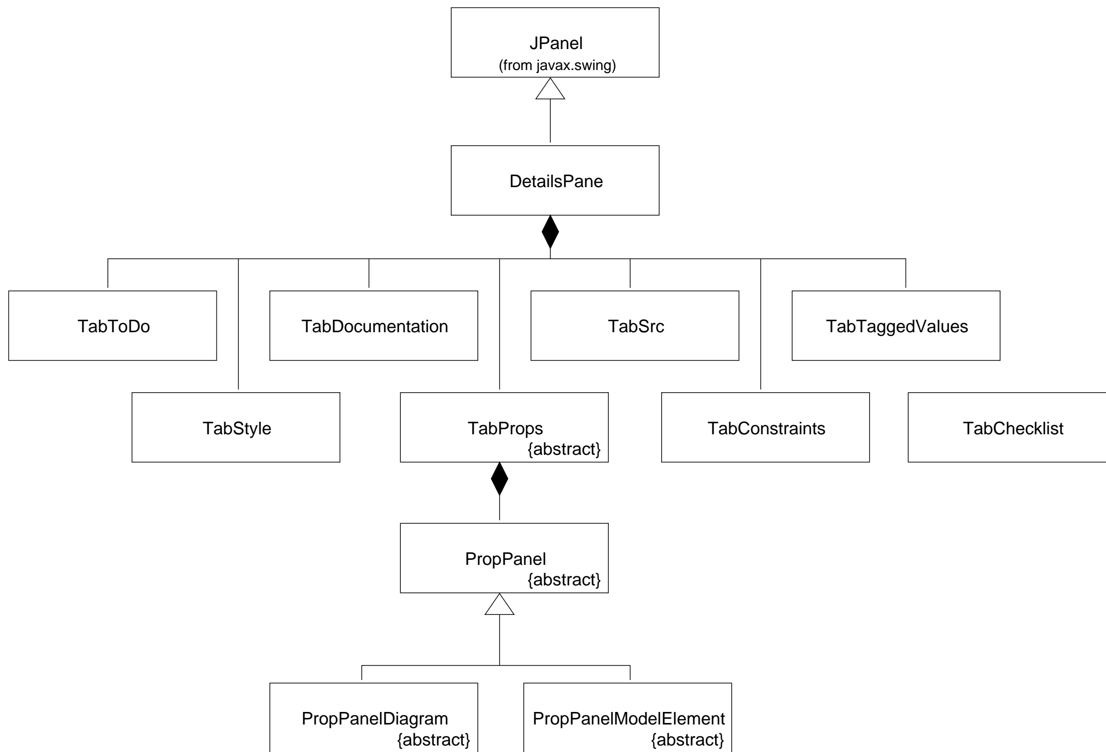


Figure 8.6.: ArgoUML details pane classes

4. a adapter class (section 3.2) realizing `MutableGraphModel` allowing GEF to access and modify the UML model.

Support for each type of UML diagram is encapsulated in its own subpackage hierarchy.

8.2. The Details Pane

The details pane is below on multieditor pane. It consists of several tabs, each of which shows details about the currently selected object. This object can be an element in the user's model, a diagram or a figure. This allows the user to inspect and edit the object's attributes in a form-based manner.

ToDo Tab is used by the todo pane to show details about the selected todo item

Properties Tab shows the attibutes and references of the currently selected model element. Its contents is dependent on the metaclass of the element – a class has different attribues than an actor. These different contents are encapsulated in subclasses of `PropPanel`.

Documentation Tab allows the user to add textual documentation to individual model elements. These text strings stored as tagged values in the model.

Style Tab allows the user to change appearance of the figures in the diagram. This does not change the model in any way, only the visual representation of the model elements.

Source Tab lets the user enter implementation details directly into the model.

Constraint Tab displays the constraints linked to the selected model element.

Tagged Values Tab shows the tagged values of the selected element.

CheckList Tab is used by the cognitive support package.

8.3. The Navigator Pane

The navigator pane is shown to the left of the editor pane. It shows the user's model in a tree-like diagram. It is based on the swing class `JTree`, which requires a class implementing `TreeModel` from which it gets the tree's data.

Since the user's model is a graph and not necessarily a tree, there are many ways (*perspectives*) to represent the model in a tree-like structure. One possibility is to start with the model, branch into the namespaces and packages (and subpackages), branching further into the individual classes and finally to the class attributes and operations (which are leaves in the tree). This perspective is named 'package-centric' and is one instance of `NavPerspective`. Another instance is the 'diagram-centric' perspective, where the root of the tree is the project, which branches into the various diagrams, which branch into the diagram's figures.

The only real difference between the perspectives are the parent-to-child branching rules. In the above example, the package-centric perspective has the rules 'package-to-subpackage', 'package-to-class', 'class-to-attribute', etc. These rules are implemented in `Go`*Parent*`To`*Child* classes realizing the `TreeModelPrereqs` interface. All the rules known to the system are registered in the static vector `NavPerspective.rules`.

One interesting class is the `GoFilteredChildren` class. It does not implement a rule itself, but is used with another rule, filtering out any children not of a specific type. A `GoFilteredChildren` instance configured to filter out anything but stereotypes, together with `GoModelToElements` (which returns all owned elements of a namespace) would make a 'namespace-to-stereotype' rule.

Each `NavPerspective` instance serves as the `TreeModel` for the UI `JTree` component. Through the use of the composite pattern (section 3.5) each `NavPerspective` instance consists of subtrees, which can be either instances implementing the branching rules or other perspectives. This offers a very flexible system for constructing new perspectives. ArgoUML even allows the user to create new perspectives at run time by combining the various rules.

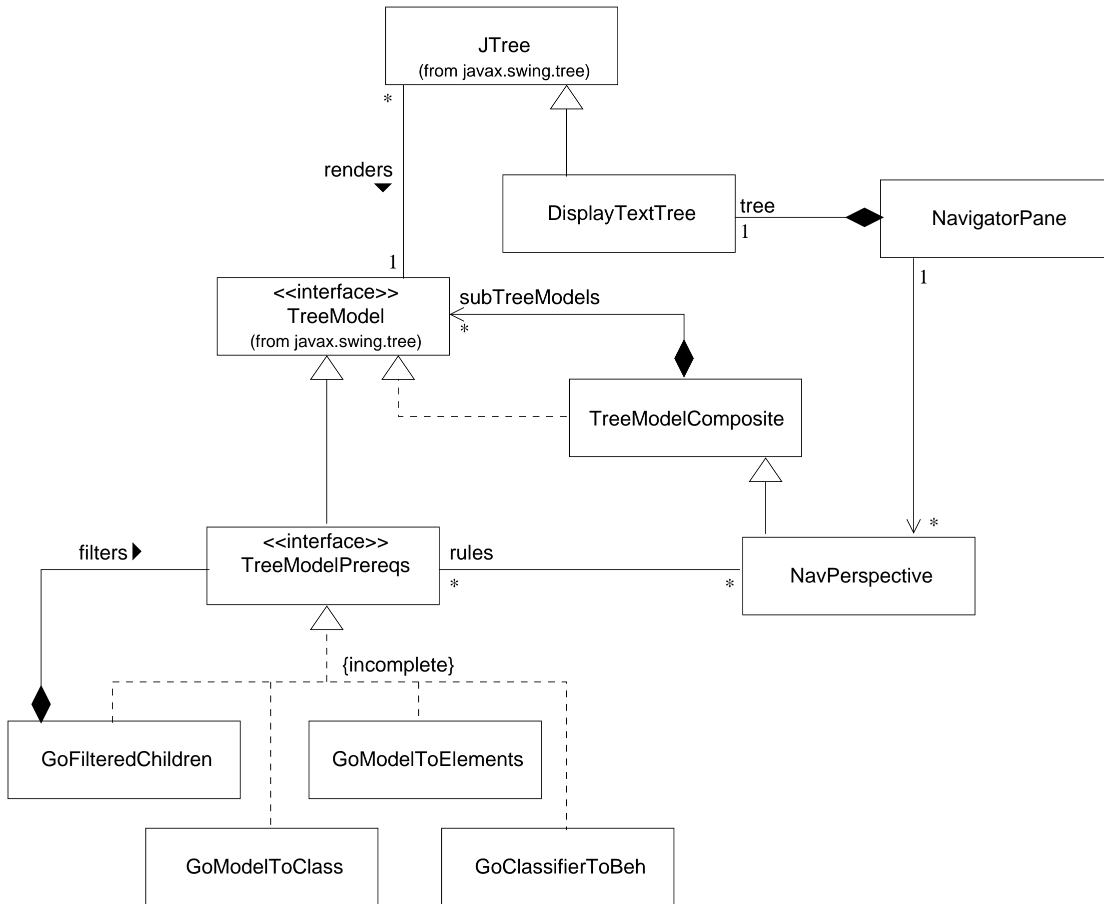


Figure 8.7.: ArgoUML navigator pane classes

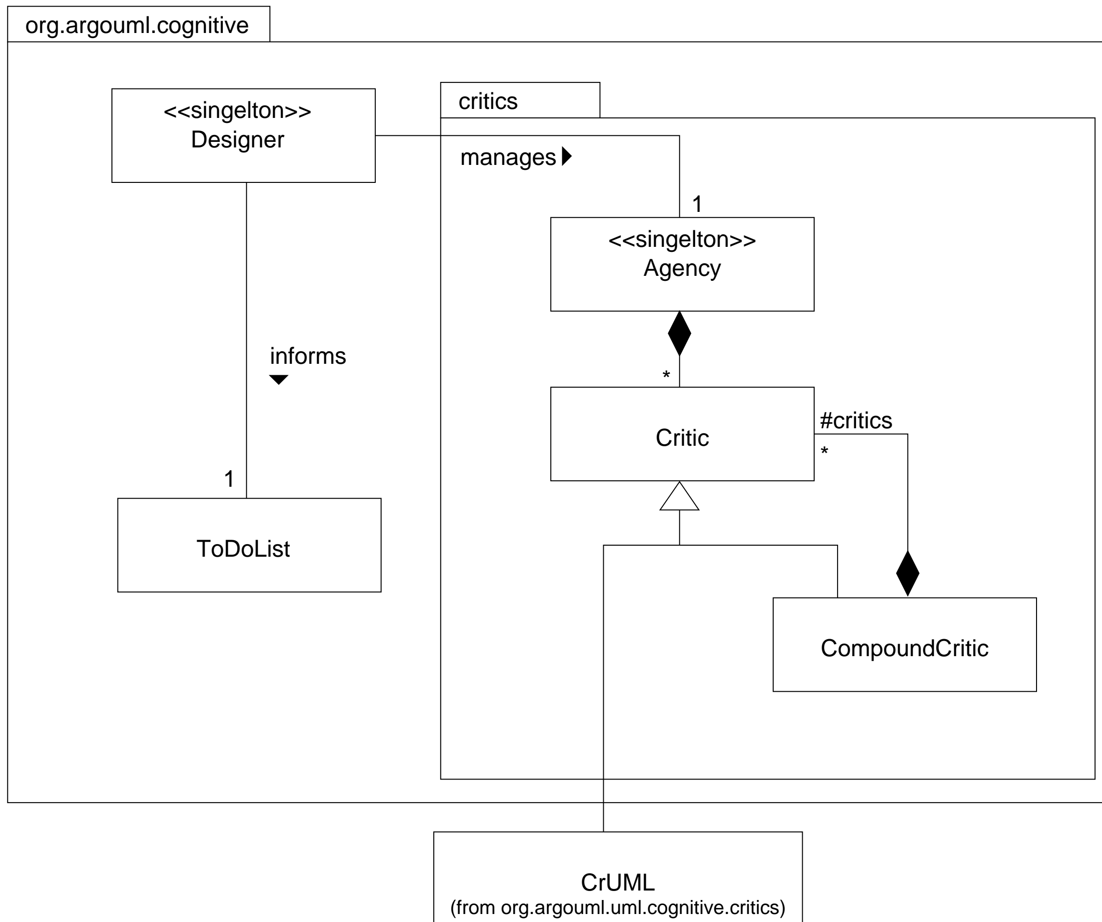


Figure 8.8.: Design critics major classes

8.4. The Design Critics

The design critics are part of ArgoUML's cognitive support package which aids the developer in making and documenting the design decisions.

The `Designer` singelton (section 3.3) constantly observes the user's model. In a background thread, it uses the `Agency` to analyse the model's structure. Specialized `Critic` instances check the user's model for common design errors (for example, an illegal class name or a state without transitions). Through the use of the composite pattern (section 3.5), complex critics can be constructed from simple critics.

If a concrete `Critic` discovers an problem in the model, the `Designer` adds a item to the `ToDoList`, which appears in the todo panel. It is then up to the user to decide whether this issue should be considered or ignored. A critic can also offer a wizard to aid the user in handling the isssue. A wizard could, for example, immediatly show the offending model element in the properties panel.

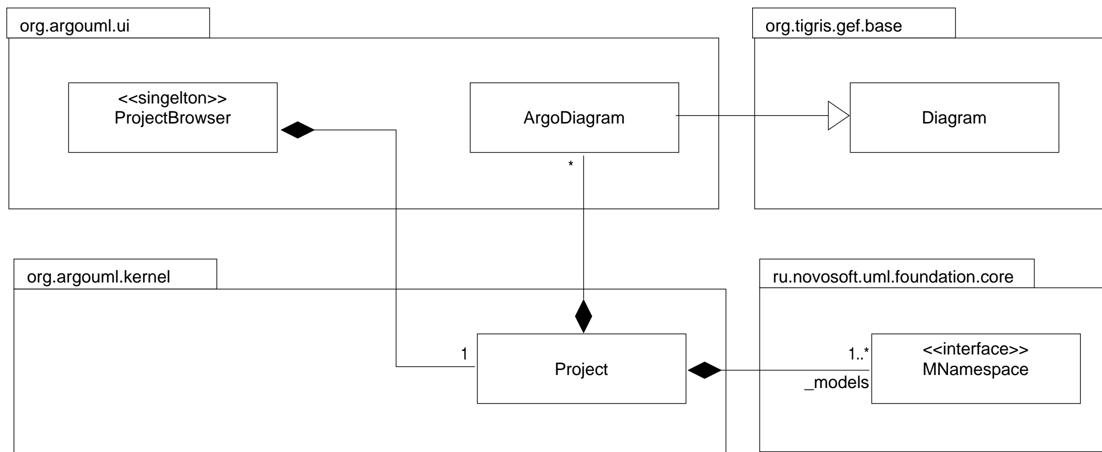


Figure 8.9.: The class `Project` contains the models and the diagrams

8.5. The ArgoUML project

All of the uml diagrams and models in a project are contained in a single `Project` instance[2]. This class is also responsible for loading and saving diagrams and models.

An Argo Project consists of many members, each of which has its own requirements for persistance. The UML model, the diagrams, and information about the project itself must all be saved to a file and, of course, be restored again whenever the user reopens the project. Since both GEF and NSUML can save and restore XML documents, Argo only has to invoke the appropriate methods.

The abstract class `ProjectMember` forms an interface between Argo and the individual member. The concrete classes `ProjectMemberDiagram` and `ProjectMemberModel` override the `save` and `load` methods and call the approriate GEF and NSUML methods.

The `ActionSaveProject.trySave` method (package `org.argouml.ui`) is invoked whenever the user click on the appropriate button or selects the 'Save Project' menu item. This method first saves the project information in an argo-specific XML document. This is done in the same way GEF creates its XML documents: a template file (argo.tee) specifies which member variables of the project are to be saved. Upon completion, every `ProjectMember` gets a chance to save its information:

- The `ProjectMemberModel.save` method creates an `XMIWriter` instance and generates an XMI document as detailed in section 6.10.
- The `ProjectMemberDiagram.save` method creates an `OCLExpander` instance and generates a PGML document as detailed in section 7.3.

[2]ArgoUML has exactly one open project - although the architecture could easily support many simultaneously opened projects

In order to avoid having several, semantically bound documents as separate files, Argo packs them all together into a standard ZIP-file with the extension zargo.

Restoring the state of a project from the XML documents is similar. An `XMIReader` is responsible for recreating the model, and `OCLExpander` instances recreate the project status and the diagrams.

Part III.

The CCL Modelling Tool

9. Extending the ArgoUML Modelling Tool

The main goal of this work is to create a design for the CCL modelling tool by extending ArgoUML to support the concepts and methods introduced in [Bübl2001-CoCons]. The first task was to decide on a general strategy on how to extend ArgoUML while keeping the impact on the ArgoUML sources as small as possible. The resulting plugin mechanism is detailed in section 9.1.

The new concepts require a UML 1.3 metamodel with a subset of the UML 1.4 specification in addition to the CoCon metamodel. Since ArgoUML uses the Novosoft UML 1.3 java implementation library, it is necessary to extend this library to incorporate the new metamodel. The design of the metamodel extension library is detailed in chapter 10, which lays the foundation for the CoCons modeling tool.

A high-level view of the package structure is shown in figure 9.1. Brief descriptions of the package responsibilities follow:

org.cocons.uml contains the java implementation of the CoCons metamodel. The classes implementing model persistance are in the xmi subpackage.

org.cocons.argo contains the extension to the ArgoUML editor.

org.cocons.argo.plugin contains the classes necessary to support the plugin mechanism.

org.cocons.argo.ui contains the user interface components for the new concepts introduced in the CoCons metamodel. This include the property panels which allow the user to edit the model in a dialog-box manner, instead of visually.

org.cocons.argo.cognitve contains the design critics classes which inform the user when the design does not conform to certain semantic rules. One example is a design critic which tests whether the model conforms to all context-based constraints.

org.cocons.argo.diagram contains the classes which implement the new CCL diagram types. This package (and the subpackages) rely heavily on the GEF library.

9.1. Adding a Plugin Mechanism

ArgoUML is not a finished product and its development is advancing at a rapid pace. This poses an interesting challenge for the CCL extensions: since these extensions are not yet ready for production use it is unlikely that they will be incorporated into the main ArgoUML source trunk. Until this time, a parallel branch will exist. Each time

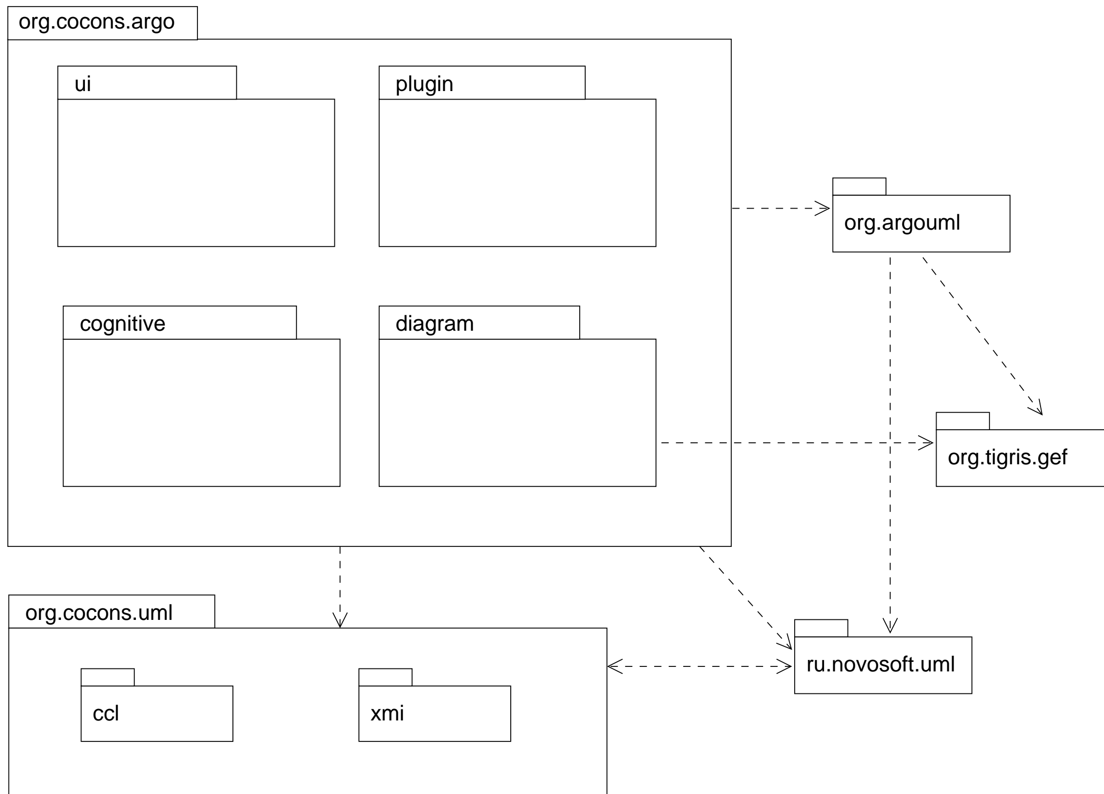


Figure 9.1.: The CoCons modeling tool architecture

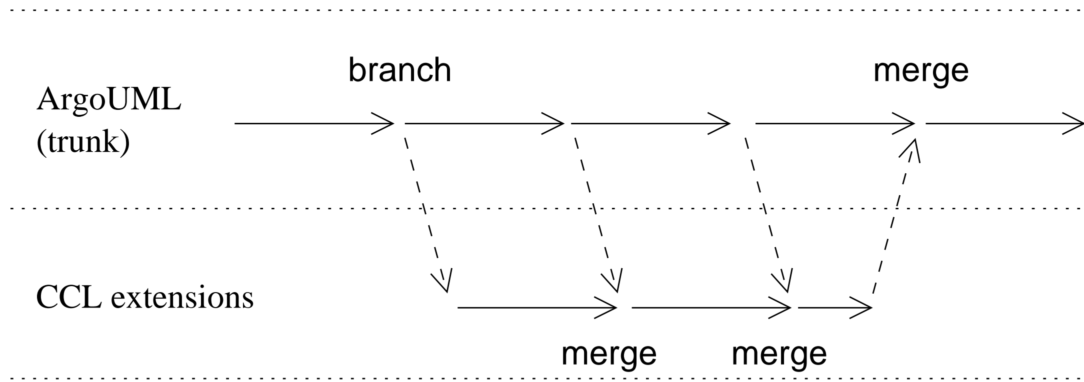


Figure 9.2.: Branching and merging the sources

the ArgoUML sources are changed, these changes will have to be merged with the CCL branch in order to profit from any enhancements. Any ArgoUML files which have been modified in the trunk (by the Argo developers) as well as in the branch (by the CCL developers) may cause merge conflicts, which have to be resolved manually. Depending on the number of files which have CCL specific modifications, the merge procedure could become quite tedious. Once the CCL extensions are ready for production use, they would have to be incorporated back into the main trunk, resulting in yet another merge.

Even then, the system (i.e. ArgoUML and the CCL extension) will continue to evolve due to changes in the environment: newer versions of the unified modeling language will have to be supported, new technologies will appear, etc. It should be possible to modifiy the system in such a way that a consistent system state is achieved while satisfying any new requirements. As described in [Große-Rhode+2000], the decompostion of a system into components can increase the adaptability of the system by limiting dependencies. By separating the system into an ArgoUML-component and a CCL-component, the first step is taken to avoid the two of dependencies that can occur: source dependency and distribution dependency.

9.2. Avoiding source dependency

Source dependency occurs when one package is imported into another package. If a class in the first package is modified, all dependent packages must be rebuilt. Source dependency can be avoided by ensuring that no ArgoUML package imports any CCL packages – even though ArgoUML must be able to (directly or indirectly) access the CCL classes.

Through the use of java interfaces we can prevent ArgoUML from being dependent on the CCL packages (figure 9.3). The class `ProjectBrowser` can invoke methods of `CCLPlugin` through the `ArgoPlugin` interface. Implementation details of the CCL extension remain unknown to ArgoUML and ArgoUML is not dependent on the CCL package. In this way, ArgoUML can be developed regardless of the extension – reducing

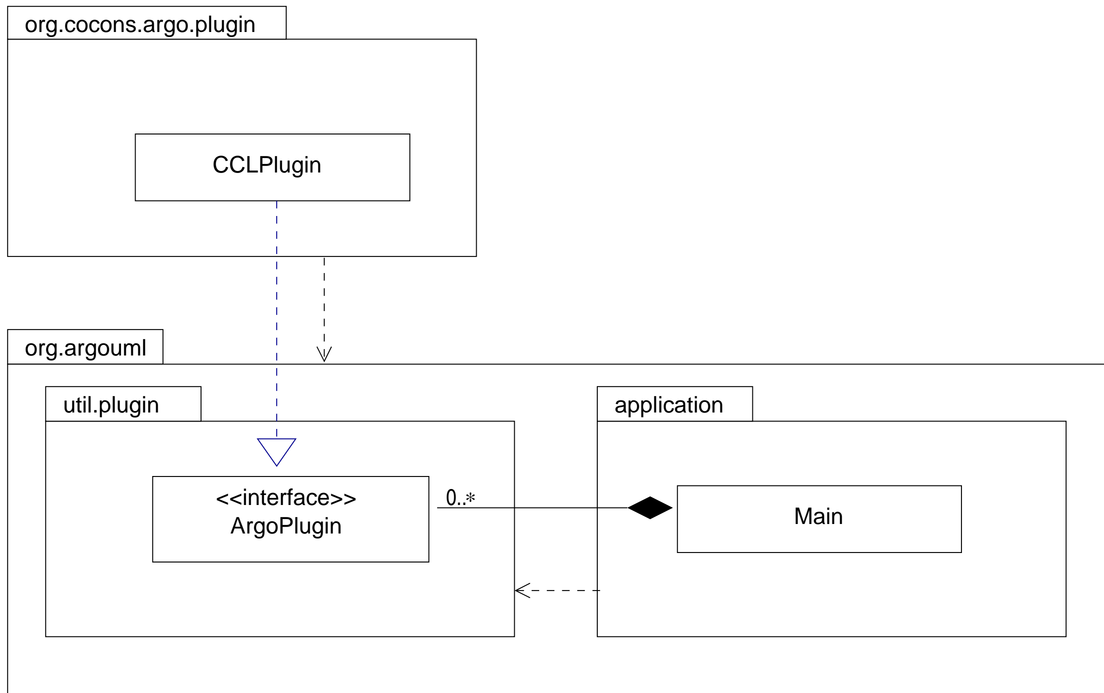


Figure 9.3.: Using interfaces to avoid source dependency

the cost of merges considerably.

9.3. Avoiding distribution dependency

In order to keep ArgoUML distribution independent, the CCL-specific modifications should be implemented as independently distributable plugins. A generic plugin mechanism is an important step in isolating the CCL extensions from the ArgoUML distributables. Such a mechanism would allow the development of ArgoUML extensions requiring no further modification of ArgoUML classes. Ideally, the plugins would be distributed as java archives (.jar) and would simply be copied into a `plugin` directory. They could be developed, deployed, updated or removed independently from ArgoUML without having to reinstall or recompile ArgoUML itself. The user would be able to install plugins without having access to the source of ArgoUML or the extension.

This can be achieved through the use of java's *dynamic extension*[Venners99]. Java's architecture allows applications to dynamically extend themselves by loading and using classes at runtime – even if the classes did not yet exist when the application was compiled. The simplest way to load classes at runtime is with the static method `forName(String className)` from `java.lang.Class`. The `forName` method searches the CLASSPATH for the .class file and loads the class dynamically. This class can then be instantiated with `newInstance()`:

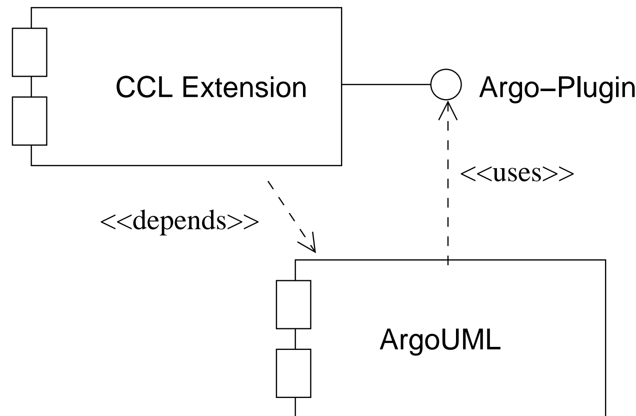


Figure 9.4.: Using interfaces to avoid distribution dependency

```
Class c = Class.forName("DynamicClass");
MyInterface i = (MyInterface) c.newInstance();
i.init();
```

This code dynamically loads the class `DynamicClass`, creates an instance and calls the method `init()`. The class `DynamicClass` may not even have existed when this code was written and compiled – as long as it implements the interface `MyInterface`. Unfortunately, this is not enough to meet the requested requirements: the necessity of having to set the CLASSPATH to include each plugin prohibits easy installation.

A more flexible way of dynamically loading classes is by writing a custom class loader. Java has two kinds of class loaders – the primordial class loader build into the JVM which normally loads all classes and interfaces used by an application, and custom class loaders (also called class loader objects). A custom class loader is not restricted to searching the current CLASSPATH – it can be constructed to load classes from any source: a file, an archive or even a HTTP stream.

Each java archive plugin is represented by an instance of `PluginJarFile`. Each instance contains a custom class loader able to load the plugin's classes on demand from the .jar file. Archive files are great for distribution, but having to construct a .jar file and copy it into the plugin directory after every make cycle would complicate the development process. During development the plugin consists of many .class files. For this reason, an alternative loading mechanism is available. Instances of `PluginClassFile` (also implementing the `PluginSource` interface) each represent a plugin and all of its .class files. The contained `FileClassLoader` is responsible for dynamically loading these .class files. The `PluginManager` acts as a facade hiding all the details of the plugin mechanism. During construction it builds a collection of all plugins – each represented by an object supporting the `PluginSource` interface. Once creation is completed, the application can request a collection of all plugins by calling `getPluginFiles()`.

In order to actually load the plugin classes, the application must iterate through this collection and call `PluginSource.loadClass()`. The returned classes may then be instantiated.

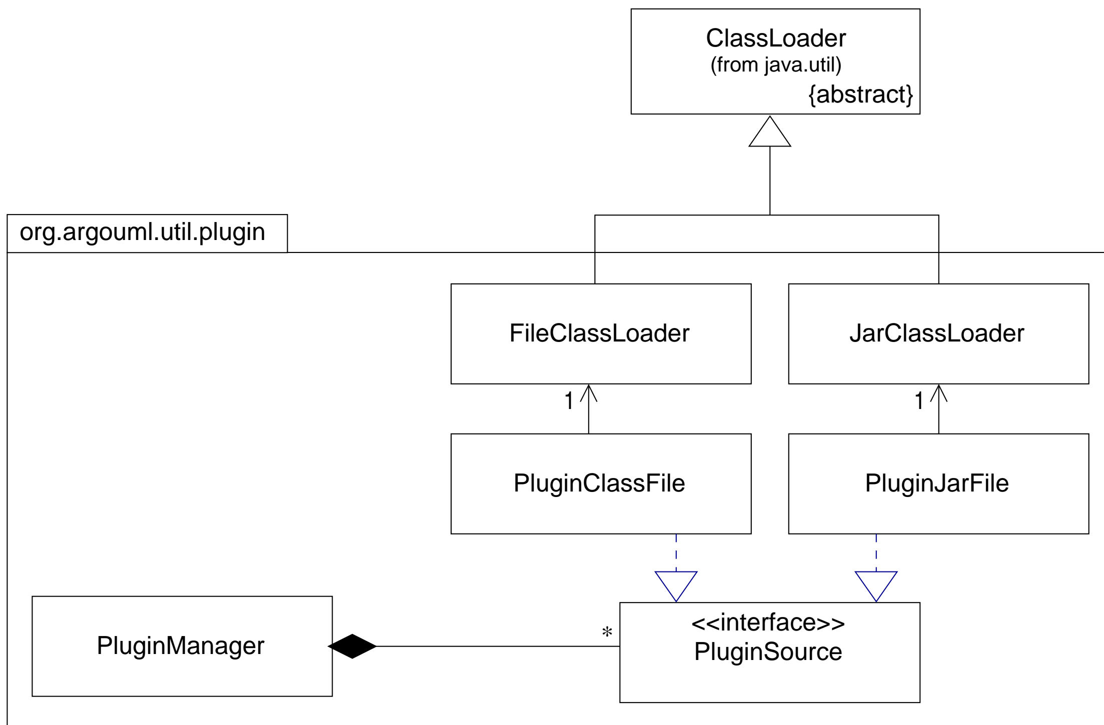


Figure 9.5.: The plugin mechanism custom class loaders

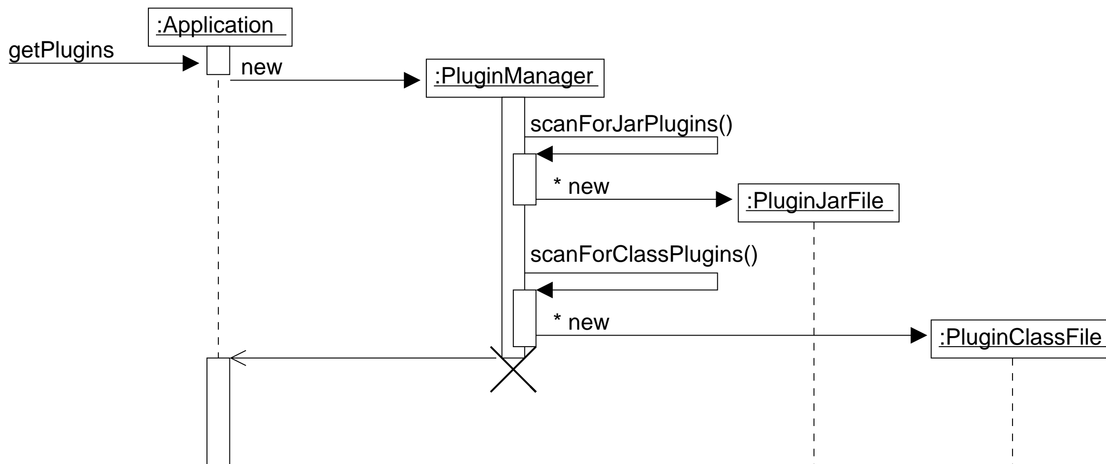


Figure 9.6.: The `PluginManager` finds the plugin sources

```
Vector pluginSources = pluginManager.getPlugins();
Iterator i = classnames.iterator();
while (i.hasNext())
{
  PluginSource pluginSource = (PluginSource) i.next();
  Class c = pluginSource.loadClass();
  ArgoPlugin i = (ArgoPlugin) c.newInstance();
  i.init();
}
```

Exactly what then happens to these instances is up to the application. The plugin mechanism makes no assumptions about the classes to be loaded – although the application itself will require a specific interface. (`ArgoPlugin` in the above example).

9.4. Plugin dependencies

With this architecture, the application is no longer dependent on the plugin, and the plugin's interface is well specified. However, the plugin is heavily dependent on the application. Once the plugin is initialized by the application, it uses whatever classes and data structures of the application it needs. If the application is ever changed in a fundamental way, it will likely 'break' any existing plugins.

The next step would therefore be to minimize the plugin's dependency on the application by defining a set of interfaces plugins may use[1]. As ArgoUML evolves, backward-compatible versions of these interfaces should remain available in order to support older plugins. This would make the application, as well as the plugins, replaceable in regard to the system as a whole.

9.5. Limits

Since version 0.93 ArgoUML is deployable by Java *Web Start*. When using this loading mechanism, the application's distributables are stored on a central http-server. When the application is executed by a client, the distrubutables are downloaded on demand to a client-side cache, where they are then executed. The advantage: no client-side installation of the application is necessary.

The plugin architecture does not account for this variant – although it could be extended to support Web Start. The main issue is where to install the plugins. If the user is to have control over the plugins, then a client-side plugin directory would be necessary. Additionally, ArgoUML would have to know the location of this directory (perhaps specified by an environment variable). Client-side installation (of the plugins) is then required – which defeats the very purpose of Web Start. A second alternative would be server-side plugins. A new class implementing `PluginSource`, analog to the `PluginJarFile` class, but able to load classes from a http server would be required. The application's administrator (not the user) would decide which plugins are to be made available to the users.

[1]as this work was nearing completion, the Argo community started developing a set of plugin-APIs.

10. The CoCons Physical and Object Models

The CoCons physical model relates to the CoCons metamodel [Bübl2001-CoCons, section 3] in the same way the UML physical model relates to the UML metamodel. There are fewer changes necessary than with the UML models, since the CoCons metamodel contains no association classes:

- unnamed association ends are named
- all assocations are bidirectionally navigatable
- metaattributes changed to start with a lowercase letter
- rolenames changed to start with a lowercase letter
- packages names changed to contain only lowercase letters

The CoCons CCL object model contains the changes necessary to implement the CoCons physical model in Java. The following requirements should be met:

1. The CoCons CCL object model should fully implement the CoCons physical model
2. The CoCons CCL object model should follow the syntatic and semantic conventions of the NSUML object model
3. All features offered by the NSUML classes should also be offered by the CCL classes (undo/redo, persistance, etc.)

10.1. Extending the NSML Library

The ConCons metamodel extends the UML 1.4 metamodel. Unfortunatly, at the time of this writing, NSUML only implements the UML 1.3 object model. Future versions of the Novosoft libraries will support automatic generation of of an object model from an abstract metamodel specification (NSGEN and NSMDF libraries). This will open new possibilites for modelling tools. It will then be possible to simply modify the metamodel specification and re-generate the java library.

Until ArgoUML supports these NSGEN and NSMDF releases, the NSUML library must be extended to include the modifications introduced in the UML 1.4 specification.

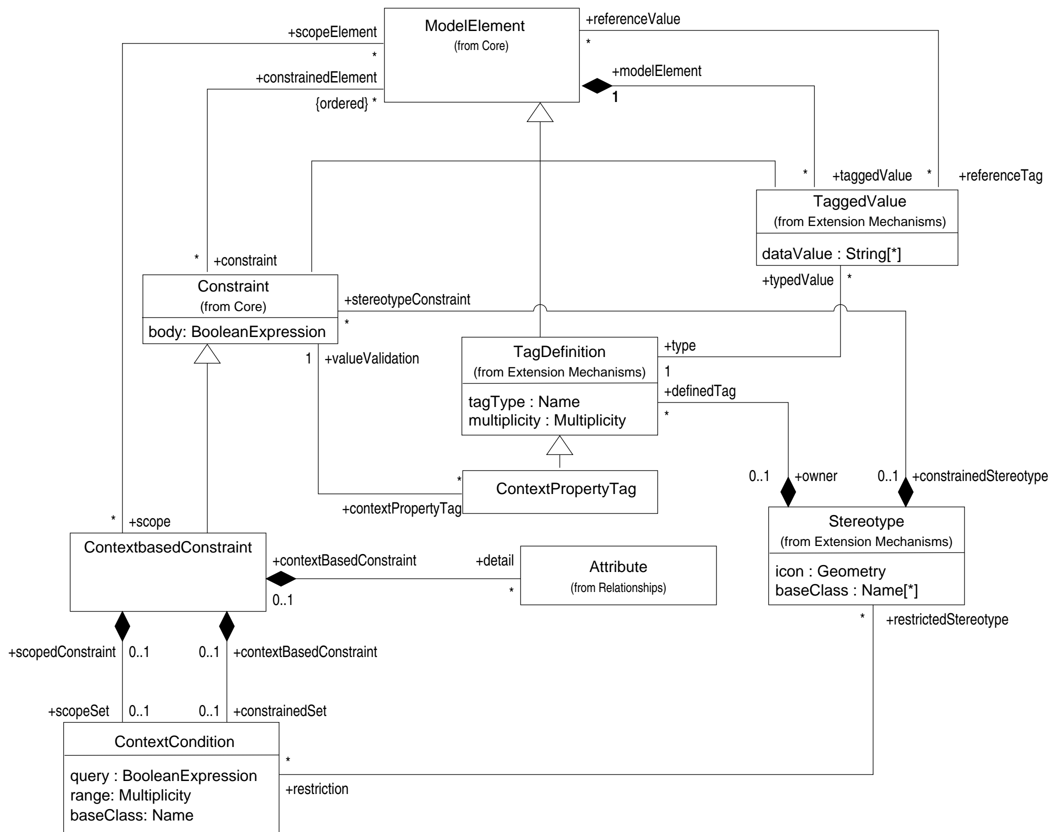


Figure 10.1.: CoCons physical model

Since CoCons only relies on a few UML metaclasses, supporting the following subset of the UML 1.4 specification is sufficient:

1. multiplicity of `ModelElement` end of association `modelElement (ModelElement) / taggedValue (TaggedValue)` changed from 0..1 to 1
2. association `referenceValue (ModelElement) / referenceTag (TaggedValue)` added
3. `TaggedValue` attribute `value` replaced with `dataValue` – datatype changed from `String` to `String[*]`
4. `TaggedValue` is now derived from `ModelElement` – attribute `name` now inherited from `ModelElement`
5. datatype of `Stereotype` attribute `baseClass` changed from `Name` to `Name[*]`
6. multiplicity of `ModelElement` end of association `constrainedElement (ModelElement) / constaint (Constraint)` changed from 1..* to *
7. unnamed association role `(Stereotype) / stereotypeConstraint (Constraint)` now named `constrainedStereotype (Stereotype) / stereotypeConstraint (Constraint)` [1]
8. association `requiredTag (TaggedValue) / stereotype (Stereotype)` removed
9. metaclass `TagDefinition` added as subclass of `ModelElement`
10. association `typedValue (TaggedValue) / type (TagDefinition)` added
11. association `owner (Stereotype) / definedTag (TagDefinition)` added

This can be achieved in two ways:

1. By subclassing the necessary NSUML classes and interfaces. A new UML 1.4 class (e.g. `CStereotypeImpl`) would be derived from the UML 1.3 class (`MStereotypeImpl`) and override the appropriate methods. Modifying the `MFactory` class factory (section 3.4) would then ensure that only the new classes are instantiated. This would avoid having to modify any of the NSUML source files, avoiding the brach/merge problems described in section 9.1. Problems could occur if ArgoUML instantiates the UML 1.3 classes directly without using factory methods.
2. By directly modifying the NSUML sources. There would be no obsolete classes, avoiding the above problem – at the cost of the branch/merge problem.

[1] in the physical model the role was named `constrainedElement2` in UML1.3

An analysis of the sources reveals that ArgoUML does not make extensive use of the class factory – the UML 1.3 classes are instantiated directly. The modifications necessary to make all ArgoUML sources use the class factory would far outweigh simply modifying the NSUML sources. Additionaly, the current version of NSUML (version 0.4.19) seems to be quite stable – until the next release of the NSGEN/NSMDF packages. Both of these reasons favor the second alternative – modification of the NSUML sources:

Modification 1 Since the java object model does not differentiate between the multiplicities 0..1 and 1 (section 6.6), no modification is necessary

Modification 2 Appropriate bag role (section 6.6) members must be added to `MModelElement` and `MTaggedValue`

Modification 3 This array-of-strings datatype did not exists in UML 1.3. In order to support this in a way consistent to NSUML, it should be implemented as a `Collection` with the appropriate `getDataValues`, `setDataValues`, `addDataValue` and `removeDataValue` methods.

Modification 4 The `MTaggedValue` interface and `MTaggedValueImpl` class must be changed to be derived from `MModelElement` and `MModelElementImpl`. The obsolete inspector and mutator methods for the UML 1.3 fields `tag` and `value` should be implemented to throw a `NoSuchMethodError`.

Modification 5 Analog to modification (3): the appropriate `getBaseClasses`, `setBaseClasses`, `addBaseClass` and `removeBaseClass` methods must be added.

Modification 6 Analog to modification (1): no changes to the list role are necessary.

Modification 7 The `*constrainedElement2`-members should be renamed to `*constrainedStereotype`.

Modification 8 The appropriate bag role methods should be added to `MConstraint`.

Modification 9 The bag role methods for this association should be removed from both classes.

Modification 10 The interface `MTagDefintion` and implementation class `MTagDefinitionImpl` should be added (derived from `MModelElement` and `MModelElementImpl`, respectivly)

Modification 11 The appropriate bag role members should be added to `MTagDefinition` and reference role members to `MTaggedValue`.

Modification 12 The appropriate bag role members should be added to `MStereotype` and reference role members to `MTagDefinition`.

In addition to the above modifications, the XMI support classes must also be modified to support the new metamodel.

With these modifications, the UML library is effectivly extended to support UML 1.4 (at least the subset required by the CoCons metamodel), which can then serve as a foundation for the ConCons CCL object model. If and when NSUML supports UML 1.4, these modifications will not longer be necessary and it should take little effort to modify the CoCons CCL library to use the future NSUML 1.4 classes directly.

10.2. The CoCons CCL Java Library

The CoCons CCL Java library is a direct java implementation of the CoCons CCL object model. In order to be consistent with the NSUML object model, the same conventions are used:

- Each metaclass is implemented as a java class. Analog to the NSUML naming convention, the class shoud be named after the metaclass, prefixed with the letter "M" and suffixed by "Impl" (e.g. `MContextConditionImpl`).
- Each metaclass has an interface specifying the inspector and mutator methods. The interface should be named after the metaclass, prefixed with the letter "M" (e.g. `MContextCondition`) – the java objects should always be referenced by their interfaces – never directly.
- Attributes should never be declared public – the appropriate `get`*`Attribute`* / `set`*`Attribute`* methods must be defined for each attribute.
- Associations must always be implemented in the classes at both ends. Depending on the multiplicity the role, different inspector and mutator methods are specified:
  - reference roles (multiplicity 0..1 or 1) define `set`*`Role (OppositeRole)`* and `get`*`Role()`* methods
  - bag roles (unordered multiplicity other than 0..1 or 1) define `set`*`Roles`*` (Collection)`, `get`*`Roles`*`()`, `add`*`Role`*` (`*`OppositeRole`*`)` and `remove`*`Role`*` (Opposite`*`Role`*`)` methods
  - list roles (ordered multiplicity other than 0..1 or 1) define `add`*`Role`*` (int, `*`OppositeRole`*`)`, `remove`*`Role`*` (int)`, `set`*`Role`*` (int, `*`OppositeRole`*`)` and `get`*`Role`*` (int)` methods to allow accessing the roles at specific positions in the list, in addition to the methods defined by bag roles.
- Regardless of the multiplicity, `internalRef`*`Role`* and `internalUnref`*`Role`* methods must be implemented for each role. These methods must be called by the association mutator methods at the opposite end to ensure the state of an association remains consistent at both ends.

- Every class must override the appropriate `reflective*` methods to allow setting and getting the attributes and roles.
- To support event notification, all mutator methods must begin with an call to `operationStarted()` and end with `operationFinished()`. Whenever the state is changed, the appropriate `fire*` method must be called, generating an event.
- To support undo/redo functionality each state change must be logged with the appropriate compensation operation. The necessary compensation methods should be stored as private static member variables:
  - attributes specifiy `_`*attribute*`_setMethod` method objects.
  - reference roles specify `_`*role*`_setMethod` method objects.
  - bag roles specify `_`*role*`_setMethod`, `_`*role*`_addMethod` and `_`*role*`_removeMethod` method objects.
  - list roles specify `_`*role*`_listSetMethod`, in addition to the method objects specified by bag roles.
- Every class must be derived from `MBaseImpl` or one of its subclasses (e.g. `MModelElementImpl`), every interface must be derived from `MBase` or one of its subclasses.
- Every instance must have exactly one container[2]. Subclasses of `MModelElement` are already contained by their namespace. If an class can be contained through another association (e.g. `MConstraint` can be contained either by its namespace or by its stereotype), then the set* and internalRefBy* methods must also call `MBase.setElementContainer`.

10.3. CoCons Model Persistance

According to the XMI specification, it should be possible to create an XML document specifying the metamodel according to the MOF-DTD. With the appropriate tools, a CCL-DTD could then be automatically generated from this MOF document. Sadly, there is a lack of freely available tools which support this. For this reason, the UML-DTD must be extended manually to include the new metaclasses and associations of the CCL object model. This extended DTD ('CCL10.DTD'), located in the `org.cocons.uml.ccl` package, can then be used to validate CCL-XMI documents containing CCL user models.

10.3.1. Extending the UML-Document Type Definition

In the UML-DTD, each metaclass is represented by its own XML-element. This element contains subelements for each of its attributes and roles. Additionally, every XML-element representing a metamodel class must define the XML-attributes %XMI.element.att

[2]the top-level namespace (or model) is the exception to this rule

and %XMI.link.att. These XML-attributes are used for resolving references when restoring the model from an XMI document.

```
<!ELEMENT CoCons.Uml.Ccl.ContextCondition (
CoCons.Uml.Ccl.ContextCondition.query?,
CoCons.Uml.Ccl.ContextCondition.range?,
XMI.extension*,
CoCons.Uml.Ccl.ContextCondition.contextPropertyValue*,
CoCons.Uml.Ccl.ContextCondition.restrictedStereotype*,
CoCons.Uml.Ccl.ContextCondition.scopedConstraint?,
CoCons.Uml.Ccl.ContextCondition.contextBasedConstraint?
)?>
<!ATTLIST CoCons.Uml.Ccl.ContextbasedConstraint
%XMI.element.att;
%XMI.link.att;
>
```

The element 'XMI.extension' is a predefined XML-element which is used by some applications to store data in the XMI document which is out of the scope of the XMI specification. It is should be placed between the attributes and the roles.

The subelements used in the above definition must also be defined:

```
<!ELEMENT CoCons.Uml.Ccl.ContextCondition.query
(Foundation.Data_Types.BooleanExpression)
>
<!ELEMENT CoCons.Uml.Ccl.ContextCondition.range
(Foundation.Data_Types.Multiplicity)
>
<!ELEMENT CoCons.Uml.Ccl.ContextCondition.contextPropertyValue
(Foundation.Extension_Mechanisms.TaggedValue)
>
...
```

Note the definition of `contextPropertyValue`: it contains the element `TaggedValue` – this is how references are encoded in XMI. If the referered element has subclasses, they too must be listed in the reference. For example, the role `feature` of a `Classifier` refers to the metaclass `Feature`. It has the subclasses `StructuralFeature` and `BehavioralFeature`, which have the subclasses `Attribute`, `Operation` and `Method` and `Reception`. This means that a `feature` could refer to any of these subclasses. Therefore the XML-element representing the role is defined as:

```
<!ELEMENT Foundation.Core.Classifier.feature
(Foundation.Core.Feature |
Foundation.Core.BehavioralFeature |
Behavioral_Elements.Common_Behavior.Reception |
Foundation.Core.Operation |
Foundation.Core.Method |
Foundation.Core.StructuralFeature |
Foundation.Core.Attribute
)* >
```

If the new metaclass is subclass of an existing class, then all the inherited attributes and roles must also be included in the XMI-element definition:

```
<!ELEMENT CoCons.Uml.Ccl.ContextbasedConstraint
(Foundation.Core.ModelElement.name?,
Foundation.Core.ModelElement.visibility?,
Foundation.Core.ModelElement.isSpecification?,
Foundation.Core.Constraint.body?,
CoCons.Uml.Ccl.ContextbasedConstraint.priority?,
XMI.extension*,
Foundation.Core.ModelElement.namespace?,
Foundation.Core.ModelElement.clientDependency*,
Foundation.Core.ModelElement.constraint*,
Foundation.Core.ModelElement.supplierDependency*,
Foundation.Core.ModelElement.presentation*,
Foundation.Core.ModelElement.targetFlow*,
Foundation.Core.ModelElement.sourceFlow*,
Foundation.Core.ModelElement.templateParameter3*,
Foundation.Core.ModelElement.binding*,
Foundation.Core.ModelElement.comment*,
Foundation.Core.ModelElement.elementResidence*,
Foundation.Core.ModelElement.templateParameter2*,
Foundation.Core.ModelElement.stereotype*,
Foundation.Core.ModelElement.behavior*,
Foundation.Core.ModelElement.classifierRole*,
Foundation.Core.ModelElement.collaboration*,
Foundation.Core.ModelElement.partition*,
Foundation.Core.ModelElement.elementImport*,
Foundation.Core.Constraint.constrainedElement*,
Foundation.Core.Constraint.constrainedStereotype?,
Foundation.Core.Constraint.contextPropertyTag*,
Foundation.Core.ModelElement.templateParameter*,
Foundation.Core.ModelElement.taggedValue*,
Foundation.Core.ModelElement.referenceTag*
)? >
```

Additionally, anywhere one of the superclasses is referenced, the new class must be added to the list:

```
<!ELEMENT Foundation.Core.ModelElement.constraint
(Foundation.Core.Constraint |
 CoCons.Uml.Ccl.ContextbasedConstraint
)* >
```

To sum all this up: when adding a new metaclass to the metamodel, the following changes to the XMI-DTD must be made:

1. XMI-elements representing the direct (i.e. not inherited) attributes and roles of the new metaclass must be defined
2. an XMI-element representing the metaclass must be defined. This XMI-element must contain sub-elements representing all (direct and inherited) attribues and roles of the metaclass

3. any XMI-elements representing roles which refer to any superclass of the new metaclass must be extended to include the new metaclass

When a new association or attribute is added to an existing metaclass

1. XMI-elements representing the new attributes or roles must be defined
2. the contents of the XMI-elements representing the metaclass to which the attribute or role belongs *or any of its subclasses* must be extended to contain the new XMI-elements

10.3.2. Writing XMI documents

The class `XMIWriter` (package `ru.novosoft.uml.xmi`) is responsible for encoding a given model as a XMI document. It has been refined to allow extensions by subclassing. The subclass `CCLXMIWriter` (in `org.cocons.uml.xmi`) contains the methods which encode the CoCons metaclasses.

The classes `CCLXMIReader` and `CCLXMIWriter` offer ConCons model persistance by reading and writing CCL10.DTD valid XMI documents. They extend the (modified) NSUML classes `XMIWriter` and `XMIReader` by adding support for the metaclasses and associations introduced in the CoCons metamodel.

10.4. Object Model Test Cases

The UML1.4 metamodel and the CoCons metamodel are both built on the same metaelements: they contain metaclasses, the metaclasses are connected by metaassociations and have metaattributes – in other words, they have the same meta-metamodel. For this reason, the number of *kinds* of test cases is quite limited. A simple test suite could consist of tests to:

- test the inspector and mutator methods for metaattributes
- test the inspector and mutator methods for metaassociations
- test the undo and redo support
- test loading and saving XMI files

The class `MMTestCase` (derived from `TestCase`) contains several auxillary methods which can be used when creating test cases.

10.4.1. Testing metaattributes

The method `testAttrib` perfoms the following operations on a single attribute:

1. the attriute is set to an arbitrary value

2. the attribute is set to null
3. the ‘set to null’ operation is undone
4. the ’set to null’ operation is redone

Since some attributes have data types based on collections (e.g. `TaggedValue.dataValue`), there is also the test method `testAttrib_Bag`:

1. add two different values to the attribute
2. remove one value from the attribute
3. undo the remove operation
4. redo the remove operation
5. set the attribute to null (removing all elements from the collection)

After each operation the state of the attribute is compared with the expected value.

10.4.2. Testing metaassociations

The metaassociation test methods are similar to the metaattribute methods but also test the symmetry of the association. As with the metaassocaition methods, different methods are available, depending on the multiplicity of the roles. Both methods perform the following operations:

1. an association is set/added at the far end
2. the association is removed from the near end
3. the remove operation is undone
4. the remove operation is redone
5. the association is added to the near end
6. all associations are cleared at the far end

When writing a test case for associations with the same multiplicity at both ends, it can’t hurt to call the test method twice, reversing the far and near ends between the calls.

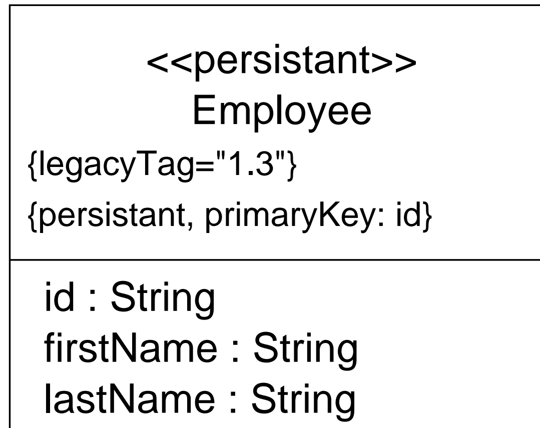


Figure 10.2.: The test user model (M1)

10.4.3. Testing persistance

In order to test the XMI methods, the following step are performed:

1. create a test model
2. save the model to an XMI document
3. validate the XMI document to the DTD
4. restore the model from the XMI document
5. compare the restored model to the original test model

The test model must be carefully considered: it should contain as many metamodel features as possible, yet the test code must not become overly complex in order to avoid having to debug the test case itself.

For this reason, the user model in figure 10.2 was choosen. Not shown in the diagram is the specification of the stereotype `<<persistant>>`. This stereotype defines two tag definitions: `isPersistant`, which specifies tags containing a single boolean value, and `primaryKey`, which specifies tags referring to at least one attribute in the user model. The class `Employee` includes both of these tags: `isPersistant` has the value 'true' and `primaryKey` refers to the attribute `id`. Additionaly, a UML 1.3-compatible tag, `legacyTag` is also included.

How this user model is represented in the UML 1.4 metamodel is shown in figure 10.3. In the class UMLTest this model is built, saved as an XMI document and then restored. The objects and links of the restored model are then examined for correctness.

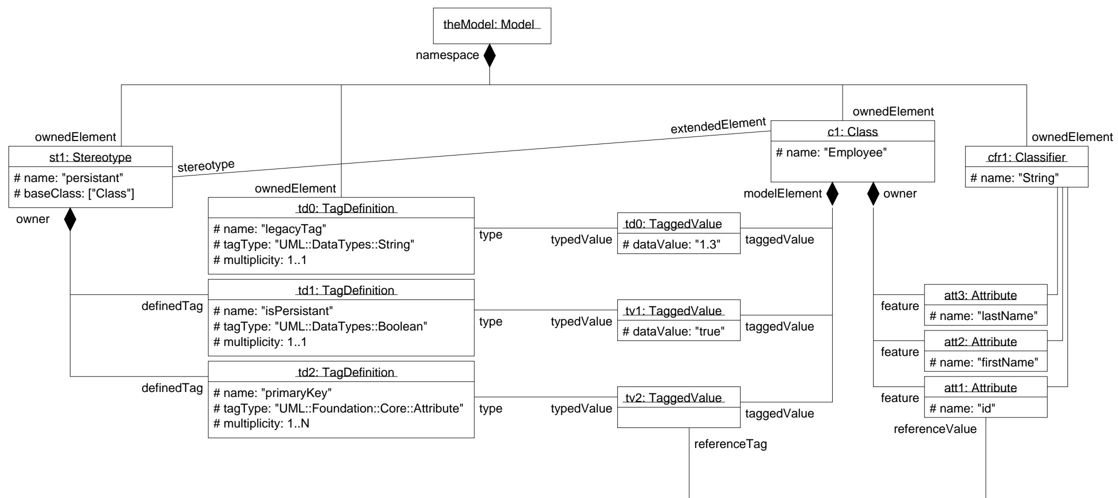


Figure 10.3.: The test model as a M2 object diagram

11. The CCL Editor Diagrams

When modelling, the user will spend most of his time working with the various diagrams representing his model. Therefore it is important to keep the user interface as intuitive as possible so the user can concentrate on his model, not the modelling tool.

The various diagrams in ArgoUML are implemented using the GEF library. In this framework, most of the diagram functionality is not in the diagram itself, but in the figures which make up the diagram. The diagrams are easily extensible and not only the appearance but also the behavior is encapsulated in the individual figures.

Classes specific to a certain diagram are structured in diagram packages. Classes which are common to all diagrams are placed in a separated `diagram.ui` package. Each diagram requires the following classes:

DiagramGraphModel derived from `MutableGraphSupport` and implementing `MutableGraphModel`. This class forms the adapter between the GEF framework and the metamodel. If a change in the diagram requires a change in the underlying model then this class must perform the necessary operations on the model. The base class `MutableGraphSupport` reacts to changes in the model and notifys the diagram that it may need to be updated.

DiagramRenderer implementing `GraphNodeRenderer` and `GraphEdgeRenderer`. This class is responsible for mapping the model elements to diagram figures.

DiagramDiagram is the main class representing the diagram. It must create the proper renderer and graph model classes. It is responsible for adding the commands the user may activate to edit the diagram. These classes should be derived from an abstract `CoConDiagram` class.

The `CCLDiagram` class (packaged in `diagram.ui`) extends the `UMLDiagram` class with functionality common to the new CCL diagrams. These diagrams will need, for example, an 'add context-property' action which is included in the diagrams toolbar. Another common action could be a 'filter model elements' action, which modifies the diagram to show only the model elements which have certain context properties.

These common actions should be declared as static member variables of the `CCLDiagram` class. The subclasses can then add these instances to their toolbars as required.

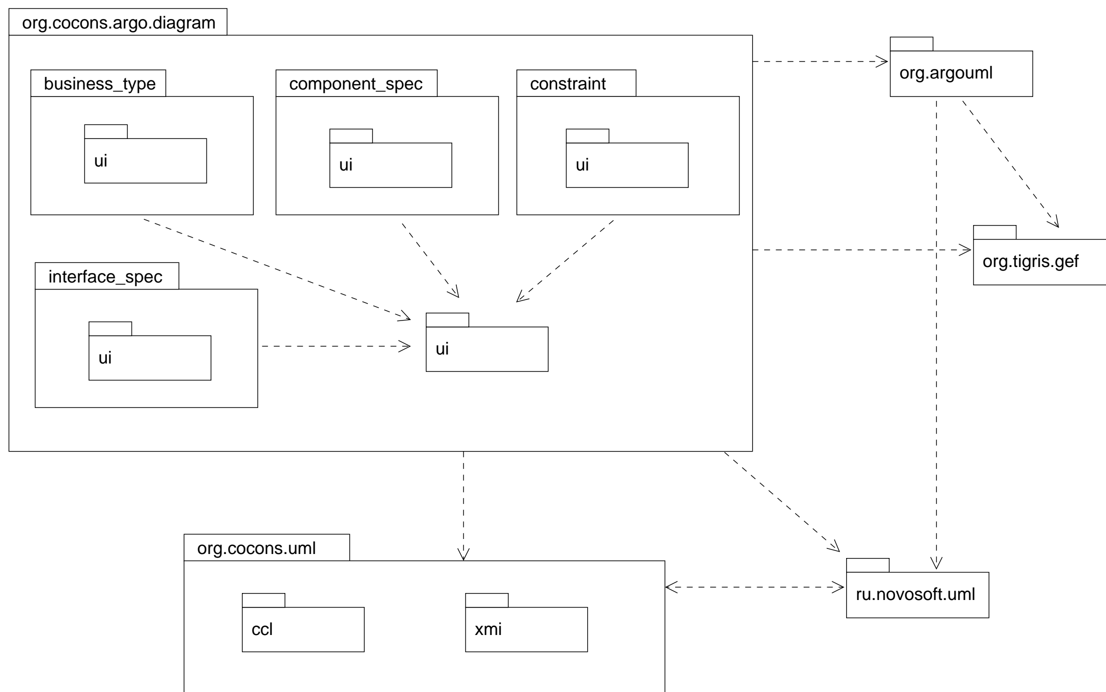


Figure 11.1.: The CoCons diagram package structure

11.1. The Figure Classes

The GEF library uses figures to graphically represent the nodes and edges in the graph model as detailed in section 7.1. As a rule of thumb, `FigNode` subclasses should be used if the figure can be used independently, and `FigEdge` subclasses should be used if the figure can only be used to join two nodes.

Although GEF supports the concept of ports, the AgroUML diagrams make no use of this feature and simply define each node as a single encompassing port. Determining whether two nodes may be connected by an edge is then left up to the class implementing `MutableGraphModel`.

ArgoUML supplies some auxillary classes which should be used when creating new figure classes:

`FigNodeModelElement` should be subclassed if the metaclass to be represented is derived from `MModelElementImpl` and the figure can be independently placed on a diagram. (e.g. `FigClass`, `FigActor`, `FigMessage`)

`FigEdgeModelElement` should be subclassed if the metaclass to be represented is derived from `MModelElementImpl` and the figure should connect nodes to each other (e.g. `FigGeneralization`, `FigTransition`)

These classes already declare subfigures for the `ModelElement.name` and `ModelElement.stereotype` attributes, but it is up to the subclass to actually add them to the

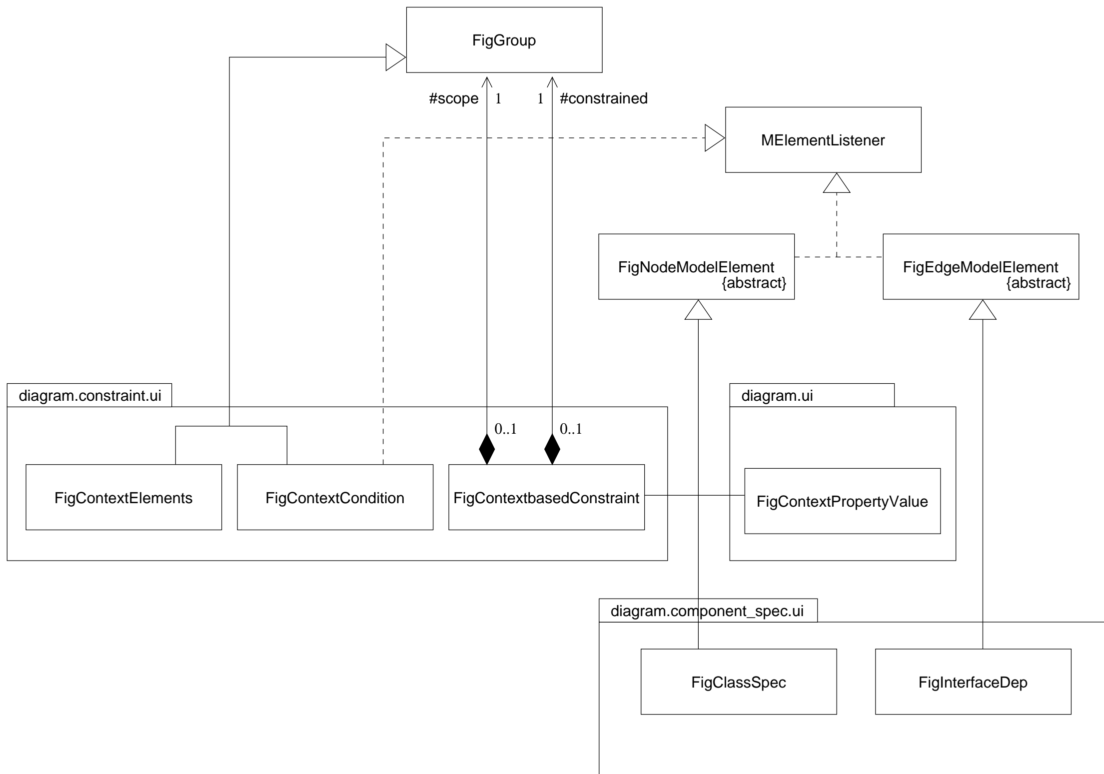


Figure 11.2.: CoCons figure classes

figure.

In some cases, figure classes are used which have no corresponding node or edge in the graph model. One example of this is the `FigEdgeNote` class which connects a `FigNote` (representing a `MComment`) to its model element figure.

Creational methods

The following methods are called when figures are created:

default constructor in the default constructor any subfigures should be created.

clone creates an exact copy of the figure, including copies of all subfigures.

placeString this text will be displayed while placing a new figure on a diagram. Usually 'new *metaclass*'.

Style methods

The following methods change the appearance of the figure without modifying the model:

getFillColor return the current color used to fill closed shapes.

setFillColor sets the color used to fill new shapes. Should be passed on to any subfigures.

getFilled returns true if the figure fills closed shapes.

setFilled sets whether closed shapes should be filled with the fill color. Should be passed on to any subfigures so the entire figure is drawn in a uniform style.

getLineColor returns the color to draw the lines in the figure.

setLineColor set the color which is used to draw lines in the figure. Should be passed on to any subfigures so the entire figure is drawn in a uniform style.

getLineWidth returns the width (strength) the of lines in the figure.

setLineWidth set the line width which is used to draw lines in the figure. Should be passed on to any subfigures so the entire figure is drawn in a uniform style.

getMinimumSize returns the minimum size the figure has to be. If the figure is resizeable, the user cannot resize it smaller than this value.

setBounds is called whenever the bounding box is changed. The figure should be rescaled and/or rearranged to fill the new bounding box.

isResizeable returns whether the user can resize the figure by dragging the corners of the bounding box.

Model Manipulation methods

The following methods access or manipulate the model:

setOwner sets the model element which should be represented by this figure

makeSelection should create a new `Selection` object

modelChanged is called whenever the model element (the owner) is changed. By overriding this method a figure can redraw itself to show the new state of the model element

updateStereotypeText is called whenever the stereotype of the model element is changed. This should be overridden if the stereotype is to be shown in the figure

The behavoir of selected figures is encapsulated in `Selection` objects. They are created by the `SelectionManager` whenever a figure is selected (section 7.2). Every `Fig`-subclass should be able to create a selection object, implementing the operations which can be used on this figure (or the model element the figure represents). In addition to the selection classes offered by GEF, ArgoUML also defines the following selection classes:

SelectionEdgeClarifiers extends `SelectionReshape` and assumes it was created by a `FigEdgeModelElement`, giving this figure a chance to draw handles on the selected bounding box. The user is still able to change the path of the edge by adding and moving embedded points.

SelectionNodeClarfiers extends `SelectionResize` and assumes it was created by a `FigNodeModelElement`, giving this figure a chance to draw handles on the selected bounding box. The user is still able to move or resize the figure by moving the corner handles of the bounding box.

SelectionMoveClarifiers extends `SelectionMove` and assumes it was created by a `FigNodeModelElement`, giving this figure a chance to draw handles on the selected bounding box. The user is still able to move the figure by dragging the bounding box.

SelectionWButtons is an abstract class extending `SelectionNodeClarifiers`. Subclasses of this class may implement figure-dependent 'buttons' which appear when the figure is selected.

11.1.1. The Context-based Constraint Figure

The `FigContextBasedConstraint` class is used to represent an instances of the `MContextbasedConstraintImpl` class. Since what is represents is both a node (in GEF terms) and a model element (in UML terms) it is derived from `FigNodeModelElement`.

It contains several primatve sub-figs (the triangle, the arrow, etc.) as well as `FigGroups` which draw the figure depicting the constrained elements and the scoped elements. Some of these sub-figs do not have a constant size – the text fields expand to the size of the text and the size of the scope and constrained sub-figs is determinied by the type of elements they reference. For this reason, the coordinates must be recalculated whenever the model elements changes.

Wc is the width of the figure depicting the constrained elements

Wt is the width of the widest of the text fields above and below the arrow

Ws is the width of the figure depicting the scoped elements

Depending of whether the scoped or constrained elements are directly or indirectly defined, the subfigures are either `FigContextElements` or `FigContextCondition` instances:

Associated Elements		Use Subfigure	
scoped	constrained	scoped	constrained
direct	direct	FigContextElements	FigContextElements
direct	indirect	FigContextElements	FigContextCondition
indirect	direct	FigContextCondition	FigContextElements
indirect	indirect	FigContextCondition	FigContextCondition

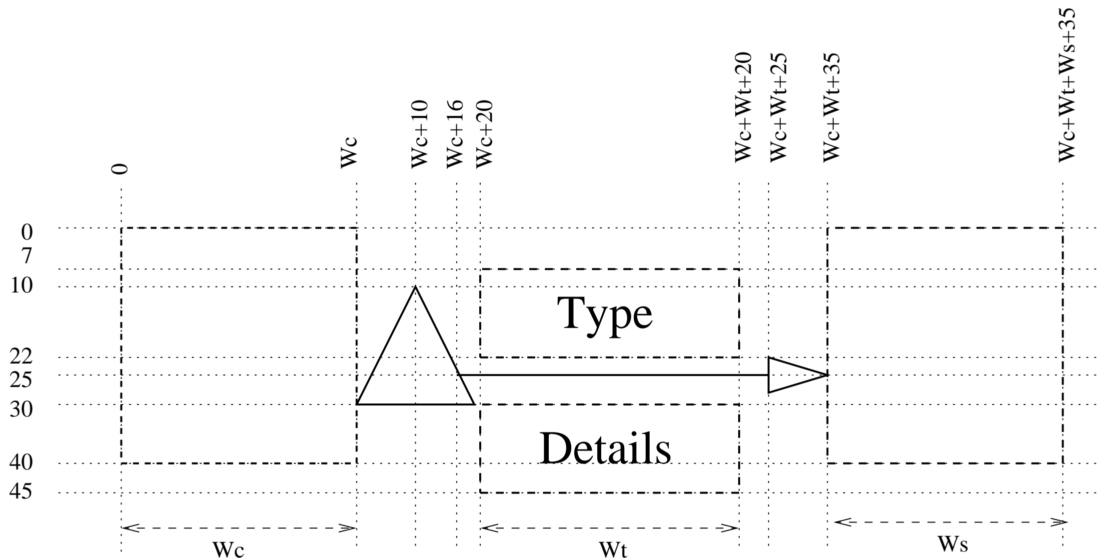


Figure 11.3.: The context-based constraint figure layout

Whenever the model is changed, the figures have to be redrawn. If a context-based constraint is changed, `modelChanged` is called. In this method, the new width and height of the entire figure is calculated, then `setBounds` is called which actually rearranges the subfigures to fill out the bounding box. If one of the roles `constrainedElements` or `scopedElements` (directly associated elements) has been changed, then the respective subfig should also be updated. This class does not have to worry about changes in the context condition, since `FigContextCondition` is notified directy by its owner if the model changes.

The inherited `makeSelection` method creates a `SelectionNodeClarifier`, allowing the user to move and resize the figure.

11.1.2. The Context Condition Subfigure

This figure is part of the context-based constraint figure and is used to depict an indirect association. Note that the `FigContextCondition` is not derived from `FigNodeModelElement`, since the metaclass is represents is not derived from `MModelElement`. It is also not a node, but a part the context-based constraint, so it should not be derived from `FigNode`. However, it should also be notified whenever its element in the model changes, so it should implement the interface `MElementListener` and register itself as an observer.

This figure is a special case: depending on its state, its appearance is different. If a baseclass is defined, the figure should draw the icon of the metaclass. Otherwise, it should draw a question mark. Underneath the icon, the query should be displayed in a rectangle with folded corners.

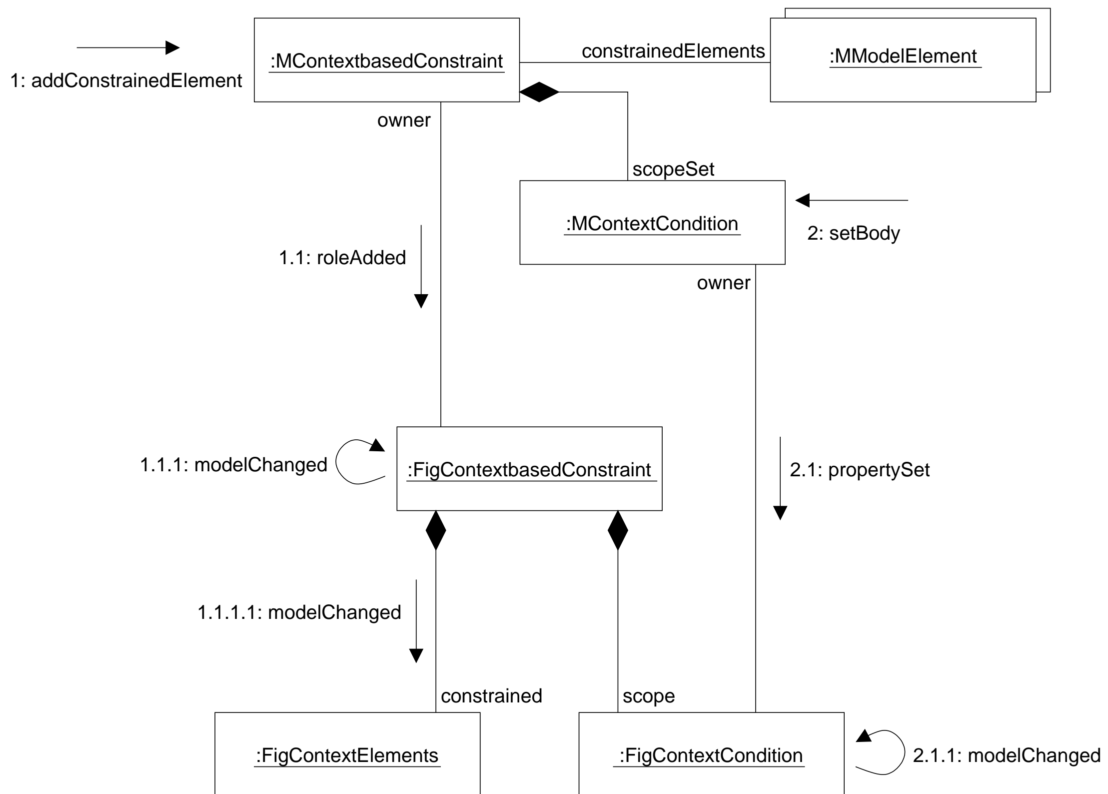


Figure 11.4.: When the model is changed, `FigContextCondition` is notified directly, `FigContextElement` via its `FigContextbasedConstraint`

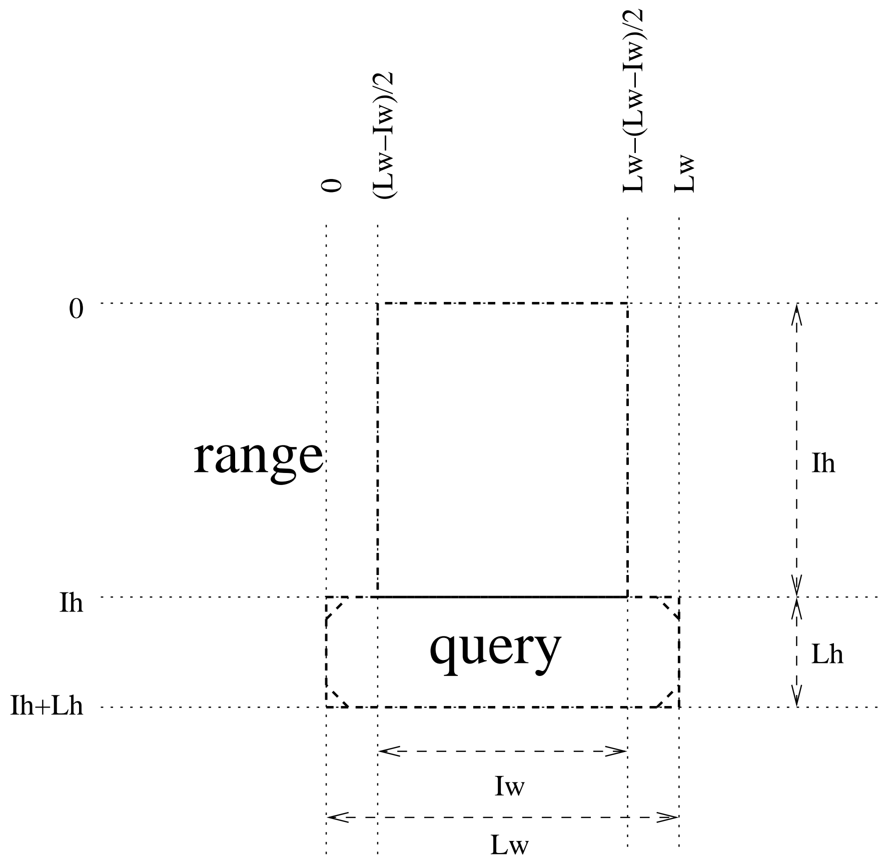


Figure 11.5.: The context condition figure layout

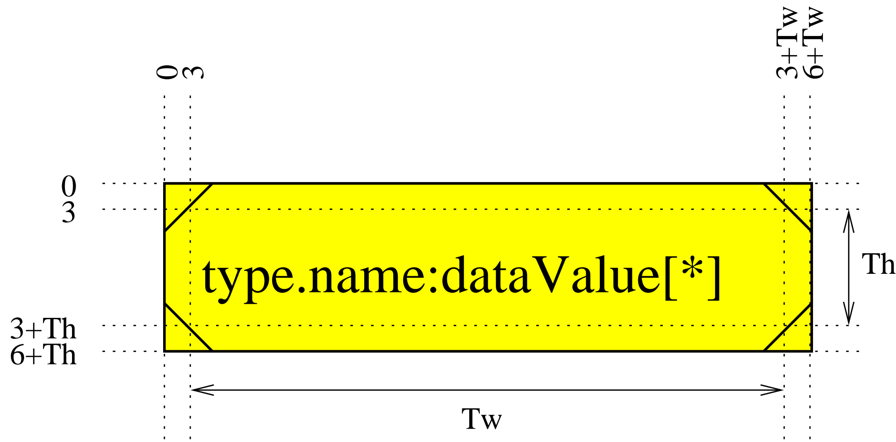


Figure 11.6.: The context property figure layout

Ih the height of the baseclass icon

Iw the width of the baseclass icon

Lh the height of the label containing the query string

Lw the width of the label containing the query string

11.1.3. The Context Elements Subfigure

This figure is part of the context-based constraint figure and is used to depict a direct association. Its appearance is dependent on the state of the roles `scopeElement` or `constrainedElement` in `MContextbasedConstraint`. This figure should appear as the symbol(s) of the metaclass(es) in these roles.

It is not necessary to implement the `MElementListener` interface in this figure, since the context-based constraint figure to which it belongs will already be notified whenever an element is added or removed from its roles. The `modelChanged` method of `FigContextBasedConstraint` should call `modelChanged` in `FigContextElements` so that this figure will be updated.

11.1.4. The Context Property Figure

This figure depicts a context property. It should display the context property's name (according to the `tagType` attribute in the tagged value's `type`) as well as the current values. The figure should be able to automatically resize itself to enclose the displayed text. Since this figure can appear in a variaty of diagrams, it is placed the `org.cocons.argo.diagram.ui` package.

The inherited implementation of `makeSelection` creates a `SelectionNodeClarifier` instance, allowing the user to move and resize the figure. The contents of the text

field should be automatically generated from the field values of the `MTaggedValue` this figure respesents.

The user should be able to the edit context properties of a model element directly in the diagram. As described in section 2.2, only certain context properties and values may be added to a model element, depending on its stereotype. Enforcing this in the user interface could become very complicated. The user must first define the stereotype, then the context property tag, and only then add context properties to the model element. If the stereotype definition is later changed, what should happen to the context properties already added? What if a context property tag is deleted? To avoid all of these problems, no valididation will be done by the user interface itself. This allows the user to add a context property, and later add a matching stereotype. Validating the context properties will be left up to a design critic, so errors in the configuration of context properties will not go unnoticed.

This simplifies the implementation of this class. If the enclosed text is edited, the `textEdited` method is invoked. This gives the class a chance to parse the text and update model[1]. If a context property tag with the given name does not already exists, then a new one should be created. If the model element has a stereotype, the new context property tag should be added to this stereotype, otherwise, it should be added to the namespace. If the edited text is not formatted correctly, the model should not be changed.

11.1.5. The Component Spec and Interface Type Figure

When a model element of type `Class` is shown in a component specification diagram (section 1.4), it's appearance is dependent on its stereotype. If the stereotype is <<interface type>>, it should be drawn as a lollipop, otherwise as a normal class rectangle, but without any methods or attributes.

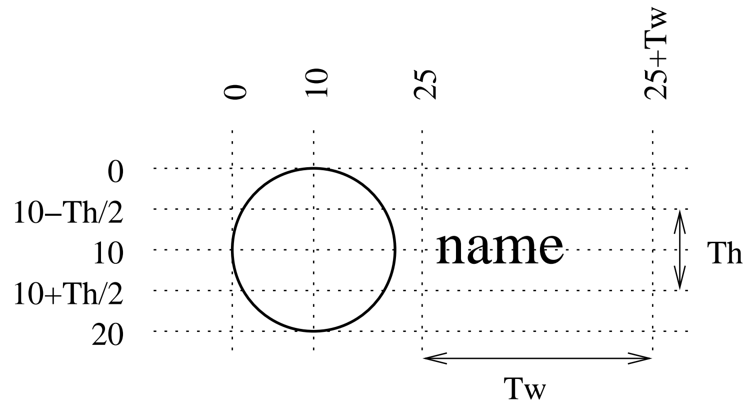
Since the GEF framework is based on the assumption that each node is rendered by exactly one figure, it would be very difficult to use two separate figures to render `Class` nodes (What would happen if the user changes the stereotype of a class? Any figures representing this class would have to *replace themselves* with other figures. Not an easy task in java, although the state pattern [Gamma+95, page 305] could be used here).

A more straightforward way to achieve this behavior is to create `FigClassSpec`, a subclass of `FigNodeModelElement` which can change its appearance. Whenever `modelChanged` is invoked, this class should update its appearance depending on the stereotype of its model element. If the stereotype is <<offers>>, the bottom 'lollipop' layout is used; in all other cases the top default layout is used. Since this figure is only used by the component specification diagram, it should be placed in the `component_spec.ui` package.

Since it makes no sense to allow the user to resize this figure, the `makeSelection` method should be overriden to return a new `SelectionMoveClarifiers` instance.

[1] Compare this to the `FigClass.textEdited` implementation. This class allows the user to edit attributes and methods directly in the class figure.

interface type class (stereotype <<inteface type>>)



component spec class (any other or no stereotype)

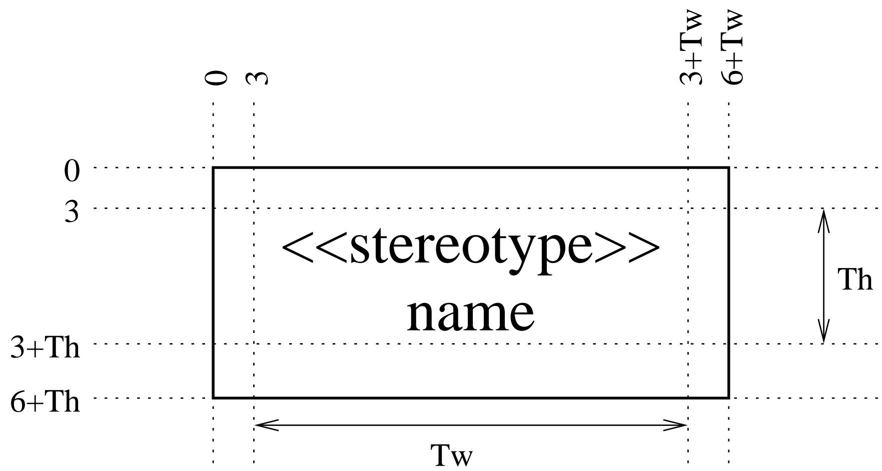


Figure 11.7.: The component spec and interface type layouts of `FigClassSpec`

export dependency (stereotype <<offers>>)

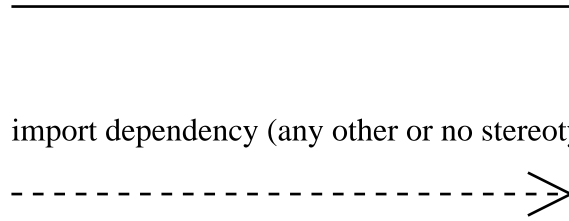


Figure 11.8.: The export and import interface dependency layouts

11.1.6. The Interface Dependency Figure

In the component specification diagrams, interfaces are linked to components with dependency relations (section 1.4 and [Cheesman+2001, section 3.9]). An export relation is explicitly marked with the <<offers>> stereotype and should be rendered as a solid line without an arrowhead. Import relations are rendered as a dashed line with an open arrowhead.

This `FigInterfaceDep` class (derived from `FigEdgeModelElement`) should override the `modelChanged` method, setting its style (`setDashed`) and arrowhead (`setDestArrowHead`) according to the stereotype of its model element.

11.2. The constraint diagram

The constraint diagram is where the user edits the context-based constraints contained in the model (section 2.3). This diagram can contain only one type of model element: the context-based constraint. For this reason, the constraint diagram's graph contains no edges, only nodes – each node representing a single context-based constraint.

Context-based constraints have both a graphical and a textual notation – this diagram is responsible for the graphical notation. The textual notation is implemented by the context-based constraint's property panel.

11.2.1. The Diagram class

The `CCLConstraintDiagram` class represents the constraint diagram itself. As a subclass of `UMLDiagram`, it can inherit most of the required implementation. Still, the following methods should be implemented:

CCLConstraintDiagram() the default constructor should call `setName`, setting the default name of the diagram. This name should be descriptive and unique, for example, 'CoCons Constraint Diagram N' where N is a natural number which is incremented for each instance.

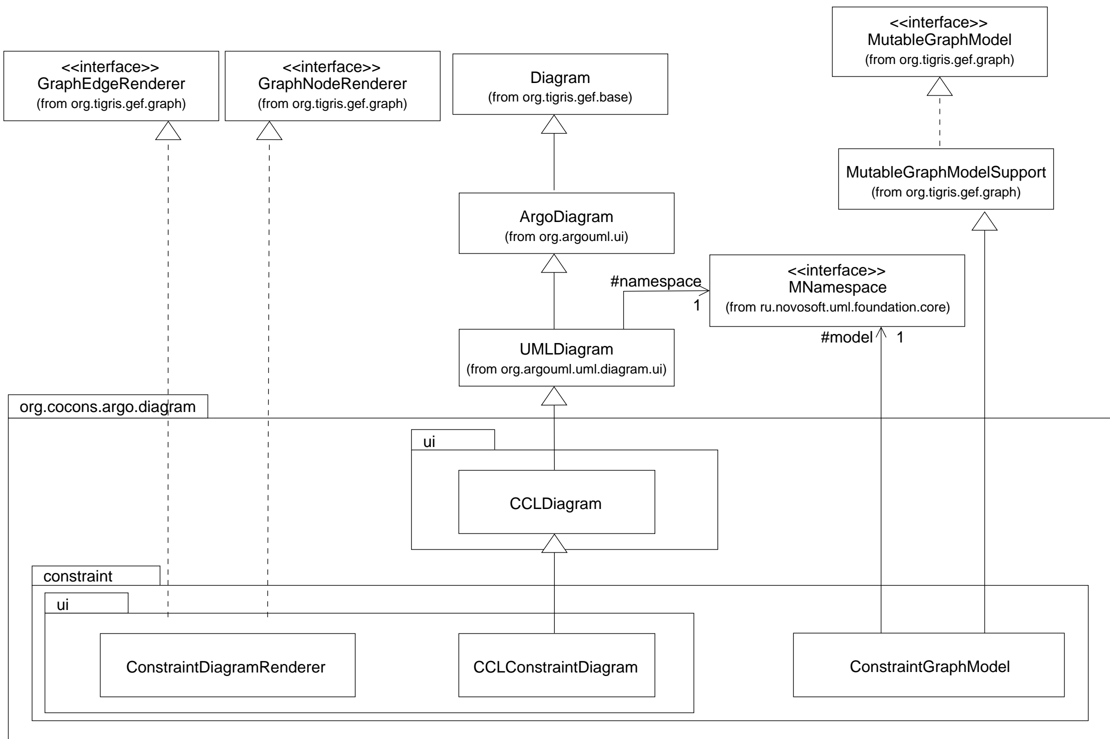


Figure 11.9.: The constraint diagram classes (figure classes not shown)

CCLConstraintDiagram(MNamespace n) should call the default constructor and then set the current model with `setNamespace(n)`.

setNamespace(MNamespace n) is called whenever the model for this diagram is set. Since this method overrides the superclasses method, it should call `super.setNamespace(n)`. Afterwards, it should:

1. create the proper graph model for this instance,
2. set the graph model's namespace to `n`,
3. add the graph model to the diagram with `setGraphModel`,
4. create a new `LayerPerspective` with the namespace and graph model,
5. add the `LayerPerspective` to the diagram,
6. create a new diagram renderer,
7. add the diagram renderer as both a node renderer and an edge renderer to the diagram.

initToolbar should create a new `Toolbar` instance, setting the `_toolbar` member inherited from `Diagram`. This toolbar should include, at the very least, a `CmdCreateNode` instance which creates new context-based constraints. Several actions are already instantiated by `UMLDiagram` any may also be added to the toolbar.

11.2.2. The Graph Model

The `ConstraintDiagramGraphModel` class is the adapter between the diagram and the user's model. It must implement the `GraphModel` interface, allowing it to visualize the current model, as well as `MutableGraphModel`, allowing the user to manipulate the model by editing the diagram. As a subclass of `MutableGraphModelSupport`, these interfaces are already partly implemented.

The `GraphModel` interface should be implemented as follows. Since the graph of this diagram does not have any edges, the edge and port methods should return null or empty values.

getNodes() should return all the context-based constraints in the diagram's graph.

getEdges() return empty vector

getPorts() return empty vector

getOwner(Object port) return null

getInEdges(Object port) return empty vector

getOutEdges(Object port) return empty vector

getSourcePort(Object edge) return null

getDestPort(Object edge) return null

addGraphEventListener(GraphListener listener) already implemented in superclass.

removeGraphEventListener(GraphListener listener) already implemented in superclass.

The `MutableGraphModel` interface should be implemented as follows.

canAddNode(Object node) should return true if the node to be added is a context-based constraint and is not already in the diagram.

addNode(Object node) should add the node to the graph as well as to the model (namespace), if not already present. Should also fire a node-added event.

removeNode(Object node) should remove the node from the graph, but not from the model. Should also fire a node-removed event.

canAddEdge(Object edge) return false

addEdge(Object edge) do nothing (should never be called – maybe throw an exception?)

removeEdge(Object edge) do nothing (should never be called – maybe throw an exception?)

canConnect(Object fromPort, Object toPort) return false

connect(Object fromPort, Object toPort, Class edgeClass) do nothing (should never be called – maybe throw an exception?)

addNodeRelatedEdges(Object node) do nothing

canDragNode(Object node) is already implemented in superclass to always return `false`, which disables dragging.

dragNode(Object node) is already implemented in superclass to do nothing, which disables dragging.

containsEdge(Object edge) is already implemented in superclass to use `getEdges()`.

containsNode(Object node) is already implemented in superclass to use `getNodes()`.

The other ArgoUML graph model classes also implement the `MElementListener` and `VetoableChangeListener` interfaces, although it is unclear whether they are actually used. Possibly they will be required for a future feature, so they should also be implemented in this graph model. Additionaly, the following methods should be implemented, as they will be called by the diagram class upon creation.

setNamespace(MNamespace m) is called when the namespace (usually the model) is set. Its implementation should register itself as a `MElementListener`, so it will be notified when the model is changed.

getNamespace() should return the current namespace.

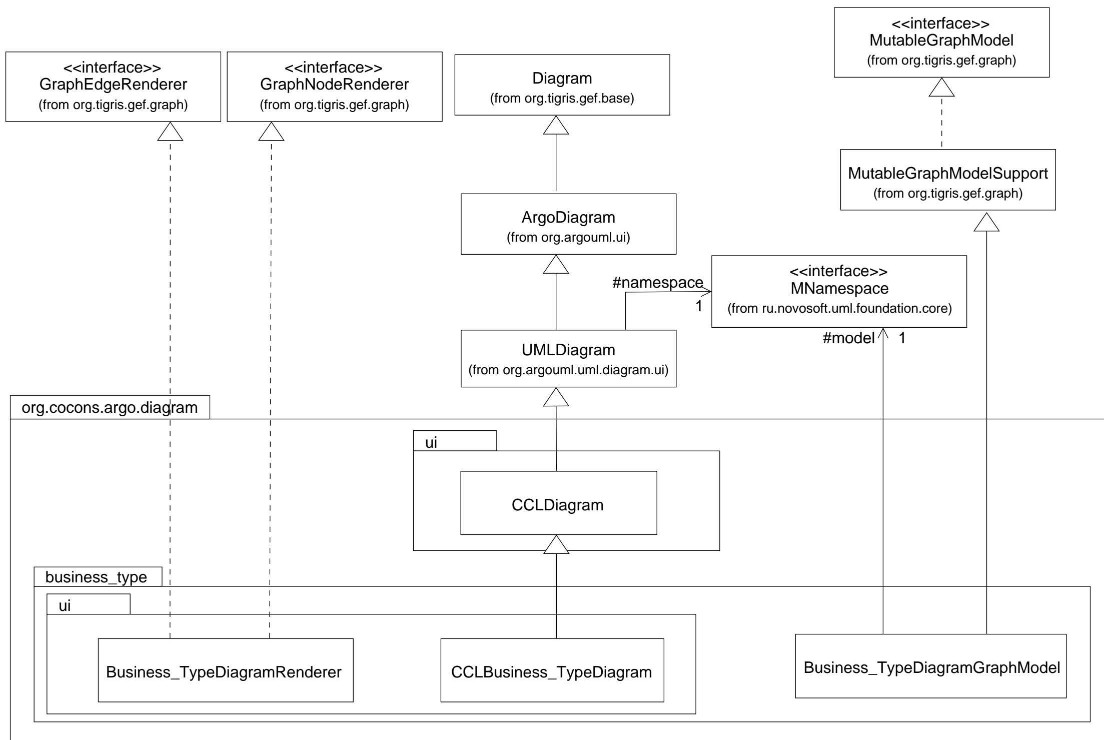


Figure 11.10.: The business type diagram classes (figure classes not shown)

11.2.3. The Diagram Renderer

The `ConstraintDiagramRenderer` class is responsible for creating the proper figure for any node or edge in the graph. Since the constraint diagram contains only nodes of context-based constraints, only the `getFigNodeFor(Object node)` must be implemented:

getFigNodeFor(Object node) if the node is a context-based constraint, return a new `FigContextbasedConstraint` representing this node. Otherwise, return null.

getFigEdgeFor(Object edge) return null.

11.3. The business type diagram

This diagram implements the global enhanced business type diagram described in section 1.3. It is very similar to UML's static structure diagram, and can contain many of the same elements. This diagram is more specialized, however, since the model elements it contains should be one of the business types (i.e. model elements of metaclass `Class`, but with one of the predefined business-type stereotypes).

The model elements this diagram can contain and the figures used to render this elements is shown in the following table:

UML metaclass	Figure Class	Graph Element
MClass	FigClass	Node
MPackage	FigPackage	Node
MGeneralization	FigGeneralization	Edge
MAssociation	FigAssociation	Edge
MDependency	FigDependency	Edge
MContextProperty	FigContextProperty	Node

Table 11.1.: Business type diagram figures and model elements

11.3.1. The Diagram class

The `CCLBusiness_TypeDiagram` class represents the business type diagram itself. As a subclass of `CoConDiagram`, it can inherit most of the required implementation. Still, the following methods should be implemented:

CCLBusiness_TypeDiagram() the default constructor should call `setName`, setting the default name of the diagram. This name should be descriptive and unique, for example, 'Business Type Diagram N' where N is a natural number which is incremented for each instance.

CCLBusiness_TypeDiagram(MNamespace n) should call the default constructor and then set the current model with `setNamespace(n)`.

setNamespace(MNamespace n) is called whenever the model for this diagram is set. Since this method overrides the superclass's method, it should call `super.setNamespace(n)`. Afterwards, it should:

1. create the proper graph model for this instance,
2. set the graph model's namespace to `n`,
3. add the graph model to the diagram with `setGraphModel`,
4. create a new `LayerPerspective` with the namespace and graph model,
5. add the `LayerPerspective` to the diagram,
6. create a new diagram renderer,
7. add the diagram renderer as both a node renderer and an edge renderer to the diagram.

initToolbar should create a new `Toolbar` instance, setting the `_toolbar` member inherited from `Diagram`. This toolbar should include

- `CmdCreateNode` instances which create classes and packages,
- `CmdSetMode` instances which switch to the `ModeCreatePolyEdge` mode, allowing the addition of generalization, dependency and association edges to the diagram,

- the `ActionAddAttribute` singelton (from `org.argouml.uml.ui`), which allows the user to add an attribute to the selected class,
- the `ActionAddContextProperty` instance (inherited from `CoConDiagram`), which allows the user to add a context-property to the selected class

Instead of a generic `CmdCreateNode` which adds an empty class to the model, it may be desirable to use specialized commands which create a class and automatically assign it one the business-type stereotypes. The `CmdCreateNodeStereotype` (from `org.cocons.argo.ui`) class can be used for this.

11.3.2. The Graph Model

The `Business_TypeDiagramGraphModel` class is the adapter between the diagram and the user's model. It must implement the `GraphModel` interface, allowing it to visualize the current model, as well as `MutableGraphModel`, allowing the user to manipulate the model by editing the diagram. As a subclass of `MutableGraphModelSupport`, these interfaces are already partly implemented.

The `GraphModel` interface should be implemented as follows.

getNodes() should return all the nodes in the diagram's graph.

getEdges() should return all the edges in the diagram's graph.

getPorts() since ArgoUML assumes that each node and each edge is one big port, should simply return all edges and ports in the diagram's graph.

getOwner(Object port) since ArgoUML assumes that each node and each edge is one big port, should simple return the object itself.

getInEdges(Object port) should return all edges currently connected with the given node. Since the graph is not considered directed, should return the same as `getOutEdges()`.

getOutEdges(Object port) should return all edges currently connected with the given node. Since the graph in not considered directed, should return the same as `getInEdges()`.

getSourcePort(Object edge) should return the first node of the given edge. This can be considered as the child of a generalization, the first end of an association or the client of a dependency

getDestPort(Object edge) should return the last node of the given edge. This can be considered as the parent of a generalization, the last end of an association or the supplier of a dependency

addGraphEventListener(GraphListener listener) already implemented in superclass.

removeGraphEventListener(GraphListener listener) already implemented in superclass.

The `MutableGraphModel` interface should be implemented as follows.

canAddNode(Object node) should return `true` if the node is one of the metaclasses in table 11.1 and is not already in the diagram.

addNode(Object node) should add the node to the graph as well as to the model (namespace), if not already present. Should also fire a node-added event.

removeNode(Object node) should remove the node from the graph, but not from the model. Should also fire a node-removed event.

canAddEdge(Object edge) should return `true` if the edge is one of the metaclasses in table 11.1, both ends of the edge are `MClass` nodes and the edge is not already in the diagram.

addEdge(Object edge) should add the edge to the graph as well as to the model (namespace), if not already present. Should also fire a edge-added event.

removeEdge(Object edge) should remove the edge from the graph, but not from the model. Should also fire an edge-removed event.

canConnect(Object fromPort, Object toPort) should return `true` if both ports are `MClass` nodes.

connect(Object fromPort, Object toPort, Class edgeClass) should create a new instance of `edgeClass`, setting each end of the edge to `fromPort` and `toPort`. Depending on the `edgeClass`, this will be the ends of an association, the child and parent of a generalization or the client and supplier of a dependency.

addNodeRelatedEdges(Object node) should add an new edge for each generalization, association and dependency connected to `node` and not already in the diagram.

canDragNode(Object node) is already implemented in superclass to always return `false`, which disables dragging.

dragNode(Object node) is already implemented in superclass to do nothing, which disables dragging.

containsEdge(Object edge) is already implemented in superclass to use `getEdges()`.

containsNode(Object node) is already implemented in superclass to use `getNodes()`.

As with the constraint diagram, the unused `MElementListener` and `VetoableChangeListener` interfaces should be implemented. Additionaly, the following methods should be implemented:

UML Metaclass	Figure Class	Graph Element
MClass	FigClassSpec	Node
MDependency	FigInterfaceDep	Edge
MContextProperty	FigContextProperty	Node

Table 11.2.: Component specification diagram figures and model elements

setNamespace(MNamespace m) is called when the namespace (usually the model) is set. Its implementation should register itself as a `MElementListener`, so it will be notified when the model is changed.

getNamespace() should return the current namespace.

11.3.3. The Diagram Renderer

The `Business_TypeDiagramRenderer` class is responsible for creating the proper figure for any node or edge in the graph. The `GraphNodeRenderer` and `GraphEdgeRenderer` interfaces should be implemented to return the values shown in table 11.1.

Since business types typically do not have methods, `setOperationVisible(false)` should be called immediatly after creating figures for classes. This will only set the default appearance of classes in this diagram – the user can still display the operations by checking the appropriate box the the class property panel.

11.4. The component specification diagram

This diagram implements the enhanced component specification diagram described in section 1.5. Although the graphical notation used in this diagram is not part of the standard UML specification, the underlaying model is. As detailed in [Cheesman+2001, section 3.9], the interface 'lollipops' are actually classes with the stereotype <<interface type>>. The relation between a component specification and an interface specification is modeled using UML dependencies. Export interfaces are explicitly marked with an <<offers>> stereotype.

As shown in table 11.2, this diagram does not use the normal figure classes to visualize the model elements. The figure classes in this diagram must be able to change their appearance when the stereotype of their model element changes. This stereotype-dependent behavior is encapsulated in the respective figure classes.

Since the design of this diagram is very similar to the business type diagram in the previous section, only the differences will be mentioned here.

11.4.1. The Diagram class

The only difference between the `CCLComponent_SpecDiagram` class and the `CCLBusiness_TypeDiagram` class is the contents of the toolbar. This toolbar should contain at least the following actions:

- a `CmdCreateNodeStereotype` instance which creates a new class with the stereotype <<comp spec>>.
- a `CmdCreateNodeStereotype` instance which creates a new class with the stereotype <<interface type>>.
- a `CmdSetMode` instance which creates a new dependency without a stereotype, allowing the user to link an interface type to a component specification as an import interface.
- a `CmdCreateEdgeStereotype` instance which creates a new dependency with the stereotype <<offers>>, allowing the user to link an interface type to a component specification as an export interface in one step.
- the `ActionAddContextProperty` instance which adds a context property to the selected model element.

11.4.2. The Graph Model

The `Component_SpecDiagramGraphModel`'s methods should be implemented as follows.

canAddNode(Object node) should only return true if the node matches one of the classes and stereotypes in table 11.2.

canAddEdge(Object edge) should only return true if the edge matches one of the classes and stereotypes in table 11.2.

canConnect(Object fromPort, Object toPort) should only return true if `fromPort` is a <<comp spec>> and `toPort` is an <<interface spec>> class[2].

connect(Object fromPort, Object toPort, Class edgeClass) should create a new dependency (`edgeClass` should always be `MDependencyImpl`), setting the supplier to `fromPort` and the client to `toPort`.

11.4.3. The Diagram Renderer

The `Component_SpecDiagramRenderer` methods should be implemented to return the figures shown in table 11.2. The stereotype-dependent appearance of the metaclasses are encapsulated in the figure classes themselves.

11.5. The interface specification diagram

This diagram implements the enhanced interface specification diagram described in section 1.4. This interface types in this diagram are the same model elements as in the

UML Metaclass	Figure Class	Graph Element
MClass	FigClass	Node
MAssociation	FigAssociation	Edge
MContextProperty	FigContextProperty	Node

Table 11.3.: Interface specification diagram figures and model elements

component specification diagram, but they are shown here in more detail, using the standard figure for representing classes.

Since the design of this diagram is very similar to the business type diagram, only the differences will be detailed here.

11.5.1. The Diagram class

The only difference between the `CCLInterface_SpecDiagram` class and the `CCLBusiness_TypeDiagram` class is the contents of the toolbar. This toolbar should contain at least the following actions:

- a `CmdCreateNodeStereotype` instance which creates a new class with the stereotype <<interface spec>>.
- a `CmdCreateNodeStereotype` instance which creates a new class with the stereotype <<type>>.
- a `CmdCreateNodeStereotype` instance which creates a new class with the stereotype <<info type>>.
- a `CmdCreateEdgeStereotype` instance which creates a new association.
- the `ActionAddContextProperty` instance which adds a context property to the selected model element.

11.5.2. The Graph Model

The `Interface_SpecDiagramGraphModel` methods should be implemented as follows:

canAddNode(Object node) should only return `true` if the node is an instance of `MClass` and the stereotype is either <<interface spec>>, <<type>> or <<info type>>.

canAddEdge(Object edge) should return true if the edge is an instance of `MAssociation`.

canConnect(Object fromPort, Object toPort) should only return true if both ports are instances of `MClass`.

[2]This will not prevent the user from retroactively changing the stereotype to an invalid configuration. A design critic could be created to catch this kind of error.

connect(Object fromPort, Object toPort, Class edgeClass) should create a new association (`edgeClass` should always be `MAssociation`), setting the association ends to `fromPort` and `toPort`.

11.5.3. The Diagram Renderer

The `Interface_SpecDiagramRenderer` methods should be implemented to return the figures shown in table 11.3.

11.6. The Diagram Actions

The diagrams in the previous sections add various actions to their toolbars. Most of the actions are already defined by GEF and ArgoUML, but the following action classes give the user new possibities to edit the diagram.

The actions which allow the user to create a stereotyped model element are purely convenience actions – the user would get the same result by adding a model element and then changing the stereotype in the property panel. Allowing the user to add a context property is a new functon, based on the concepts introduced in the CoCons metamodel.

11.6.1. Creating Stereotyped Nodes

Normally, a diagram uses the `CmdCreateNode` action, which creates a new object of a specified type (e.g. `MClassImpl`) before the user adds it to the diagram. This can be extended by adding a subclass `CmdCreateNodeStereotype`, which allows the user to create a new model element with a predefined stereotype in one step.

The constructor of this subclass should take one additional argument: the stereotype which new model elements should have. It is up to the caller (usually the diagram) to ensure that the stereotype exists in the current namespace. The `makeNode` method should be overridden to set the stereotype of the newly created model element before returning it to the caller.

11.6.2. Creating Stereotyped Edges

The action which creates edges is more complicated – the `CmdSetMode` action does not create new edges itself, but creates a new mode (usually `ModeCreatePolyEdge`), which is responsible for creating the new edge (actually the mode uses the graph model's `connect` method, but this is not important here).

The `CmdSetMode`'s constructor accepts arguments, which are later passed on to the newly created mode. `ModeCreatePolyEdge` expects a single argument named 'edgeClass', whose value is the class of the edge to be created (e.g. `MDependencyImpl`).

In order to extend the edge creating step, it is therefore necessay to create a new mode which not only creates a new edge, but also sets its stereotype. This new mode, `ModeCreateEdgeStereotype`, should be derived from `ModeCreatePolyEdge`. It should

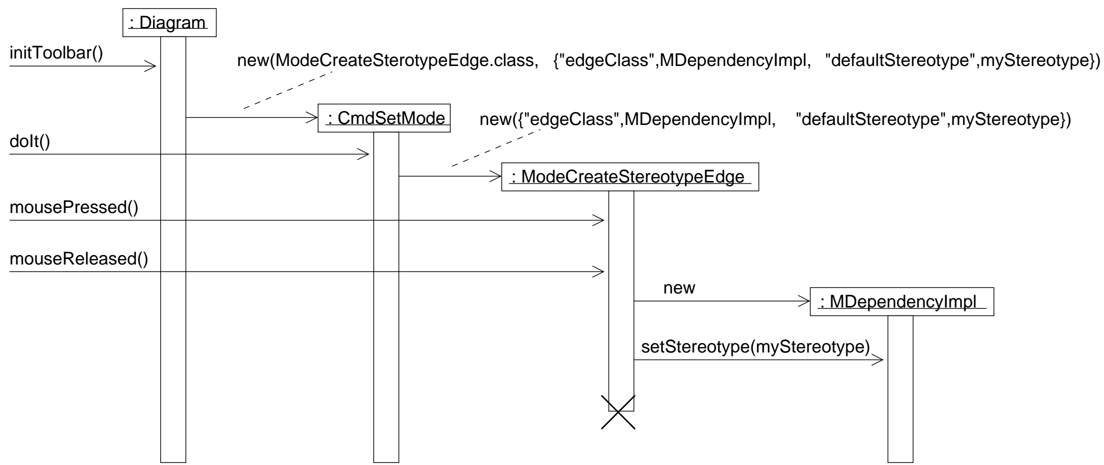


Figure 11.11.: `CmdSetMode` and `ModeCreateEdgeStereotype` work together to create new edges

be able to interpret an additional argument 'defaultStereotype' and set the stereotype of the newly created edge to this value.

It is then up to the diagram to specify the stereotype to be used when creating the action instance for its toolbar. The action passes this argument to the mode when the action is triggered.

11.6.3. Adding Context Properties

The user should be able to add context properties to a model element directly in the diagram. This action should be able to add a `FigContextProperty` to the diagram, connect the figure to the model element figure with a line, and add an empty tagged value to the model[3]. The user can then edit the text field in the figure directly. Parsing the text in order to update the context property is then handled by the `FigContextProperty` class.

11.7. Integrating the diagrams in ArgoUML

New menu items must be added to the application menu bar to enable the user to use the new diagrams. These new menu items are represented by subclases of `UMLAction`. Whenever a menu item is selected, its `actionPerformed` method is invoked. To enable the user to create a new diagram, this method should to implemented to:

1. create a new instance of the specific diagram class (e.g. `CCLConstraintDiagram`) for the current model.

[3]This is very similar to the `ActionAddNote` action, which allows the user to add a note to a model element

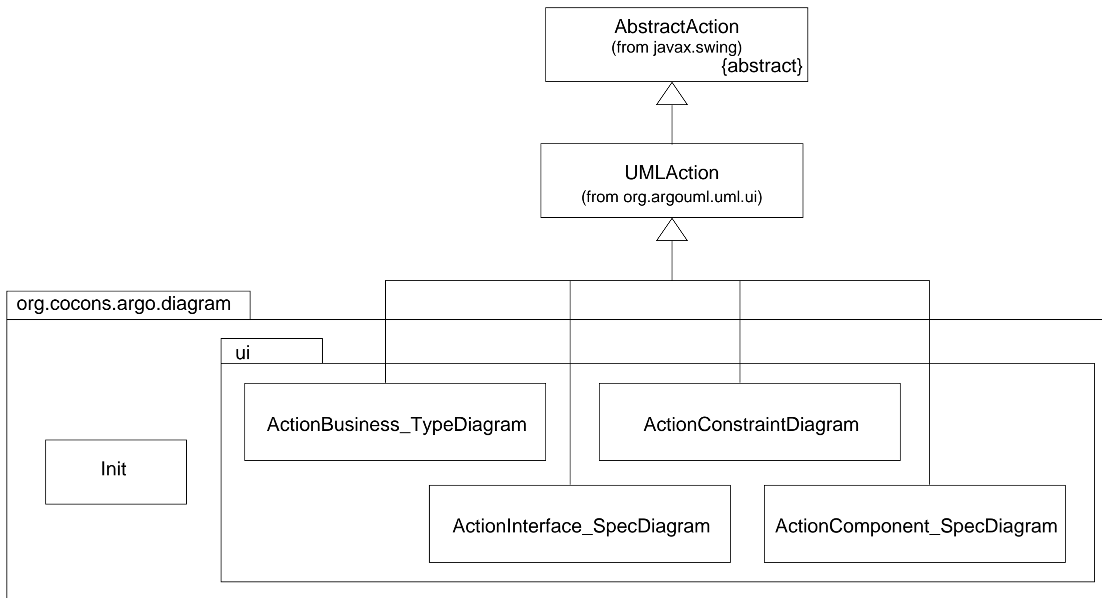


Figure 11.12.: Diagram menu item classes

2. add the new diagram to the current project.
3. add the diagram to the navigator pane's history stack so the user can undo the following target change.
4. set the project browser's target to the new diagram.
5. notify the project that unsaved changes have occured.
6. notify `allActions` that the project state has changed.

The `Init` class will be used by the plugin initialization (section 12.4) and is responsible for creating the actions and adding them to the appropriate pull-down menu. This can either be done in `Init`'s constructor (then then new menu items will be available immediatly at the cost of a slightly longer application load time) or in the `run` method (faster application loading, but the menu items will not appear until after the post-load phase).

12. Extending the ArgoUML Workspace

The diagrams may be the main user interface component[1] of ArgoUML, but the other components are just as important. These components must also be extended to support the CoCons metaclasses and other CCL concepts.

Although the graphical representation of a model does have its merits, some aspects of a model can not be adaquately represented in a visual notation. One example of this is the definition of stereotype. For this reason, ArgoUML includes a *Details Pane* which is displayed under the diagram. Section 12.1 describes how to extend this pane to support the new CoCons metaclasses.

To the left of the diagram pane is the *Navigator Pane*, showing the user's model in a tree-like structure. This pane allows the user to quickly navigate to any part of the model. Section 12.2 shows which classes must be added to enable this pane to support the new CoCons metaclasses.

All of these components are based on java's Swing class library. Since this library is very well documented ([Java2API] and [Swing]) this section will concentrate on the specification of the new UI components. Working out the design details of these UI components should be quite straightforward for any experienced java/Swing developer.

12.1. Extending the Details Pane

The details pane (see section 8.2) contains several tabs which show different aspects of the user's model. By implementing `TabModelTarget`, these tabs are notified whenever the user selects a new object (the *target*). The target can be an element from the user's model or anything else which can be selected in the navigator panel (e.g. a diagram).

Most of these tabs have constant contents (e.g. the 'constraints' tab) and are enabled whenever they are valid for the current target. This is a good position to add the 'tagged values' tab, since this tab should be available whenever the target is an instance of a `MModelElement`.

One of the details pane's tabs is the 'properties tab' – this tab shows the attributes and roles of the current target. Since each target has different attributes and roles (depending on its metaclass) the contents of this tab must be dynamic. This is achieved through *property panels* – each metaclass has its own property panel (derived from `PropPanel`) which supplies this tab with the appropriate contents.

[1]In this chapter, in order to impove readability, 'component' is used to refer to a 'Swing UI component'. This is not the same as the 'software component' concept.

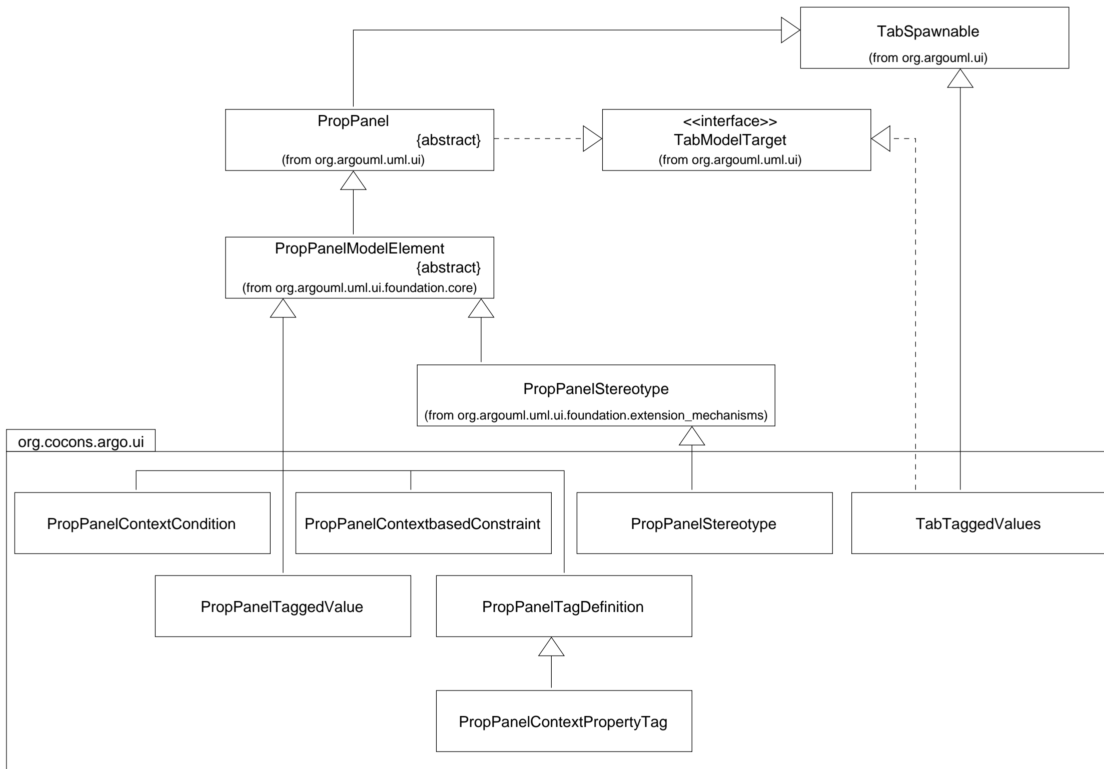


Figure 12.1.: The plugin's details pane classes

Tagged Values

Tag	Owner	Values
Personal Data	comp spec	
Tier	comp spec	Internet
Workflow		CreateNewCustomer, CreateNewContract
Author		Al Bundy

Figure 12.2.: The tagged values tab layout

Each metaclass should have its own property panel showing the details of the selected element, allowing the user to modify the attribute values and links well as navigate through the model. The property panels are subclassed from the `PropPanelModelElement` class.

These property panels classes should be registered with the details pane by the plugin (appendix A.2.3) to enable ArgoUML to display them as required. If this is done in the `Init.run` method (section 12.4) these property panels will replace any of ArgoUML's built-in panels which are registered for the same metaclass.

Each property panel consists of various sub-components. In the `org.argouml.uml.ui` package, ArgoUML already defines many specialized UI components which can be mapped to roles and attributes in the user's model.

12.1.1. The Tagged Values Tab

This tab displays all the tags for the currently selected model element (ArgoUML already has a tagged values tab, but this class only supports UML 1.3-style tagged values). Since context properties are modeled using tagged values (section 2.2), the context properties are displayed as well.

This tab contains a table (a swing `JTable`) showing all the tags the selected model element may have, what stereotype defines them, and any current values. An icon is shown in the first column if the tag definition is a context property tag.

Double-clicking on a tag should change the target to the selected tag defintion, which results in the tag definition property panel (or the context property tag panel) being displayed. Double clicking on the stereotype should change the target to the stereotype. The last column of the table should be editable, allowing the user to edit the tagged values directly.

12.1.2. The Stereotype Property Panel

ArgoUML's stereotype property panel does not support tag definitions, so a new stereotype property panel is required. This new panel can inherit all the UI components from ArgoUML's built-in stereotype panel and simply add a new label and list box to the layout showing the tag definitions.

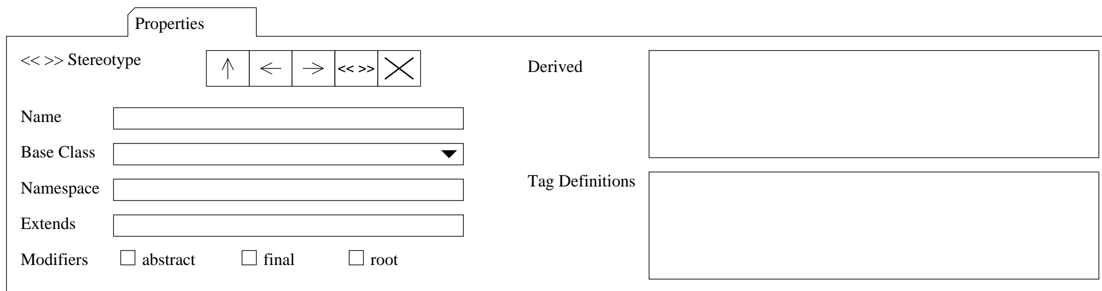


Figure 12.3.: The stereotype panel layout

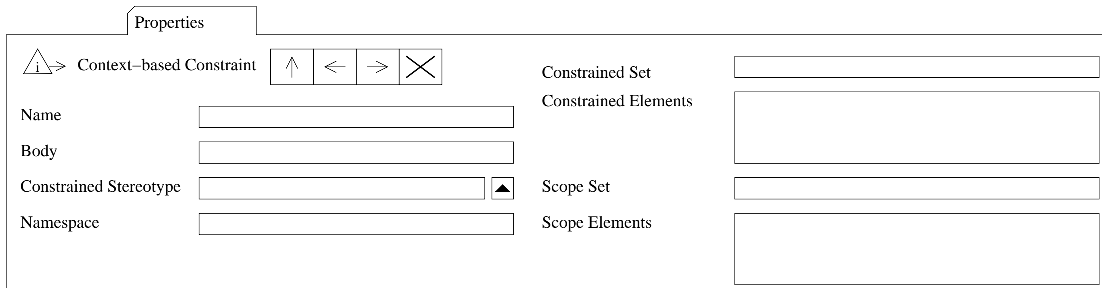


Figure 12.4.: The context-based constraint property panel layout

As with the tagged values tab, this list box should show context property tags with the appropriate icon. This list box should also support a context-menu with the following items:

open sets the target to the selected entry. This will display either the tag definition property panel or the context property tag panel, depending on the type of the entry.

add tag definition creates a new, empty tag defintion and adds it to the stereotype.

add context property creates a new, empty context property and adds it to the stereotype

delete removes the selected entry from the stereotype

Double-clicking on an item in this list-box should be equivalent to selecting 'open' from the context-menu.

12.1.3. The Context-based Constraint Property Panel

This panel shows the attribues and roles of a context-based constraint in its textual form.

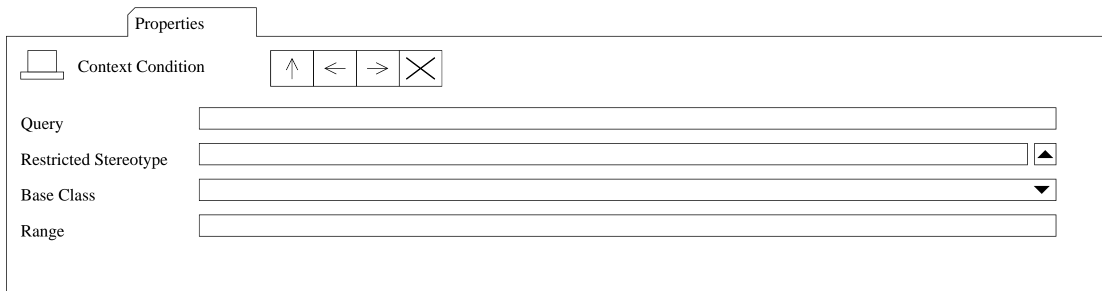


Figure 12.5.: The context condition property panel layout

Contained UI Components:

Name uses `nameField` inherited from superclass.

Body uses `UMLTextField` mapped to `MContextbasedConstraint.body`.

Constrained Stereotype a new class derived from `UMLComboBox` is required for this component allowing the user to select a stereotype.

Namespace uses `namespaceScroll` inherited from superclass.

Constrained Set / Scope Set uses `UMLList` with `UMLReflectionListModel`. This non-editable field can show the context-condition in its textual form. A double-click should change the target to the context condition.

Constrained Elements / Scoped Elements uses `UMLList` with `UMLReflectionListModel`. A double-click should change the target to the selected model element. The user interaction necessary to add a new model element to the list is a topic for further reseach.

12.1.4. The Context Condition Property Panel

This panel shows the attributes and roles of a context condition in its textual form.

Contained UI Components:

Query uses `UMLTextField` mapped to `MContextCondition.query`.

Restricted Stereotype new class derived from `UMLComboBox` is required for this component.

Base Class uses `UMLMetaclassComboBox`.

Range uses `UMLMultiplicityComboBox`.

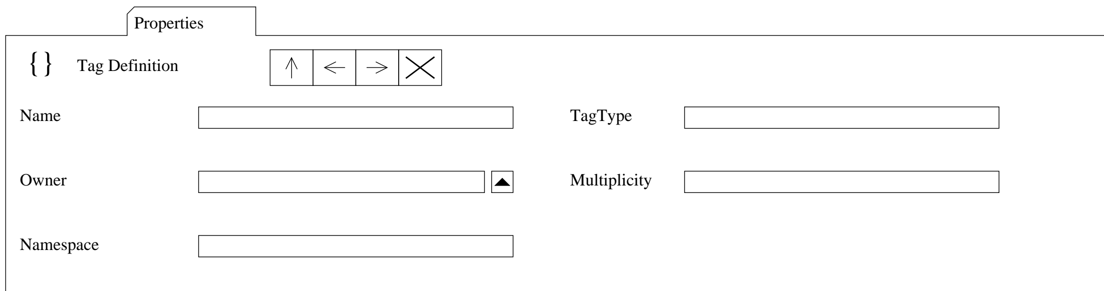


Figure 12.6.: The tag definition property panel layout

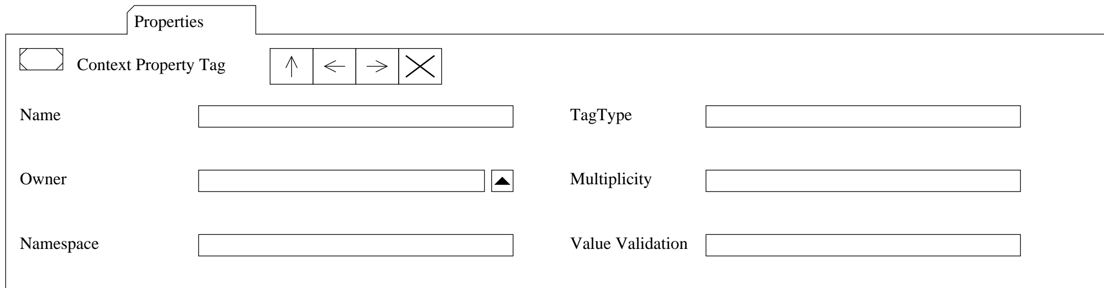


Figure 12.7.: The context property tag panel layout

12.1.5. The Tag Definition Property Panel

This panel shows a UML 1.4 tag definition.

Contained UI Components:

Name uses `nameField` inherited from superclass.

Owner a new class derived from `UMLComboBox` is required for this component allowing the user to select a stereotype.

Namespace uses `namespaceScroll` inherited from superclass.

TagType uses `UMLTextField` mapped to `MTagDefinition.tagType`.

Multiplicity uses `UMLMultiplicityComboBox`.

12.1.6. The ContextProperyTag Property Panel

This panel shows a context property tag.

As a subclass of the tag defintion panel, only the ValueValidation component must be added:

ValueValidation uses `UMLList` with `UMLReflectionListModel`. This non-editable field can show constraint which validates the context properties. A double-click should change the target to the constraint.

12.2. Extending the Navigator Pane

ArgoUML's navigator pane (section 8.3) shows the user's model in various tree-like perspectives. These perspectives should be extended to support the new metaclasses.

The 'package-centric' perspective needs new branching rules to show the context-based constraints. Since a constraint can either be owned by a namespace or a stereotype, the following rules are required:

Namespace-to-Context-based Constraint Since a model is derived from namespace, this rule enable the perspective to show all the context-based constraints not owned by a stereotype. This rule can be constructed with a `GoModelElement` rule combined with a `GoFilterChildren` instance configured to return only context-based constraints.

Namespace-to-Stereotype will enable a perspective to show all stereotypes in a namespace. This rule can be constructed with a `GoModelElement` rule combined with a `GoFilterChildren` instance configured to return only stereotypes.

Stereotype-to-Context-based Constraint will enable a perspective to show all context-based constraints owned by a stereotype. This rule requires a new class implementing the `TreeModelPrereqs` interface. It should follow the `stereotypeConstraint` role of a stereotype.

Another possibility would be a new 'CoCons-centric' perspective showing the context-based constraints and context-property tags in the model:

Project-to-Model forming the root of the tree. This is already implemented by the `GoProjectModel` class.

Namespace-to-Package showing the package hierarchy in the model. This rule is already instantiated by `NavPerspective` as `modelToPackages`.

Namespace-to-Stereotype showing all stereotypes in a namespace.

Namespace-to-Context-based Constraint showing the context-based constraints not owned by a stereotype.

Stereotype-to-Context-based Constraint showing the context-based constraints owned by a stereotype.

Namespace-to-Context Property Tag showing the context property tags not owned by a stereotype. This rule can be constructed with a `GoModelElement` rule combined with a `GoFilterChildren` instance configured to return only context property tags.

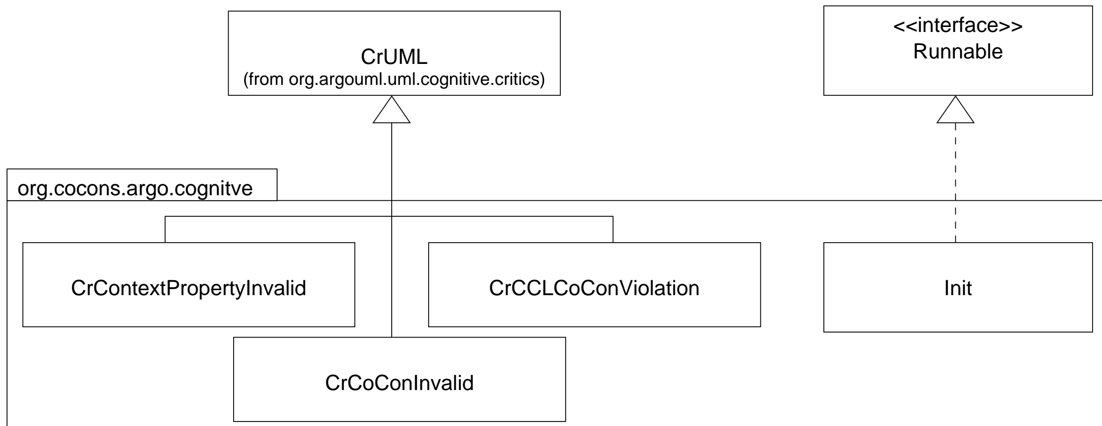


Figure 12.8.: CCL design critics

Stereotype-to-Context Property Tag showing the context property tags owned by a stereotype. The rule also requires a new class following the `definedTag` role of a stereotype. A `GoFilterChildren` instance should then remove any tag definitions which are not context property tags.

As the above examples illustrate, there are many possibilites. As a rule of thumb, each branching rule is encapsulated in a class implementing `TreeModelPrereqs` and should be able to follow exactly one role in the metamodel. The `GoModelElements` class, for example, follows the `ownedElements` role of a `MNamespace`.

New perspectives are then simply a matter of combining the rules to different configurations with the `addSubTreeModel` method. The `NavPerspective` static methods `registerRule` and `registerPerspective` add the rules and perspectives to the system. This should be done in the `Init.run` method (package `ui`) which will be called upon application startup (section 12.4).

12.3. Adding new Design Critics

ArgoUML's cognitive support and the design critics mechanism (section 8.4) make it possible for ArgoUML to check the user model for consistency. By adding new design critics ArgoUML can warn the user if certain configurations are discovered in the model.

Having all design critcs constantly checking all model elements would be very wasteful. For this reason, each critic is registered at the `Agency` to be activate when certain metaclasses are modified.

CrContextPropertyInvalid checks whether the context properties of a model element are well-formed. Should be registered to react to tagged values, context property tags and constraints. If the triggered class is a tagged value, this critic should check whether its type is a context property tag. If the triggered class is a constraint,

it should be the value validation of a context property tag. Otherwise, the critic should immediatly exit in order not to waste processing resources.

CrCoConInvalid checks whether context-based constaints are well-formed. Should be registered to react to context-based constraints and context conditions.

The definitions of 'well-formedness' can be found in [Bübl2001-CoCons].

CrCoConViolated checks whether any model elements violate a context-based constraint. The indirect nature of context-based constraints make this critic very processor-intensive, since it must be activated whenever any model element it modified. Possibile optimization strategies are a further research topic.

The `Init` class (used by the plugin package, section 12.4) is responsible for registering these critics and should be performed in the **run** method, as to not increase application loading time. The method `Agency.register` is used to register a single critic to react to a single metaclass – by calling this method multiple times the same critic can react to multiple metaclasses.

12.4. Plugin Initialization

Whereas section 9.1 outlines the plugin mechanism from an abstract, generic perspective, this section will give a more detailed description of the concrete plugin mechanism and specificy the interface to the plugin.

ArgoUML uses a two-phase initialization: the **main** method of the `Application.Main` class performs the first phase by instatiating and initializing the classes the user will immediatly need (e.g. the user interface panes). In order to avoid a long delay when starting ArgoUML, a second phase, running in a background thread, is used to preload classes and build up the various registries. In this second phase, the property panels are registered with the details pane, the design critics are started, the navigator perspectives are constructed and many UML metaclasses are preloaded.

The ArgoUML Plugin interface is based on the plugin architecture introduced in 9.1. To keep things simple, ArgoUML requires the plugin to implement the class `ArgoPlugin` which has only two methods: `init()` and `getPostLoadActions()`.

init() in invoked immediately after the plugin is loaded.

getPostLoadActions() should return a vector of post-load actions implementing `Runnable`. These actions run after all application post-load actions are executed.

With the `getPostLoadActions` method, the plugin is able to add its own initialization classes to the second phase. This is necessary since many of the structures created by the application during this phase must be modified by the plugin.

Each of the plugin's main package groups export a class `Init` implementing `Runnable`. The plugin instantiates these classes and passes them to the application – encapsualting the initialization details in the individual packages.

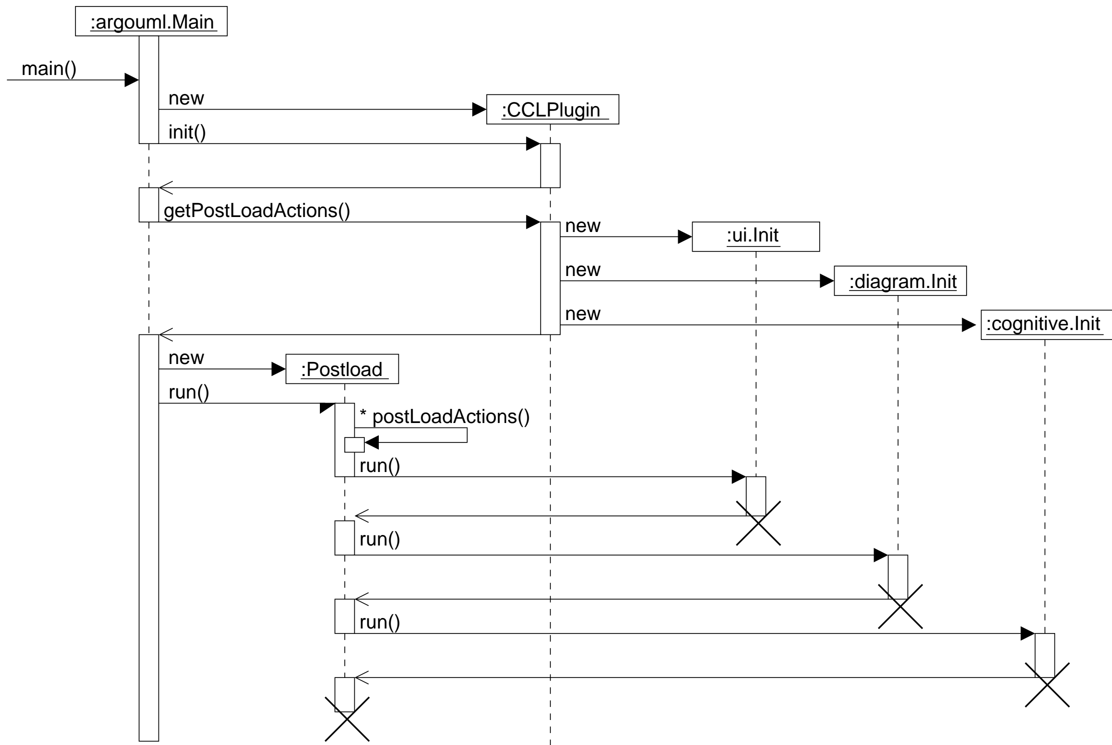


Figure 12.9.: The plugin's `Init` objects are executed in a background thread

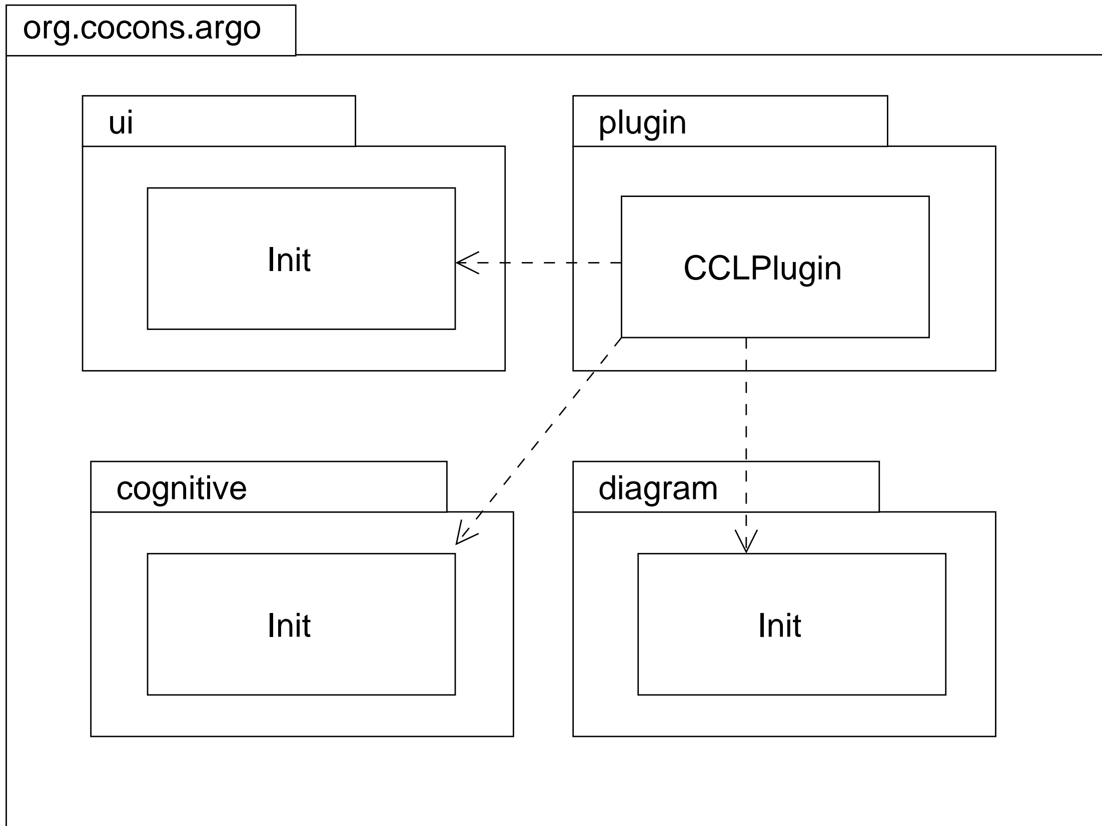


Figure 12.10.: Each top-level package has its own `Init` class

13. Conclusion

The open source modelling tool ArgoUML served as a starting point for this work - the availability of the source code making extensions possible. As with many open source projects, up-to-date documentation describing the overall design was far from sufficient. Part II of this work, the analysis of the ArgoUML architecture, resulted out of necessity - and will hopefully help other developers in the ArgoUML community.

Software components are considered as the main structuring mechanism in order to support continuous software engineering [Große-Rhode+2000]. By extending the ArgoUML modelling tool with the plugin mechanism, ArgoUML is transformed from a monolithic application to a component-based system, allowing plugins to be developed and distributed independently from ArgoUML itself. The resulting system can serve as an experimental platform for furthur research in the area of model based software development.

Designing a modelling tool supporting the context-based constraint language was the primary motivation for this thesis. The CCL-Plugin serves not only as a proof-of-concept for the plugin mechanism, but also opens the possibility to evaluate the advantages of context-based constraints in real-world modelling projects.

A. Necessary Source Modifications

A.1. Bugfix in package ru.novosoft.uml.undo

The member variable `MCheckPoint.stackPosition` should always reflect the current position of this checkpoint on the undo-stack. When items are removed from somewhere in the middle of the stack (which happens in the `MCheckPointUndoManager.forget` method) the position of all objects 'above' the removed object change – but their stack-Position variable is not updated:

```
/** forget information about all changes done before this checkpoint */
public static void forget(MCheckPoint c)
{
  // ...
  for (int i = currentCheckPointPosition-1;i>=0;i--)
  {
    undoStack.remove(0);
  }
  currentCheckPointPosition = c.stackPosition;
}
```

Since items are removed from the middle of the stack, the `stackPosition` member variable no longer has the correct value, and, as a result, `currentCheckPointPosition` is then also incorrect. This can result in a ClassCastException in the method `getCheckPoint`, since `undoStack.get( currentCheckPointPosition )` points to something undefined.

The quick solution is simply to add

```
c.stackPosition = undoStack.indexOf(c);
```

after the loop and before the assignment – making sure `stackPosition` has the correct value.

This is probably not a complete solution, but since only `forget( getCheckpoint() )` (clearing the entire stack except the last item) is called in the application, it is sufficient.

It would probably be better to remove the member variable `stackPosition` completely, replacing it with calls to `undoStack.indexOf()` throughout the code.

A.2. Modifications to the ArgoUML sources

Although one of the goals was not to modify any of the ArgoUML sources, in a few cases this simply can not be avoided. The following sections describe what modifications

are necesary. These modification have all been designed not to have any unwanted side-effects. ArgoUML will still behave as before when used with the original NSUML library. Hopefully these modifications will be incorporated into the main source trunk[1].

A.2.1. The plugin package

All classes implementing the plugin mechanism (section 9.1) are located in the package `org.argouml.util.plugin`. The following static methods should be added to the class `Main` (in package `org.argouml.application`):

`Main.getPlugins` should create a `PluginManager` instance and return an array of found plugins by calling `getPluginFiles().`

`Main.initPlugins` should instantiate and initialize each of the plugins. It should also query each plugin for any postload actions, adding these actions to the application's `postLoadActions` vector.

```
protected static Vector getPlugins() {
  PluginManager _pm = new PluginManager("ArgoUML-Plugin");
  return _pm.getPluginFiles();
}
protected static void initPlugins(Vector plugins) {
Iterator i = plugins.iterator();
while (i.hasNext()) {
  PluginSource ps = (PluginSource) i.next();
  try {
    Class c = ps.loadClass();
    ArgoPlugin plugin = (ArgoPlugin) c.newInstance();
    plugin.init();
    postLoadActions.addAll(plugin.getPostLoadActions());
  }
  catch (ClassNotFoundException e) {
    // handle error
  } catch (InstantiationException e) {
    // handle error
  } catch (IllegalAccessException e) {
    // handle error
  }
}
```

`Main.main` should then invoke these methods at the appropriate place:

```
public static void main(String args[]) {

...
// Initialize the module loader.
Argo.initializeModules();
// CCL Plugin Loader
initPlugins(getPlugins());
...
}
```

[1]In matter of speaking, the goal of this section is to make itself obsolete.

A.2.2. Accessing the Application Menu Bar

The main application window of ArgoUML is represented by the `ProjectBrowser` class. This class contains the application menu bar in the protected member variable `_menubar`.

Since the CCL plugin will need access to this member in order to add new menu items, the following method must be added to `ProjectBrowser`:

A.2.3. Extendable Property Panels

In the present version, a list of the different property panels (section 8.2) for the various metaclasses are hard-coded in the `TabProps` class (package `org.argouml.uml.ui`). This prevents plugins from adding their own property panels.

The following addition allows a new property panel to be registered with the property tab at run-time:

```
package org.argouml.uml.ui;
class TabProps
{
...
  public void addPanel(Class c, PropPanel p)
  {
    _panels.put(c,p);
  }
}
```

Since the property tab is contained by the details pane, adding the following method allows the plugin to add a new property panel through a well-known access point:

```
package org.argouml.ui;
class DetailsPane
{
...
  public void addToPropTab(Class c, PropPanel p)
  {
    for (int i = 0; i < _tabPanels.size(); i++)
    {
      if (_tabPanels.elementAt(i) instanceof TabProps)
      {
        ((TabProps)_tabPanels.elementAt(i)).addPanel(c,p);
      }
    }
  }
}
```

A.2.4. The extended NSUML library

In order to support the CCL metamodel extensions, the NSUML library has been modified to support a subset of the UML 1.4 specification. Unfortunalty, there are a few cases where UML 1.4 is not compatible with UML 1.3. Since ArgoUML is based on UML 1.3, a few classes must be changed to support the modified library.

Tagged Values and Tag Definitions

The concept of tagged values has been extended considerably in UML 1.4. Tagged values are now specified by a tag definition, which is linked to a stereotype, and can contain more than one value.

In order to still support UML1.3-compatible tagged values, the UML1.4 metamodel allows tag definitions which are not linked to a stereotype, but which are owned by the namespace of the model element[2].

In order to keep the NSUML object model backward-compatible, the UML1.3 mutators of `MTaggedValue` (`setTag`, `setValue`), have been modified to create UML 1.3-compatible tag definitions automatically, with one restriction: the tagged value must already be linked with a model element before calling these methods. Using these methods on a non-UML 1.3-compatible tagged value will result in an exception. The UML1.3 inspectors `getTag` and `getValue` have been modified to return the correct values as long as they are used on a UML 1.3-compatible tagged value. Using these methods on a non-UML 1.3 compatible tagged value will return `null` values.

The following methods in the ArgoUML sources must be modified to conform with the above restriction:

Method `setProperty` of `UMLTaggedBooleanProperty`: adding the tagged value to the element must be done before setting the tag and value:

```
if (!found) {
  MTaggedValue taggedValue = new MTaggedValue();
  element.addTaggedValue(taggedValue); // do first
  taggedValue.setTag(_tagName);
  if (newState) {
    taggedValue.setValue("true");
  }
  else {
    taggedValue.setValue("false");
  }
  // element.addTaggedValue(taggedValue);
}
```

In the method `setValueAt` of `TabTaggedValues`:

```
if (tvs.size() == rowIndex) {
  MTaggedValue tv = new MTaggedValueImpl()
  tv.setModelElement(_target); // do first
  if (columnIndex==0) tv.setTag((String)aValue);
  if (columnIndex==1) tv.setValue((String)aValue);
  tvs.addElement(tv);
  fireTableStructureChanged(); //?
  _tab.resizeColumns();
}
```

As long as only UML1.3 compatible tagged values are used, and the above restriction is met, the modified NSUML library will behave in a fully UML1.3 compatible manner.

[2] an example of this is tag 'legacy tag' in figure 10.3.

Stereotypes

The attribute `baseClass` has changed from String to String[*] in UML 1.4. The UML1.3 inspector `getBaseClass` will now return the *first* base class of the stereotype. The mutator `setBaseClass` will *remove* all current base classes and add the new base class. As long as the application uses base classes in a UML1.3 manner (each stereotype having only one base class), the library will behave in a fully UML1.3 compatible manner and require no changes to the application.

Bibliography

[Bübl2001-CCL] Bübl, Felix. *The Context Based Component Constraint Language*, TU Berlin, 2001

[Bübl2001-CoCons] Bübl, Felix. *Requirements Engineering via Context-Based Constraints*, TU-Berlin, 2001

[Große-Rhode+2000] Martin Große-Rhode, Ralf-Detlef Kutsche, Felix Bübl, *Concepts for the Evolution of Component-Based Software Systems*, Fachbereich Informatik, Technische Universität Berlin, 2000

[Gamma+95] Gamma, Helm, Johnson, Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[Beck2000] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesleym 2000

[Venners99] Bill Venners. *Design Techniques – Designing with Dynamic Extension*, JavaWorld January 1999 *(http://www.artima.com/designtechniques/index.html)*

[Fowler97] Martin Fowler. *UML Distilled – Applying the Standard Object Modeling Language*, Addison-Wesley, 1997

[Cheesman+2001] John Cheesman, John Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*, Addision-Wesley, 2001

[nsuml] The Novosoft UML API *(http://nsuml.sourceforge.net)*

[argouml] The ArgoUML Project *(http://argouml.tigris.org)*

[gef] The Java Graph Edting Framework GEF *(http://gef.tigris.org)*

[junit] The Java Unit Testing Framework JUnit (*http://www.junit.org*)

[Wake2000] William C. Wake, *The Test/Code Cycle in XP*, XPlorations, Febuary 2000 (*http://users.vnet.net/wwake/xp/xp0002/*)

[Java2API] Java 2 Platform, Standard Edition, v 1.3.1 API Specification (*http://java.sun.com/j2se/1.3/docs/api/index.html*)

[Swing] Creating a GUI with JFC/Swing, (*http://java.sun.com/docs/books/tutorial/uiswing/index.html*)

[UML1.3] OMG Unified Modeling Language Specification, Version 1.3 (*http://www.omg.org/technology/documents/formal/ unified_modeling_language.htm*)

[UML1.4draft] OMG Unified Modeling Language Specification, Version 1.4 draft February 2001 *(http://www.omg.org/technology/documents/recent/ omg_modeling.htm)*

[XMI1.1] OMG XML Metadata Interchange (XMI) Specification, Version 1.1