

**UNIVERSITÀ DEGLI STUDI DI PISA**

**Facoltà di Ingegneria**

**Corso di Laurea in Ingegneria Informatica**

Code generation starting from statecharts specified in  
UML

Tiziana Allegrini

a.a. 2001/2002



<b>1. INTRODUCTION.....</b>	<b>5</b>
<b>2. UML STANDARD .....</b>	<b>6</b>
2.1. UML.....	6
2.2. Modeling diagrams in UML .....	7
2.2.1 Use case .....	8
2.2.2 Class.....	12
2.2.3 Packages e Objects .....	16
2.2.4 Sequence .....	18
2.2.5 Collaboration.....	19
2.2.6 State .....	20
2.2.7 Activity .....	28
2.2.8 Component e deployment .....	31
<b>3. ARGOUML.....</b>	<b>33</b>
3.1. ArgoUML diagrams .....	35
3.2. Working with ArgoUML .....	37
3.2.1 ArgoUML Menu Bar .....	38
3.2.2 The Navigation Pane .....	38
3.2.3 ArgoUML Editing Pane.....	39
3.2.4 ArgoUML "To Do" Pane .....	40
3.2.5 ArgoUML Details Pane .....	40
3.2.6 Creating a class in ArgoUML .....	42
3.2.7 Creating statecharts in ArgoUML .....	43
<b>4. FROM STATECHART TO JAVA CODE.....</b>	<b>45</b>
4.1. Design Pattern and polimorfism .....	46
4.1.1 State Pattern .....	47
4.2. State pattern for code generation.....	52
<b>5. ARGOUML CODE ORGANIZATION.....</b>	<b>58</b>
5.1. General architecture .....	58
5.1.1 Packages.....	58
5.1.2 ArgoUML core .....	61
5.1.3 ArgoUML main window .....	63
5.1.4 Menu .....	65
5.1.5 Navigator pane .....	66
5.1.6 Editor .....	67
5.1.7 Details pane.....	70
<b>6. ARGOUML LIBRARIES.....</b>	<b>73</b>
6.1. GEF .....	73
6.1.1 Characteristics .....	73
6.1.2 Layers .....	74
6.1.3 Editor .....	75

6.1.4	Figs .....	76
6.1.5	Selections .....	76
6.1.6	Commands .....	78
6.1.7	Modes .....	78
6.2.	<i>NSUML</i> .....	79
6.2.1	UML Metamodel implementation .....	79
6.2.2	Packages structure .....	80
6.2.3	Foundation .....	80
6.2.4	Behavioural Elements .....	81
6.2.5	Model Management .....	82
6.2.6	API Description .....	83
6.2.7	Accessing and modifying metaattributes .....	86
6.2.8	Accessing and modifying metaassociations .....	87
6.2.9	NSUML reflective API .....	89
6.2.10	Other libraries .....	91
<b>7.</b>	<b>CODE GENERATION .....</b>	<b>92</b>
7.1.	<i>Implementation</i> .....	92
7.1.1	Classe Generator .....	92
7.1.2	Classe GeneratorJava .....	93
7.1.3	GeneratorDisplay .....	93
7.1.4	GeneratorState .....	93
7.1.5	Class properties .....	96
7.1.6	State machine properties .....	97
7.2.	<i>Generated code rendering</i> .....	100
7.2.1	Classe TabModelTarget .....	101
7.2.2	Classe TabText .....	102
7.2.3	Classe TabSrcState .....	102
7.3.	<i>Show generate code in "Source state" tab strip</i> .....	104
7.4.	<i>New menu: Generation statecharts</i> .....	106
7.4.1	Class ProjectBrowser .....	106
7.4.2	Class UMLAction .....	108
7.4.3	Class Actions .....	108
7.4.4	Class ActionGenStateFile .....	109
7.4.5	Class ClassStateGenerationDialog .....	110
7.4.6	Show the call stack from the menu .....	111
<b>8.</b>	<b>CONCLUSIONS .....</b>	<b>115</b>
<b>9.</b>	<b>THANKS .....</b>	<b>116</b>

# 1. Introduction

Far from 1980 many reserch groups developed some methodologies and notations for OO analisys and design. This situation needed a standard language and notation.

About 1994, when Jacobson arrived to the Rational society, within Booch and Rambaugh, they developed a unified language for design software, UML.

Many society were interested in UML and they built their project on UML notation, also developed their own CASE tool.

OMG in 1998, made the standard of UML in his first version.

All CASE tools had to support the whole design process of software development and they required some automatic function of code generation.

ArgoUML manage the automatic code generation based only on class diagrams.

§In this work we make ArgoUML be able to generating code from class diagrams and statecharts. Code generated restricts and controls behaviuor of the object to which the statechart is associated. We based this code generation on the pattern State architecture, such a riusable and modular software pattern.

## 2. UML standard

### 2.1. UML

During the 1980's a number of OOA&D process methodologies and notations were developed by different research teams. It became clear there were many common themes and, during the 1990's, a unified approach for OOA&D notation was developed under the auspices of the Object Management Group. This standard became known as the Unified Modeling Language (**UML**), and is now the standard language for communicating OO concepts.

The heart of object-oriented problem solving is the construction of a model. The model abstracts the essential details of the underlying problem from its usually complicated real world. Several modeling tools are wrapped under the heading of the **UML**, which stands for Unified Modeling Language.

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems.

The UML is applicable to object-oriented problem solving. Anyone interested in learning UML must be familiar with the underlying tenet of object-oriented problem solving -- it all begins with the construction of a model. A **model** is an abstraction of the underlying problem. The **domain** is the actual world from which the problem comes.

Models consist of **objects** that interact by sending each other messages. Think of an object as "alive." Objects have things they know (attributes) and things they can do (behaviors or operations). The values of an object's attributes determine its state.

**Classes** are the "blueprints" for objects. A class wraps attributes (data) and behaviors (methods or functions) into a single distinct entity. Objects are instances of classes.

At the center of the UML are its eight different kinds of modeling diagrams, which we describe here.

## 2.2. Modeling diagrams in UML

In terms of the views of a model, the UML defines the following graphical diagrams:

- use case diagram
- class diagram
- behavior diagrams:
  - statechart diagram
  - activity diagram
- interaction diagrams:
  - sequence diagram
  - collaboration diagram
- implementation diagrams:

- component diagram
- deployment diagram

Although other names are sometimes given to these diagrams, this list constitutes the canonical diagram names.

These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a selfconsistent system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees, although the UML and supporting tools will provide for a number of derivative views.

### **2.2.1 Use case**

Use case diagrams show actors and use cases together with their relationships. The use cases represent functionality of a system or a classifier, like a subsystem or a class, as manifested to external interactors with the system or the classifier.

Use case diagrams describe what a system does from the standpoint of an external bserver. The emphasis is on what a system does rather than how.

A use case diagram is a graph of actors, a set of use cases, possibly some interfaces, and the relationships between these elements. The relationships are associations between the actors and the use cases, generalizations between the actors, and generalizations, extends, and includes among the use cases. The use cases may optionally be enclosed by a rectangle that represents the boundary of the containing system or classifier.

A *use case* is a kind of classifier representing a coherent unit of functionality provided by a system, a subsystem, or a class as manifested by sequences of messages

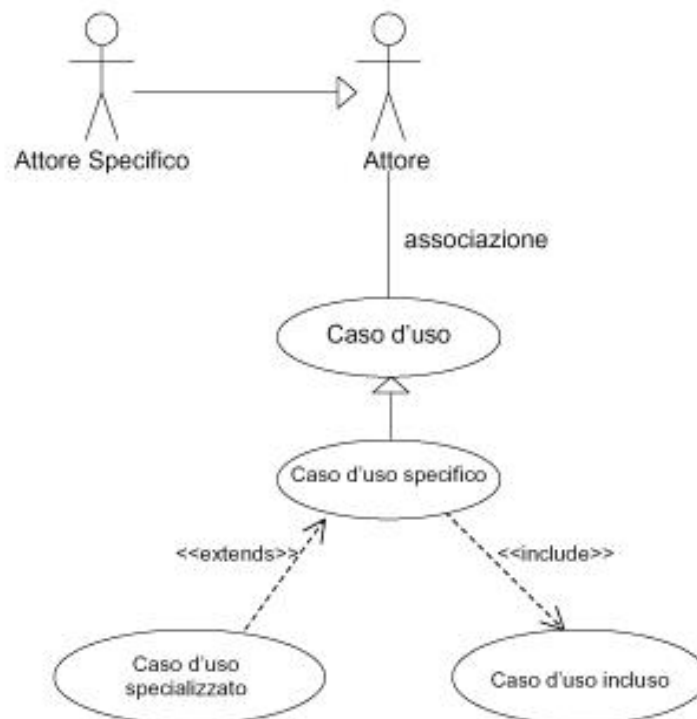


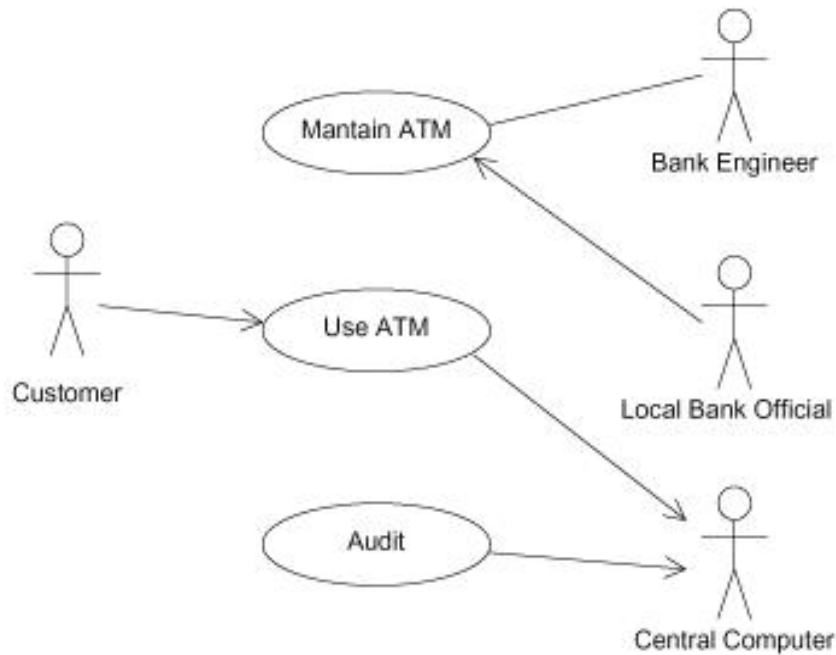
exchanged among the system (subsystem, class) and one or more outside interactors (called *actors*) together with actions performed by the system (subsystem, class).

An *extension point* is a reference to one location within a use case at which action sequences from other use cases may be inserted. Each extension point has a unique name within a use case, and a description of the location within the behavior of the use case.

An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates.

The standard stereotype icon for an actor is a “stick man” figure with the name of the actor below the figure.

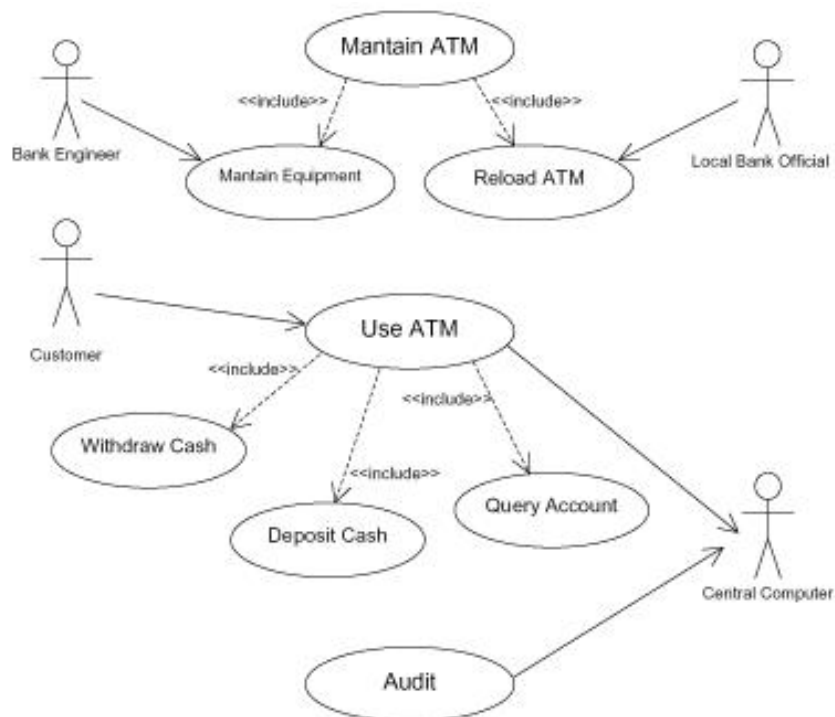
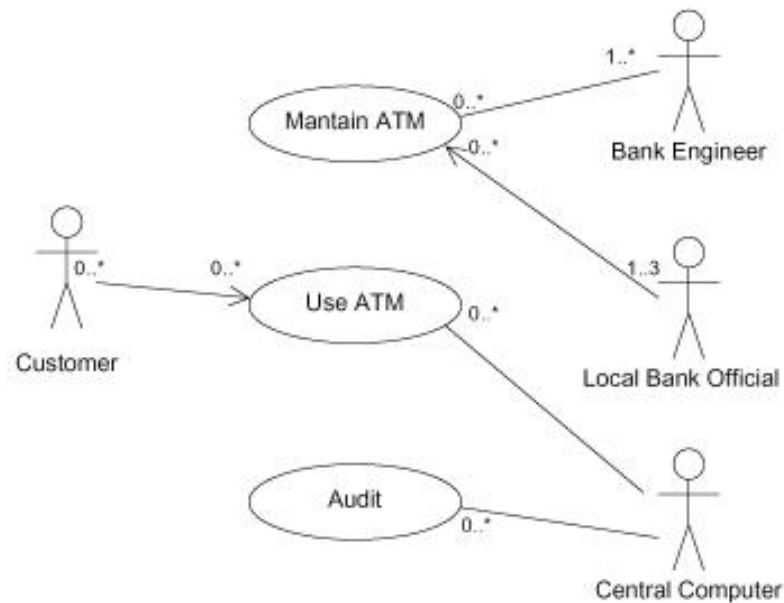




There are several standard relationships among use cases or between actors and use cases.

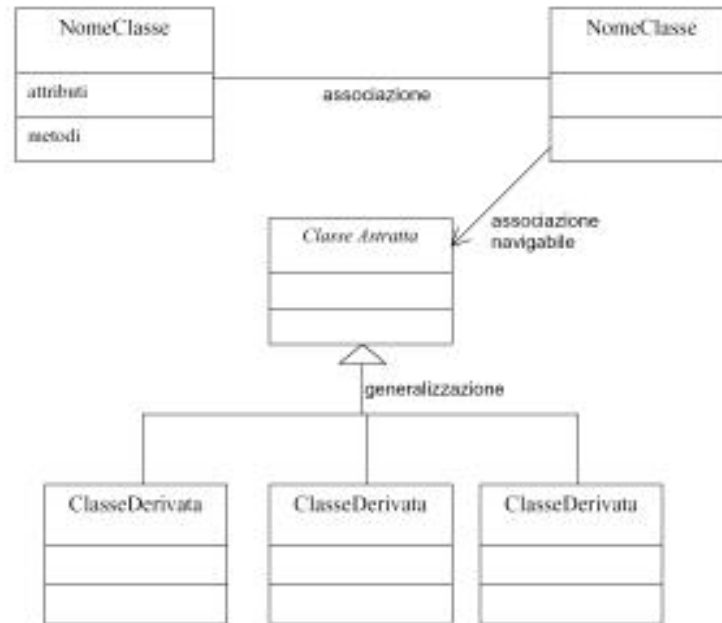
- Association – The participation of an actor in a use case; that is, instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases.
- Extend – An extend relationship from use case A to use case B indicates that an instance of use case B may be augmented (subject to specific conditions specified in the extension) by the behavior specified by A. The behavior is inserted at the location defined by the extension point in B, which is referenced by the extend relationship.
- Generalization – A generalization from use case C to use case D indicates that C is a specialization of D.

- Include – An include relationship from use case E to use case F indicates that an instance of the use case E will also contain the behavior as specified by F. The behavior is included at the location which defined in E.



## 2.2.2 Class

A Class diagram gives an overview of a system by showing its classes and the relationships among them. Class diagrams are static -- they display what interacts but not what happens when they do interact.



UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes, such as *Payment*, are in italics. Relationships between classes are the connecting links.

Our class diagram has three kinds of relationships:

**association** -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.

**aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, Order has a collection of OrderDetails.

**generalization** -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. Payment is a superclass of Cash, Check, and Credit.

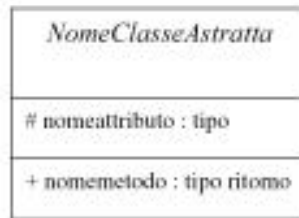
An association has two ends. An end may have a role name to clarify the nature of the association. For example, an OrderDetail is a line item of each Order.

A navigability arrow on an association shows which direction the association can be traversed or queried. The arrow also lets you know who "owns" the association's implementation; in this case, OrderDetail has an Item. Associations with no navigability arrows are bi-directional.

The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers. In our example, there can be only one Customer for each Order, but a Customer can have any number of Orders.

This table gives the most common multiplicities.

Molteplicità	Significato
0..1	Zero o una istanza
0..*    o    *	Nessun limite al numero di istanze
1	Esattamente una
1..*	Almeno una



Strings in the attribute compartment are used to show attributes in classes. A similar syntax is used to specify qualifiers, template parameters, operation parameters, and so on (some of these omit certain terms).

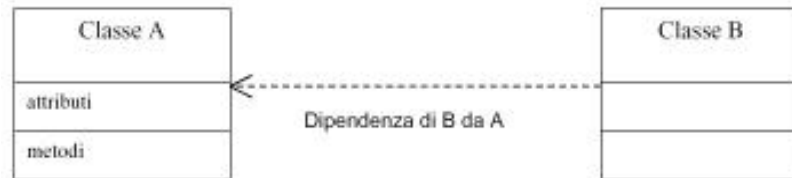
An attribute is shown as a text string that can be parsed into the various properties of an attribute model element. An operation is a service that an instance of the class may be requested to perform. It has a name and a list of arguments. An operation is shown as a text string that can be parsed into the various properties of an operation model element.

**Association** - Binary associations are shown as lines connecting two classifier symbols. The lines may have a variety of adornments to show their properties. Ternary and higher-order associations are shown as diamonds connected to class symbols by lines. A binary association is an association among exactly two classifiers (including the possibility of an association from a classifier to itself).



**Dependency** – A dependency indicates a semantic relationship between two model elements (or two sets of model elements). It relates the model elements themselves and does not require a set of instances for its meaning. It indicates a situation in which a

change to the target element may require a change to the source element in the dependency. A dependency is shown as a dashed arrow between two model elements. The model element at the tail of the arrow (the client) depends on the model element at the arrowhead (the supplier). The arrow may be labeled with an optional stereotype and an optional individual name.



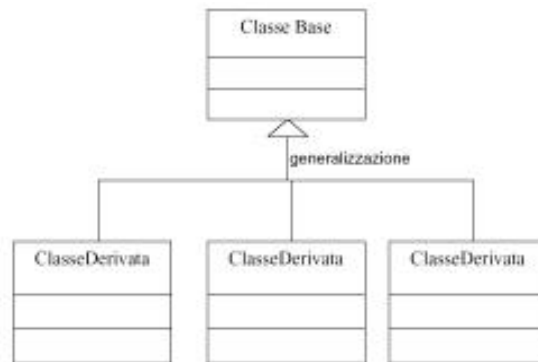
**Composition** - Composite aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. The multiplicity of the aggregate end may not exceed one (it is unshared). See Section 3.43, “Association End,” on page 3-71 for further details.

Composition may be shown by a solid filled diamond as an association end adornment. Alternately, UML provides a graphically-nested form that is more convenient for showing composition in many cases.



**Generalization** - Generalization is the taxonomic relationship between a more general element (the parent) and a more specific element (the child) that is fully consistent with the first element and that adds additional information. It is used for classes, packages, use cases, and other elements.

Generalization is shown as a solid-line path from the child (the more specific element, such as a subclass) to the parent (the more general element, such as a superclass), with a large hollow triangle at the end of the path where it meets the more general element.



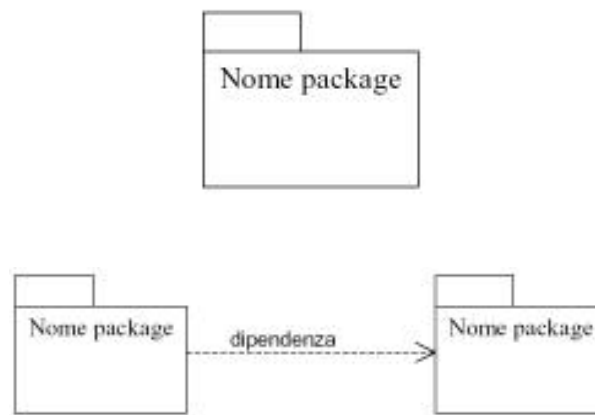
### 2.2.3 Packages e Objects

A package is a collection of logically related UML elements. The diagram below is a business model in which the classes are grouped into packages.



Packages appear as rectangles with small tabs at the top. The package name is on the tab or inside the rectangle. The dotted arrows are dependencies. One package depends on another if changes in the other could possibly force changes in the first.

Object diagrams are special kinds of class diagrams, showing instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships.



The object diagram below instantiates the class diagram, replacing it by a concrete example.

Each rectangle in the diagram corresponds to a single instance. Instance names are underlined in UML diagrams. Class or instance names may be omitted from object diagrams as long as the diagram meaning is still clear.

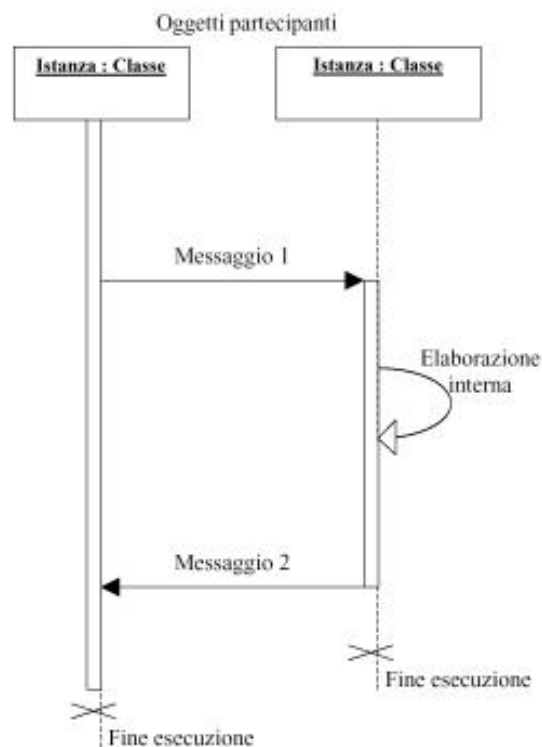
An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. The use of object diagrams is fairly limited, mainly to show examples of data structures.



## 2.2.4 Sequence

Class and object diagrams are static model views. Interaction diagrams are dynamic. They describe how objects collaborate.

A sequence diagram is an interaction diagram that details how operations are carried out -- what messages are sent and when. Sequence diagrams are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence. Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window.



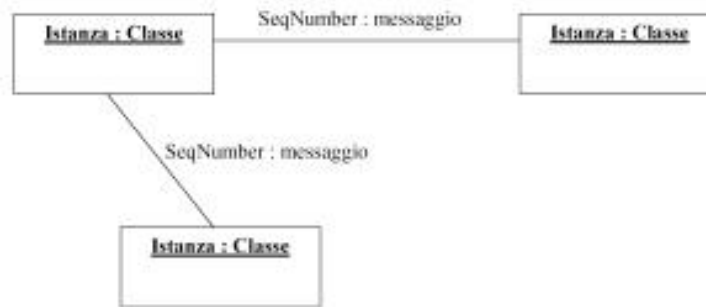
Each vertical dotted line is a lifeline, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the activation bar of the message on the receiver's lifeline. The activation bar represents the duration of execution of the message.

A sequence diagram presents an Interaction, which is a set of Messages between ClassifierRoles within a Collaboration, or an InteractionInstanceSet, which is a set of Stimuli between Instances within a CollaborationInstanceSet to effect a desired operation or result.

### **2.2.5 Collaboration**

A collaboration diagram presents either a Collaboration, which contains a set of roles to be played by Instances, as well as their required relationships given in a particular context, or it presents a CollaborationInstanceSet with a collection of Instances and their relationships. The diagram may also present an Interaction (InteractionInstanceSet), which defines a set of Messages (Stimuli) specifying the interaction between the Instances playing the roles within a Collaboration to achieve the desired result.

Collaboration diagrams are also interaction diagrams. They convey the same information as sequence diagrams, but they focus on object roles instead of the times that messages are sent. In a sequence diagram, object roles are the vertices and messages are the connecting links.



The object-role rectangles are labeled with either class or object names (or both). Class names are preceded by colons ( : ).

Each message in a collaboration diagram has a sequence number. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

## 2.2.6 State

Objects have behaviors and state. The state of an object depends on its current activity or condition. A statechart diagram shows the possible states of the object and the transitions that cause a change in state.

Statechart diagrams represent the behavior of entities capable of dynamic behavior by specifying its response to the receipt of event instances. Typically, it is used for describing the behavior of class instances, but statecharts may also describe the behavior of other entities such as use-cases, actors, subsystems, operations, or methods.

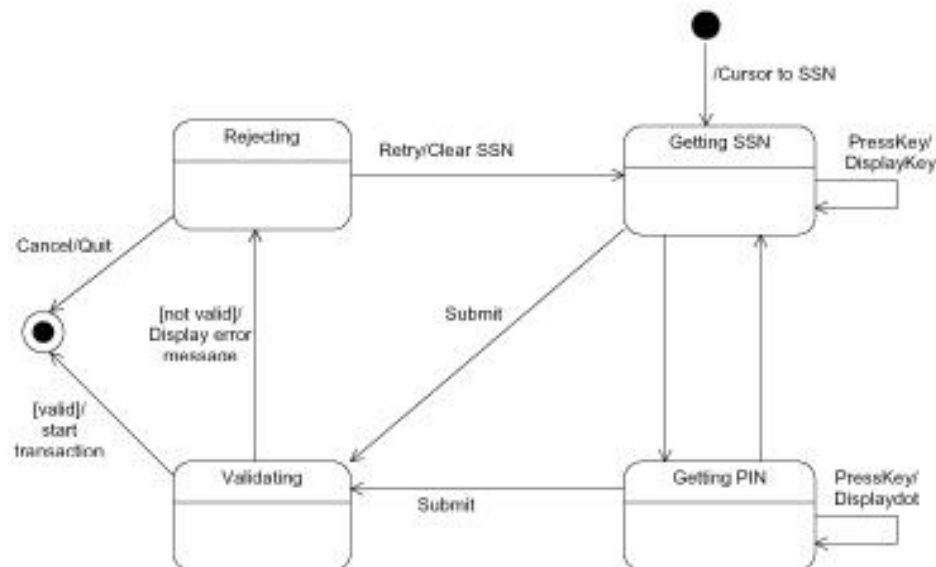
A statechart diagram is a graph that represents a state machine. States and various other types of vertices (pseudostates) in the state machine graph are rendered by appropriate state and pseudostate symbols, while transitions are generally rendered by directed arcs that inter-connect them. States may also contain subdiagrams by physical containment or tiling. Note that every state machine has a top state that contains all the other

elements of the entire state machine. The graphical rendering of this top state is optional.

A statechart diagram maps into a StateMachine. That StateMachine may be owned by an instance of a model element capable of dynamic behavior, such as classifier or a behavioral feature, which provides the context for that state machine. Different contexts may apply different semantic constraints on the state machine.

Our example diagram models the login part of an online banking system. Logging in consists of entering a valid social security number and personal id number, then submitting the information for validation.

Logging in can be factored into four non-overlapping states: Getting SSN, Getting PIN, Validating, and Rejecting. From each state comes a complete set of transitions that determine the subsequent state.



States are rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows. Our diagram has two self-transition, one on Getting SSN and another on Getting PIN.

The initial state (black circle) is a dummy to start the action. Final states are also dummy states that terminate the action.

The action that occurs as a result of an event or condition is expressed in the form /action. While in its Validating state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state.

## State

A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event. A composite state is a state that, in contrast to a simple state, has a graphical decomposition. Conceptually, an object remains in a state for an interval of time. However, the semantics allow for modeling “flow-through” states that are instantaneous, as well as transitions that are not instantaneous.



A state is shown as a rectangle with rounded corners. Optionally, it may have an attached name tab. The name tab is a rectangle, usually resting on the outside of the top

side of a state and it contains the name of that state. It is normally used to keep the name of a composite state that has concurrent regions, but may be used in other cases as well .

A state may be optionally subdivided into multiple compartments separated from each other by a horizontal line. They are as follows:

### ***Name compartment***

This compartment holds the (optional) name of the state as a string. States without names are anonymous and are all distinct. It is undesirable to show the same named state twice in the same diagram, as confusion may ensue. Name compartments should not be used if a name tab is used and vice versa.

### ***Internal transitions compartment***

This compartment holds a list of internal actions or activities that are performed while the element is in the state. The action label identifies the circumstances under which the action specified by the action expression will be invoked. The action expression may use any attributes and links that are in the scope of the owning entity. For list items where the action expression is empty, the backslash separator is optional.

### ***Initial State***

The initial state is a pseudostate representing the default state of a state machine (or composite state) when it is created. It is the state from which any initial transition is made. As a consequence it is not permissible to have incoming transitions. An initial state is represented on the diagram as a solid disc.

### ***Final State***

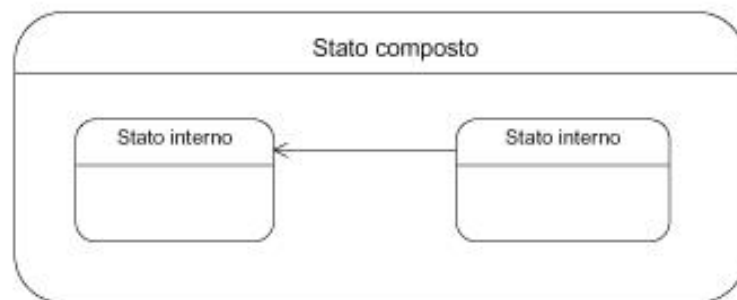
If a transition reaches a final state, it implies completion of the activity associated with that composite state, or at the top level, of the complete state machine. In the UML metamodel `FinalState` is a child of `State`. Completion at the top level implies termination

(i.e. destruction) of the owning object instance. The representation of a final state on the diagram is a circle with a small disc at its center.

### ***Composite state***

A composite state is decomposed into two or more concurrent substates (called regions) or into mutually exclusive disjoint substates. A given state may only be refined in one of these two ways. Naturally, any substate of a composite state can also be a composite state of either type.

A composite state is a state that contains other states (known as sub-states), allowing hierarchical state machines to be constructed. Sub-states are placed within a composite machine by placing them entirely within the composite state when creating them for the first time in the editing pane.



### ***Transition***

A transition is a directed relation between a source state (or composite state) and destination state (or composite state). Within the UML metamodel, Transition is a subclass of `ModelElement`.

A simple transition is a relationship between two states indicating that an instance in the first state will enter the second state and perform specific actions when a specified event occurs provided that certain specified conditions are satisfied. On such a change of



state, the transition is said to “fire.” The trigger for a transition is the occurrence of the event labeling the transition. The event may have parameters, which are accessible by the actions specified on the transition as well as in the corresponding exit and entry actions associated with the source and target states respectively. Events are processed one at a time. If an event does not trigger any transition, it is discarded. If it can trigger more than one transition within the same sequential region; that is, not in different concurrent regions, only one will fire. If these conflicting transitions are of the same priority, an arbitrary one is selected and triggered.

A transition is shown as a solid line originating from the *source* state and terminated by an arrow on the *target* state. It may be labeled by a *transition string* that has the following general format:

*event* [ *guard* ] / *action* ^ *signal*

The *event-signature* describes an event with its arguments:

*event-name* ‘(’ *comma-separated-parameter-list* ‘)’

The *guard-condition* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the object that owns the state machine. The guard condition may also involve tests of concurrent states of the current machine, or explicitly designated states of some reachable object (for example, “**in** State1” or “**not in** State2”). State names may be fully qualified by the nested states that contain them, yielding pathnames of the form “State1::State2::State3.” This may be used in case same state name occurs in different composite state regions of the overall machine.

The *action-expression* is executed if and when the transition fires. It may be written in terms of operations, attributes, and links of the owning object and the parameters of the

triggering event, or any other features visible in its scope. The corresponding action must be executed entirely before any other actions are considered. This model of execution is referred to as *run-to-completion* semantics. The action expression may be an action sequence comprising a number of distinct actions including actions that explicitly generate events, such as sending signals or invoking operations. The details of this expression are dependent on the action language chosen for the model.

## **Event**

An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition. Events may be of several kinds (not necessarily mutually exclusive).

An event is an observable occurrence. In the UML metamodel it is a child of `ModelElement`. There are a number of different types of event that are children of event within the UML metamodel.

`SignalEvent`. Associated with a signal, this event is caused by the signal being raised.

`CallEvent`. Associated with an operation of a class, this event is caused by a call to the given operation. The expected effect is that the steps of the operation will be executed.

`TimeEvent`. An event caused by expiration of a timing deadline.

`ChangeEvent`. An event caused by a particular expression (of attributes and associations) becoming true.

An action is represented by its name.

A designated condition becoming true (described by a Boolean expression) results in a change event instance. The event occurs whenever the value of the expression changes from false to true. Note that this is different from a guard condition. A guard condition is evaluated *once* whenever its event fires. If it is false, then the transition does not occur and the event is lost.

The receipt of an explicit signal from one object to another results in a signal event instance. It is denoted by the signature of the event as a trigger on a transition.

The receipt of a call for an operation implemented as a transition by an object represents a call event instance.

The passage of a designated period of time after a designated event (often the entry of the current state) or the occurrence of a given date/time is a TimeEvent.

The event declaration has scope within the package it appears in and may be used in state diagrams for classes that have visibility inside the package. An event is *not* local to a single class.

### ***Guard***

A guard is associated with a transition. At the time an event is dispatched, the guard is evaluated, and if false, its transition is disabled. In the UML metamodel, Guard is a child of ModelElement.

### ***Action***

An action specifies an activity to do and it is the abstraction of a computational procedure: it may change the state of an object. The action is “atomic” and it may be executed only due to a transition.

There are several types of actions: `CreateAction` or `DestroyAction`, that respectively creates and destroys an object, and `CallAction`, `SendAction` and `ReturnAction` that calls, send and return some parameters.

### **2.2.7 Activity**

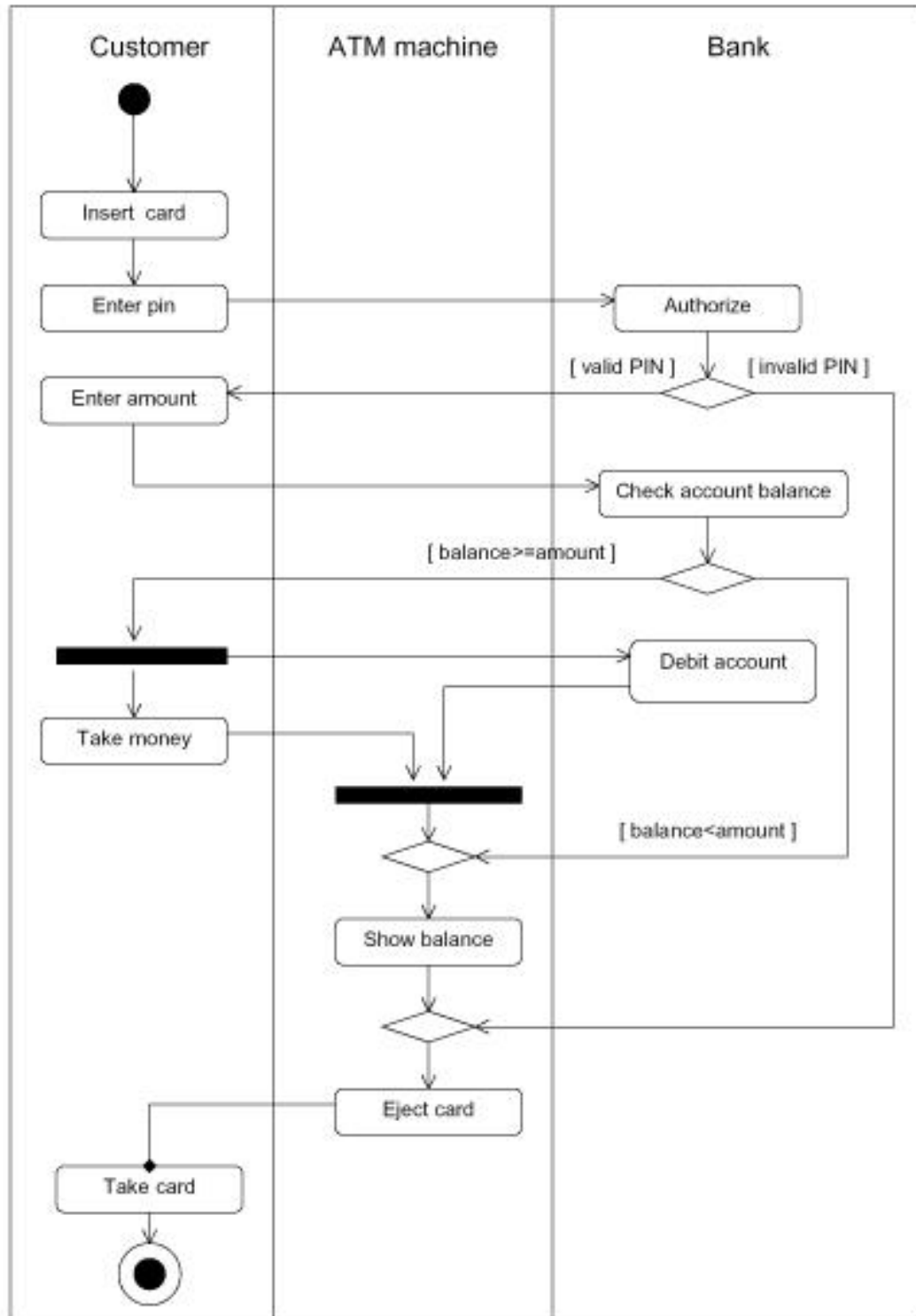
An activity diagram is essentially a fancy flowchart. Activity diagrams and statechart diagrams are related. While a statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an activity diagram focuses on the flow of activities involved in a single process. The activity diagram shows the how those activities depend on one another.

For our example, we used the following process.

"Withdraw money from a bank account through an ATM."

The three involved classes (people, etc.) of the activity are `Customer`, `ATM`, and `Bank`. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are rounded rectangles.

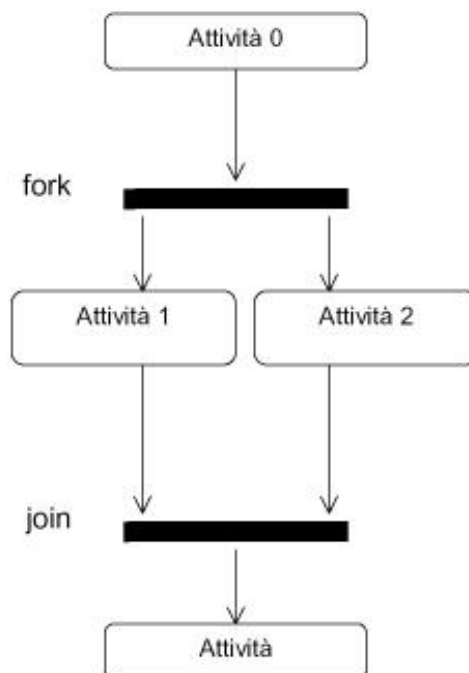
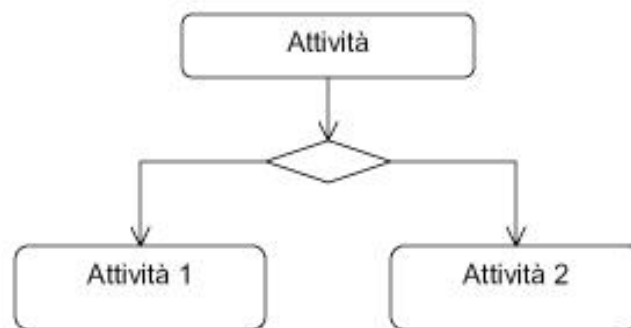
An activity graph is a variation of a state machine in which the states represent the performance of actions or subactivities and the transitions are triggered by the completion of the actions or subactivities. It represents a state machine of a procedure itself.



Activity diagrams can be divided into object swimlanes that determine which object is responsible for which activity. A single transition comes out of each activity, connecting it to the next activity.

A transition may branch into two or more mutually exclusive transitions. Guard expressions (inside [ ]) label the transitions coming out of a branch. A branch and its subsequent merge marking the end of the branch appear in the diagram as hollow diamonds.

A transition may fork into two or more parallel activities. The fork and the subsequent join of the threads coming out of the fork appear in the diagram as solid bars.



## 2.2.8 Component e deployment

A component is a code module. Component diagrams are physical analogs of class diagram. Deployment diagrams show the physical configurations of software and hardware.

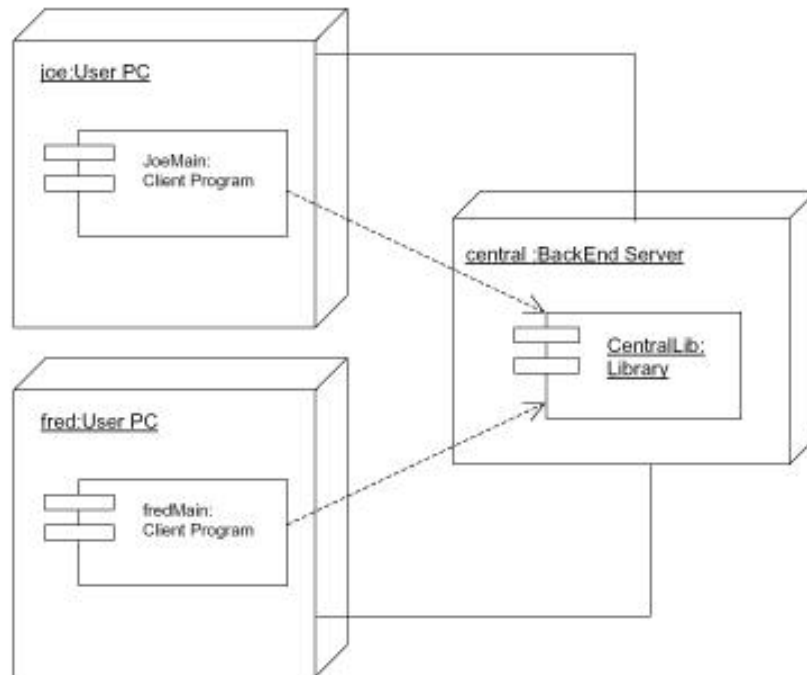
The following deployment diagram shows the relationships among software and hardware components involved in real estate transactions.

A component diagram shows the dependencies among software components, including the classifiers that specify them (for example, implementation classes) and the artifacts that implement them; such as, source code files, binary code files, executable files, scripts.

A component diagram has only a type form, not an instance form. To show component instances, use a deployment diagram (possibly a degenerate one without nodes). A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships.



The physical hardware is made up of nodes. Each component belongs on a node. Components are shown as rectangles with two tabs at the upper left.



Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that execute on them. Software component instances represent run-time manifestations of software code units. Components that do not exist as run-time entities (because they have been compiled away) do not appear on these diagrams, they should be shown on component diagrams.

A deployment diagram is a graph of nodes connected by communication associations. Nodes may contain component instances. This indicates that the component runs or executes on the node. Components may contain instances of classifiers, which indicates that the instance resides on the component. Components are connected to other components by dashed-arrow dependencies (possibly through interfaces). This indicates that one component uses the services of another component. A stereotype may be used to indicate the precise dependency, if needed.



### 3. ArgoUML

ArgoUML was conceived as a tool and environment for use in the analysis and design of object-oriented software systems. In this sense it is similar to many of the commercial CASE tools that are sold as tools for modeling software systems. ArgoUML has a number of very important distinctions from many of these tools.

ArgoUML is a powerful yet easy-to-use interactive, graphical software design environment that supports the design, development and documentation of object-oriented software applications. ArgoUML takes part to the family of software applications called Computer Aided Software Engineering (CASE) tools.

ArgoUML is based directly on the UML 1.3 specification. In fact, a large part of ArgoUML was generated automatically from the UML specification. ArgoUML is (to the best of our knowledge) the only tool that implements the UML meta-model exactly as specified. In contrast, current commercial tools use tools use basically the same internal representation of the design that they used in previous versions.

The users of ArgoUML are software designers & architects, software developers, business analysts, systems analysts and other professionals involved in the analysis, design and development of software applications.

Main features:

**Open standards:** XMI, SVG and PGML - ArgoUML supports open standards extensively - UML, XMI, SVG, OCL and others.

ArgoUML is compliant with the **OMG Standard** for UML in its latest version 1.3. The code for the internal representation of an UML model is completely

generated from the specification and, thus, follows it very closely. To achieve this, a special metamodel library (NSUML) was developed by Novosofts and made available under LGPL.

**XML Metadata Interchange (XMI)** is the standard for saving the meta-data that make up a particular UML model. In principle this will allow you to take the model you have created in ArgoUML and import it into another tool.

**UML design editing** - ArgoUML uses GEF, the UCI Graph Editing Framework to edit UML diagrams. The following diagram types are supported:

Class diagrams

State machine diagrams

Activity diagrams

Use case diagrams

Collaboration diagrams

Object/Component/Deployment diagrams

Sequence diagrams

**Object Constraint Language (OCL)** is the UML standard for expressing constraints within diagrams that express the dynamic behavior of a design. At present OCL is quite new and not widely available. ArgoUML is one of the few CASE tools to provide comprehensive support.

**Model information** for static structure and use case diagrams can now be stored to a mySQL database.

Diagrams can now be exported to gif, PostScript, eps, PGML and svg. The standard saving format for diagrams is still PGML, but it will be changed to the upcoming standard for Scalable Vector Graphics (svg) of the W3C consortium.

**100% Java** - ArgoUML is a 100% pure Java application. This allows ArgoUML to run on all platforms for which a reliable port of the Java2 platform is available. Java was conceived as an interpreted language. It doesn't have a compiler to produce code for any particular target machine. It compiles code for its own target, the Java Virtual Machine (JVM).

**Open Source** allows to extend or customize it - ArgoUML is an open source project. The availability of the source ensures that a new generation of software designers and researchers now have a proven framework from which they can drive the development and evolution of CASE tool technologies.

Cognitive features like: reflection-in-action, opportunistic design, comprehension and problem solving - The implementation of these theories within ArgoUML is through a number of techniques.

## 3.1. ArgoUML diagrams

The UML standard specifies eight principle diagrams, all of which are supported by ArgoUML.

**Use case diagram.** Used to capture and analyse the requirements for any OOA&D project.

**Class diagram.** This diagram captures the static structure of the system being designed, showing the classes, interfaces and datatypes and how they are related. Variants of this

diagram are used to show package structures within a system (the package diagram) and the relationships between particular instances (the object diagram). The ArgoUML class diagram provides support for class and package diagrams.

**Behavior diagrams.** There are four such diagrams (or strictly speaking, five, since the use case diagram is a type of behavior diagram), which show the dynamic behavior of the system at all levels.

**State diagram.** Used to show the dynamic behavior of a single object (class instance). This diagram is of particular use in systems using complex communication protocols, such as in telecommunications.

**Activity diagram.** Used to show the dynamic behavior of groups of objects (class instance). This diagram is an alternative to the state diagram, and is better suited to systems with a great deal of user interaction.

**Interaction diagrams.** There are two diagrams in this category, used to show the dynamic interaction between objects (class instances) in the system.

**Sequence diagram.** Shows the interactions (typically messages or procedure calls) between instances of classes (objects) and actors against a timeline. Particularly useful where the timing relationships between interactions are important.

**Collaboration diagram.** Shows the interactions (typically messages or procedure calls) between instances of classes (objects) and actors against the structural relationships between those instances. Particularly useful where it is useful to relate interactions to the static structure of the system.

**Implementation diagrams.** UML defines two implementation diagrams to show the relationship between the software components that make up a system (the component diagram) and the relationship between the software and the hardware on which it is deployed at run-time (the deployment diagram).

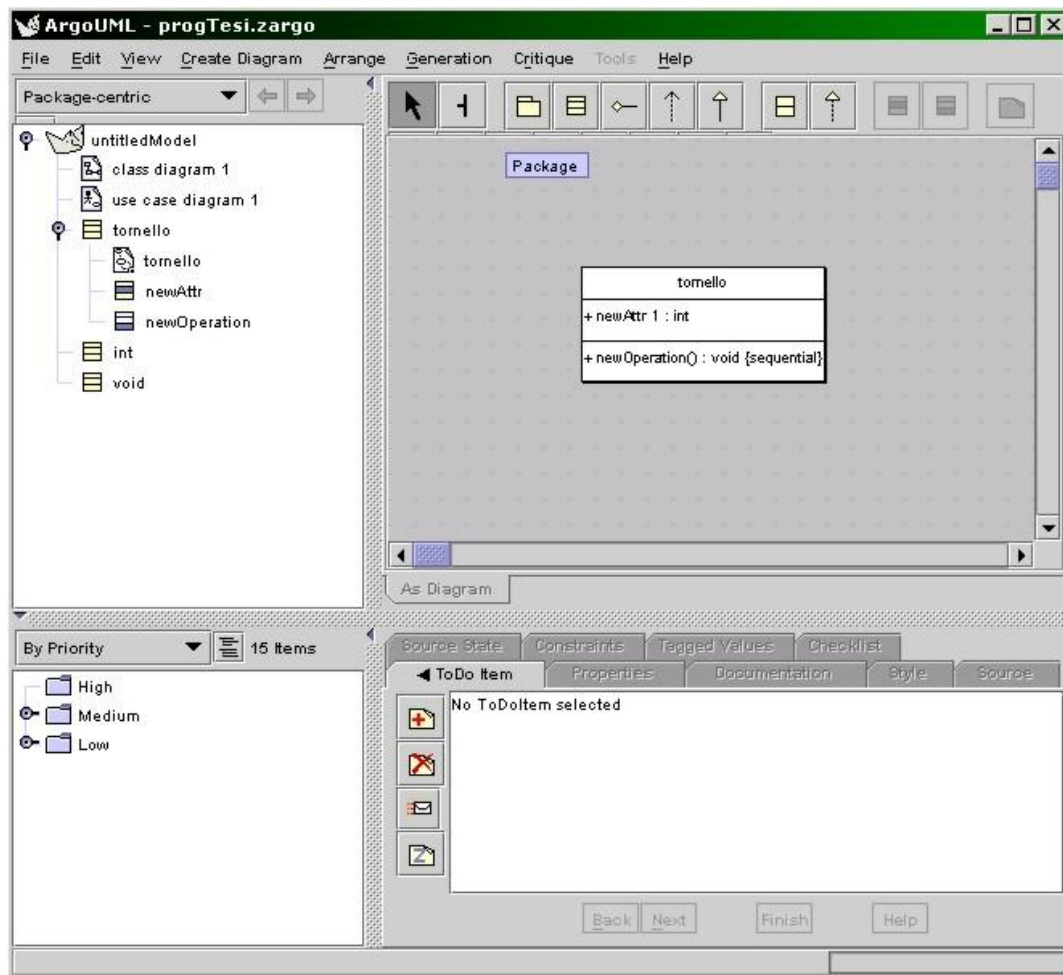
The ArgoUML deployment diagram provides support for both component and deployment diagrams.

## **3.2. Working with ArgoUML**

At the top there is a menu bar with commands available. In the file menu you can store the model or open another model instead. The bulk of the window comprises four sub-windows or Panes.

The upper left part of ArgoUML, Navigator Pane, lists the diagrams and objects of the model in one of several views. The upper right part of ArgoUML, Editing Pane, shows one diagram at the time. You can work with the objects in the diagrams, dragging and dropping and using the quick-links to create new objects connected to the already present objects.

The lower right part, Details Pane, you contains various details of various objects of the model. You select the object in one of the upper levels and choose what details you want to examine using the tabs. The lower left part, To-Do Pane, contains a list of all ToDo items for this model.



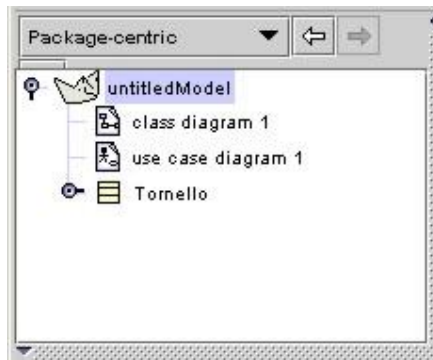
### 3.2.1ArgoUML Menu Bar

Argo's menu bar consists of menus for File, Edit, View, Arrange, Create, Generate, Critique, and Help.



### 3.2.2The Navigation Pane

Argo's Navigation Pane shows you the contents of your design. This element of Argo's UI should be familiar to anyone who has used the Microsoft Windows Explorer or any one of a number of commercial CASE tools.



The navigation pane allows the user to view the structure of the model from a number of predefined perspectives. It also allows the user to define their own perspectives for custom navigation of the model.

### 3.2.3 ArgoUML Editing Pane

Argo's Editing Pane is the main work area. You use this pane mainly to edit diagrams. However you can also use this pane to edit tables that list the contents of diagrams or other design objects. Several tabs are located at the bottom of this pane to indicate the different ways in which the main object can be viewed and edited. ArgoUML includes an "As Diagram" tab, and if you download optional jar files you may also see tabs labeled "As Table" and "As Metrics".

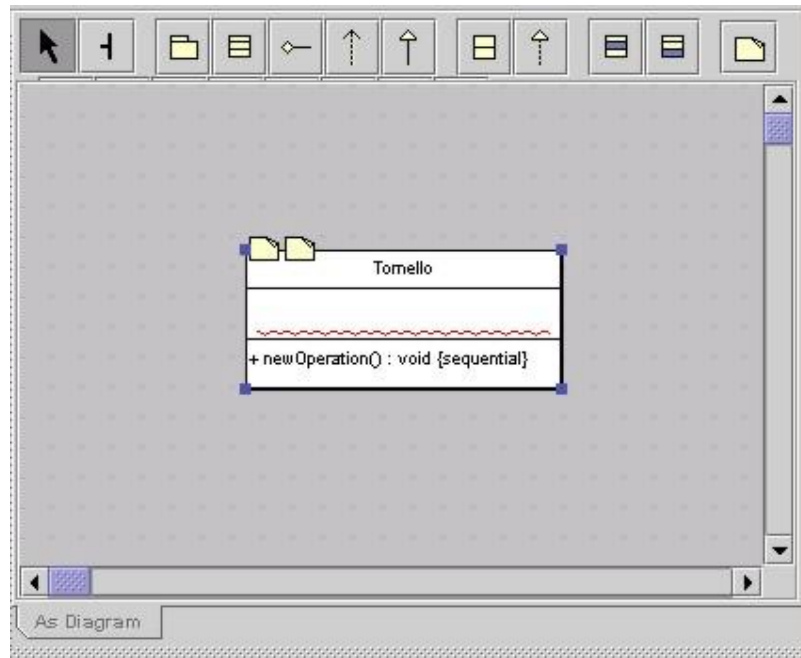
The toolbar at the top of the editing pane provides the main functions of the pane. The default tool is the Select tool. The tools fall into four categories.

Layout tools. Provide assistance in laying out artifacts on the diagram.

Annotation tools. Used to annotate artifacts on the diagram.

Drawing tools. Used to add general graphics artifacts to diagrams.

Diagram specific tools. Used to add UML artifacts specific to a particular diagram type to the diagram.



### 3.2.4ArgoUML "To Do" Pane

Argo's "To Do" Pane helps keep designers on track by reminding them of what needs to be done. Items in the To Do pane can be personal reminders entered by the designer, but most of them are generated by design critics. Design critics in ArgoUML continuously analyze the design looking for incomplete or problematic areas.

### 3.2.5ArgoUML Details Pane

Argo's Details Pane allows you to edit the details of the currently selected design element or "to do" item.

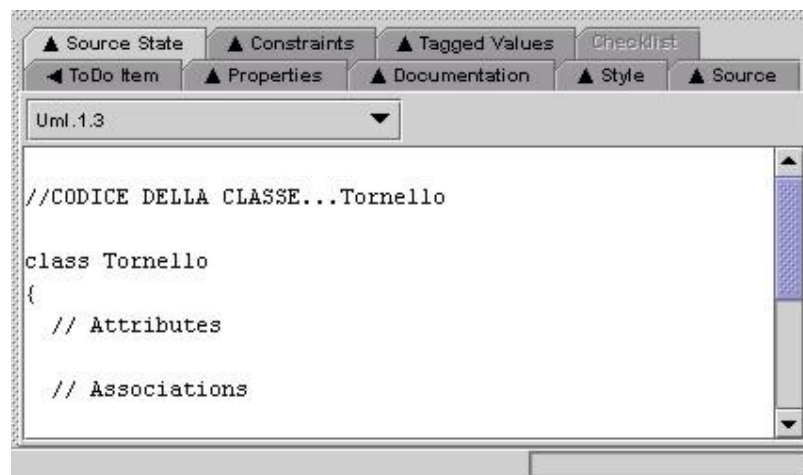
- *ToDoItem*– Argo's *ToDoItem* Tab shows the description of the selected "to do" item in the "To Do" Pane. The description consists three short paragraphs about the problem, why this problem is likely to be important to you, and steps that



you can take to resolve the problem. The `ToDoItem` tab will then function as a wizard, leading you through the steps of resolving the problem.

- *Properties*– Argo's Properties Tab shows the properties of the selected design element. The contents of the tab pane vary depending on the type of design element selected.
- *Javadocs*– Argo's Javadocs Tab allows you to enter documentation on the selected design element. Documentation templates are provided for many types of design elements to get you started.
- *Source*– Argo's Source Tab allows you to preview the Java source code that will be generated for the selected design element. If you are using JDK (not JRE) and have included `colorize.jar` in your CLASSPATH, you will see the Java source code with keywords, constants, and comments colored differently. This helps make the structure of the code more visible at a glance and makes the code easier to read quickly.
- *Constraints*– Argo's Constraints Tab allows you to enter and view OCL constraints on the selected design element. The Object Constraint Language (OCL) is a simple predicate logic language that allows you to add more meaning to your designs.
- *TaggedValues*– Argo's TaggedValues Tab allows you to enter and view TaggedValues on the selected design element. TaggedValues are simple key-value pairs that are stored with the design element, but are (usually) not interpreted by the system.

- Checklist– Argo's Checklist Tab lists questions for you to consider while making design decisions. Some of these questions are specific, while others are more open-ended. They are similar to the kinds of checklists that many software developers use in design review meetings.
- History– Argo's History Tab shows a time-ordered list of all the criticisms that were raised by design critics and how they were resolved.



### 3.2.6 Creating a class in ArgoUML

The class is the dominant artifact on a class diagram. In the UML metamodel it is a subclass of Classifier and GeneralizableElement.

A class is represented on a class diagram as a rectangle with three horizontal compartments. The top compartment displays the class name (and stereotype), the second compartment any attributes and the third any operations. The last two compartments may optionally be hidden.

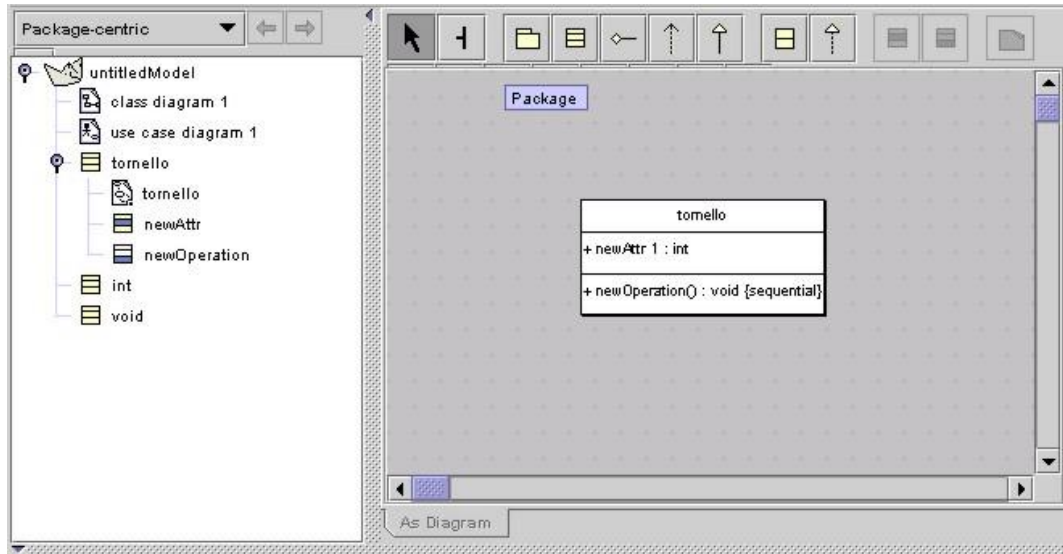
Identifying class diagrams from existing materials (Vision, Use Cases etc).

Classes can be added to the untitled diagram by clicking on the class icon in the toolbar.

The user can set the name of a class or association by simply selecting it and typing.

Double clicking on a name allows the name to be edited. You can double click on the other compartments of the class to edit them.

The "Properties" tab into the details pane shows details of the selected model element.







### 3.2.7 Creating statecharts in ArgoUML

This is what the final appearance of the State diagram inside the Enroll.argo example will look we are finished.

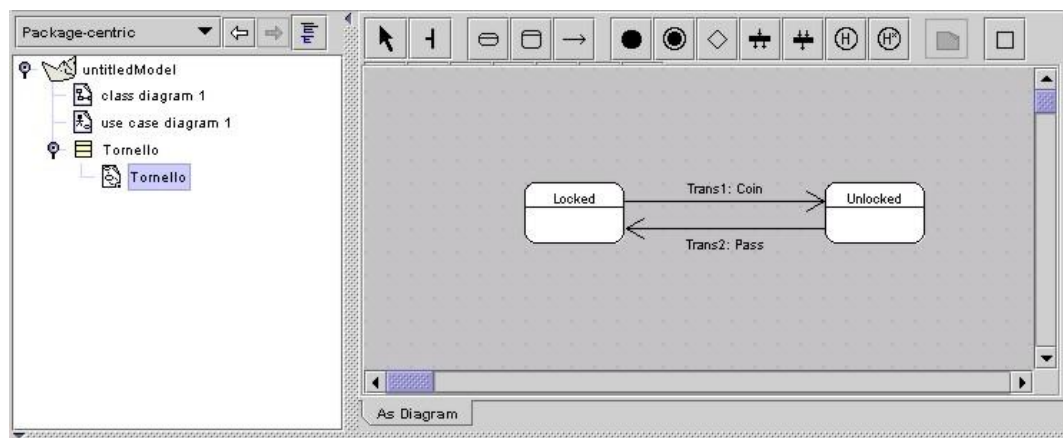
Every state diagram is associated with exactly one class. In this case, the state diagram above is associated with the Grad class from the class diagram of Enroll.argo.

To create a new state diagram, you must first select the class that you would like to make the state diagram for, and then choose Create-Diagrams-StateDiagram. Once this is done, name the state diagram "Grad states". Double-click on "Grad States" to go to the state diagram editor pane.

Use  to place the initial state, and  to place the final states. Next, add and name the Trigger for all of the transitions .

Adding a Composite State: A composite state  contains other states. Place one of these in the diagram and then put the "writeDissertation" and "finalDefense" states within it.

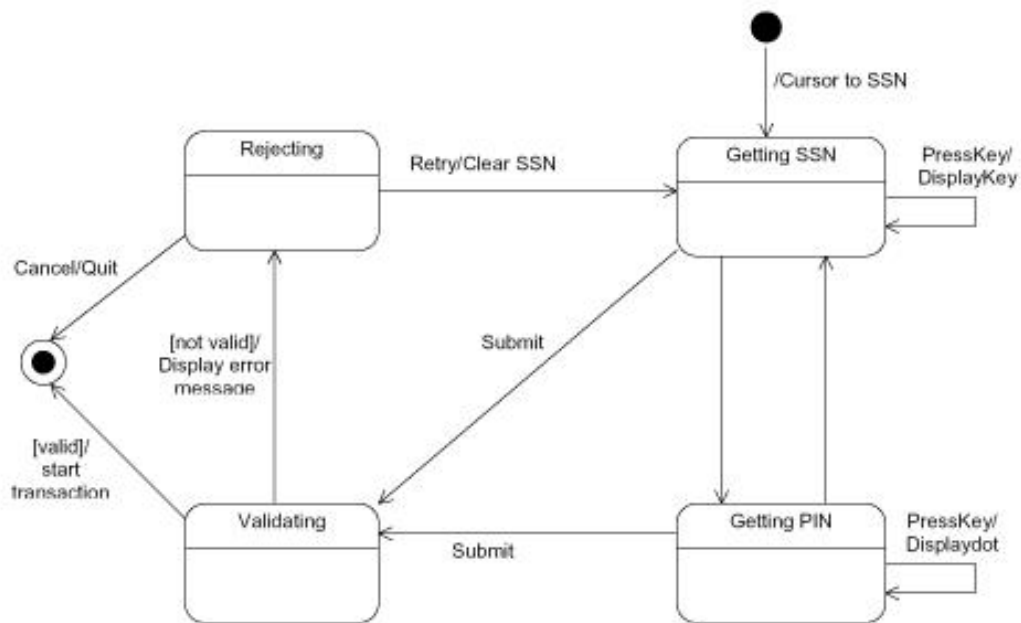
The correct title for this category should be State Machines. This is raised as an issue to be fixed in a future version of ArgoUML.



## 4. From statechart to Java code

A statechart diagram represents a state machine. The state machine belongs to an object that is an instance of such a class of the model: this class should be able to have a dynamic behaviour.

Statecharts represent behaviour of entities in such complex systems and they specify the actions happened after an event. Next there is an example of a state machine.



If we have a class A and it has a statechart S that models its behaviour, the use of the instance of class A is controlled by the statechart S.

In this work we extend class A in such a way that the call of the A-methods is controlled by and only by the statechart S. Without S every call is possible.

There are many techniques of implementing state machine, the most common implies statements like Switch/Case for distinguishing object's states. This solution is not scalable:

if the number of states of the state machine grows, the code will be not readable. More, in this solution we link together the logic of the state machines and its implementation.

So, we chose to implement statecharts referring to Design Pattern<sup>1</sup> State<sup>2</sup>.

## 4.1. Design Pattern and polymorphism

The State pattern is an example of a tremendously useful software pattern that takes advantage of polymorphism. The State pattern uses polymorphism to define different behaviors for different states of an object. It is a valuable pattern to master, because it can be used in practically any sizable application.

Separating behaviors into disparate objects makes sense when the separation takes advantage of *polymorphism*. Polymorphism allows two objects to be treated identically, using the same methods, even though the objects implement these methods in quite different ways.

Polymorphism works because the classes that implement the same method differently both derive, at some point, from a common *superclass*. A general program is written that operates on objects of the superclass type. The program is oblivious to the fact that what it thinks of as a superclass object is, in reality, an object that is a member of one of the *subclass* types. When the program invokes a method defined in the superclass, the method that gets called is actually the subclass method that overrides the superclass version.

Polymorphism allows very general programs to be written for a superclass, letting the subclasses take care of the details. An example of this is a drawing program that can

---

<sup>1</sup> See [6] and [1].

<sup>2</sup> See [1], [8] and [11].

write a loop that processes a list of graphic objects including lines, squares, and circles. It can work from the bottom most graphic object to the top, invoking the draw() method on each object in turn. The drawing program manages the sequence of drawing operations, which depends on which objects are "on top." Each object, in turn, is responsible for drawing itself. The program works on objects of type Graphic, knowing that every Graphic object always implements the draw() method. The subclasses Line, Square, and Circle, all derive from the superclass Graphic, and each overrides the draw() method in its own fashion. The program can then be extended by adding an Ellipse or Rectangle class. As long as the new classes derive from Graphic, and as long as they implement the draw() method appropriately, the basic drawing program continues to work, unchanged.

The next part describes the state pattern and then how we use polymorphism to realize and implement the state machine.

### **4.1.1 State Pattern**

Now we describe the state pattern.

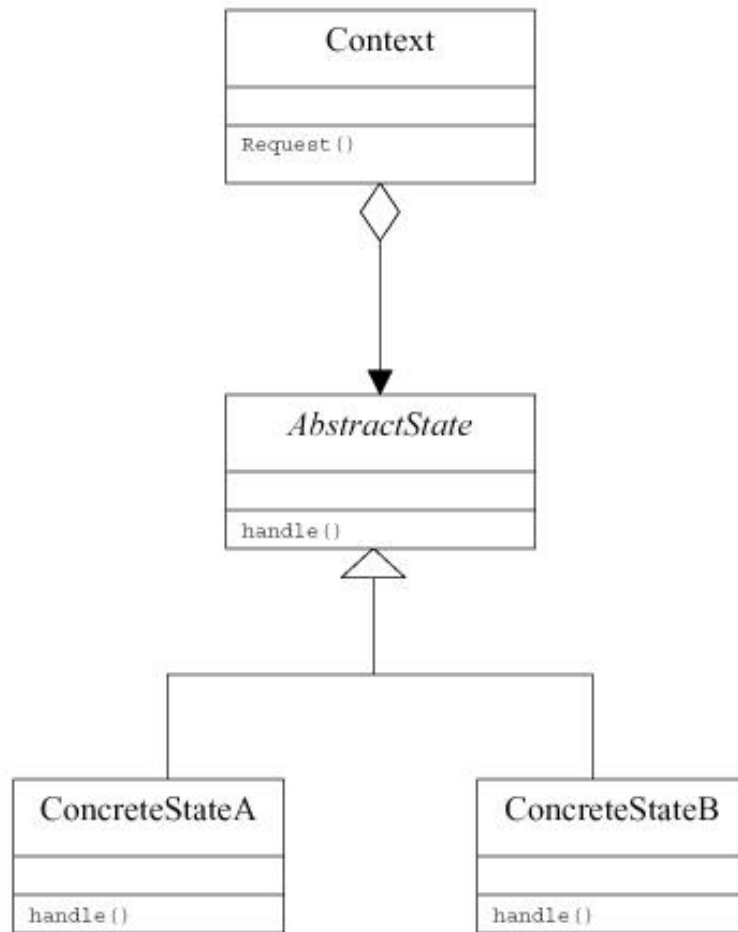
#### **Intent**

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

#### **Motivations**

Objects have behaviours that depends to the system they belong to. With this pattern, we can control the flux of the events, taking care of leave distinct the instance of the object and the instances of class that represent the state machine.

## Structure



Some code structure may be the next one.

```
class Context
{
    State state = new ConcreteState1(this);
    request() // Un evento e accade
    {
        state.handle();
    }
    state = AbstractState.getNextState(e);
}

abstract class AbstractState
{
    static final AbstractState stateA =
        new ConcreteStateA();
    static final AbstractState stateB =
```



```

                                new ConcreteStateB();
    protected State();
    abstract public void  handle(Event e);
}

class ConcreteState1 extends State
{
    public void handle(Event e)
    {
        oggetto.metodo_appropriato ();
    }
    public State getNextState(Event e)
    {
        return stato_appropriato;
    }
}
class ConcreteState2 extends State { ... }

```

## Participants

### Context class

Defines the interface of interest to clients and it maintains

### State classes

The State class is an abstract class. It provides some basic behavior, but its only real purpose is to be extended to produce one or more "real" state classes. In this case, two state classes are defined, State1 and State2. To create additional states, you would copy one of these classes.

```

abstract class AbstractState { ... }
class ConcreteState1 extends AbstractState { ... }
class ConcreteState2 extends AbstractState { ... }

```

### Static variable

The State class defines three variables:

```
static State initialState;  
static final State state1 = new ConcreteState1();  
static final State state2 = new ConcreteState2();
```

The first variable is the default initial state. The first state that gets created will be stored in that variable, in case the StateOwner object wants to use it. (It doesn't have to.) The next two variables create the state objects using the classes State1 and State2.

Because these variables are defined as static, they belong to the State class. In other words, they are *class variables*, which means they belong to the class as a whole, rather than to individual objects. Without the static keyword, they would be *instance variables*, meaning that each object created using the class (each *instance* of the class) would have its own variable.

In general, using the static keyword means there is only one copy of the variable for the entire class, no matter how many objects are created using that class. Such variables generally are used for class-wide data, such as a count of the number of objects that are created using the class.

## Constructor

The next significant part of the State class is the constructor, which saves the first state created as the default initial state. The important feature of the constructor is that it is defined as *protected*:

```
protected State()  
{  
    if (initialState == null) initialState = this;  
}
```

Because the constructor is protected, it cannot be used by other classes to create new objects. It can only be used by the State class and by those classes that extend the State class.

Since the State class is abstract, only the subclasses have access to the constructor, which is equivalent to making the subclass constructors private. In other words, the subclasses can create themselves using static variables, but they can never be used at run time to create additional objects.

In this case, the abstract State class creates the static variables state1 and state2. The protected constructor forms a "closed loop" with these static variables. The creation of the static variable state1 accesses the State1 constructor, which invokes the State constructor, which is protected. The final result is that the state variables can only be created from within the State class and its subclasses. They form a predetermined set that is defined at compile time.

## **Operations**

The last part of the State class can define some abstract methods.

Because these methods are defined as *abstract*, they have no method body. That means they must be implemented by the subclasses that extend the State class. That, fundamentally, is the reason for defining an abstract class in the first place -- to specify the kinds of methods the individual states will provide, and leave the implementation up to them.

The State class says that each state will have two methods, one for when you exit (or leave) a state, and one for when you enter (or arrive at) a state. Although you do not always need both methods, frequently it is helpful to have them. In multiple state systems, the combination of leaving one state and arriving at another can completely define the actions needed for a transition between the two states.

## 4.2. State pattern for code generation

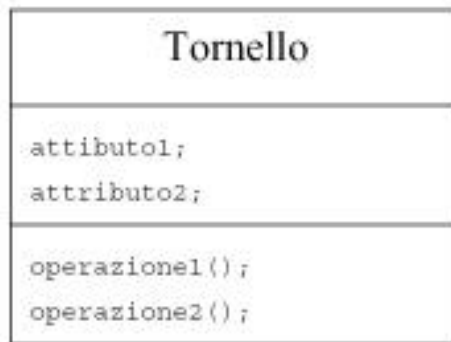
For explaining use and advantages of State pattern, we referred to an example: modeling a access control system, such as the metropolitan one. In this area user finds a turnstile before entering to the trains area. A turnstile is illustrated next.

In this context, we can find in the model, a class “Turnstile”.



The behaviour of a turnstile can be expressed in this way:

- If the turnstile is locked and user insert coin, turnstile becomes unlock and user can pass over.
- If the turnstile is locked and user try to pass, turnstile set an allarm.
- If the turnstile is unlocked ad user pass over, the turnstile returns locked.
- If the turnstile is unlocked and user inserts coin, turnstile may thanks!



The “Turnstile” class is represented above.

The statecharts may be very simple, but in the context of “Turnstile” class, it may become difficult recognizing the behaviour logic of the class from the implementation of the state machine, if it had implemented as switch/case statements. Far from these problems, we use State pattern for implementing the statechart (displayed next).



For generating some code, we assumed that the designer wanted to represent events (such as “Coin” and “Pass”) by operations declared in “Turnstile” class. We supposed that in the CASE tool the designer:

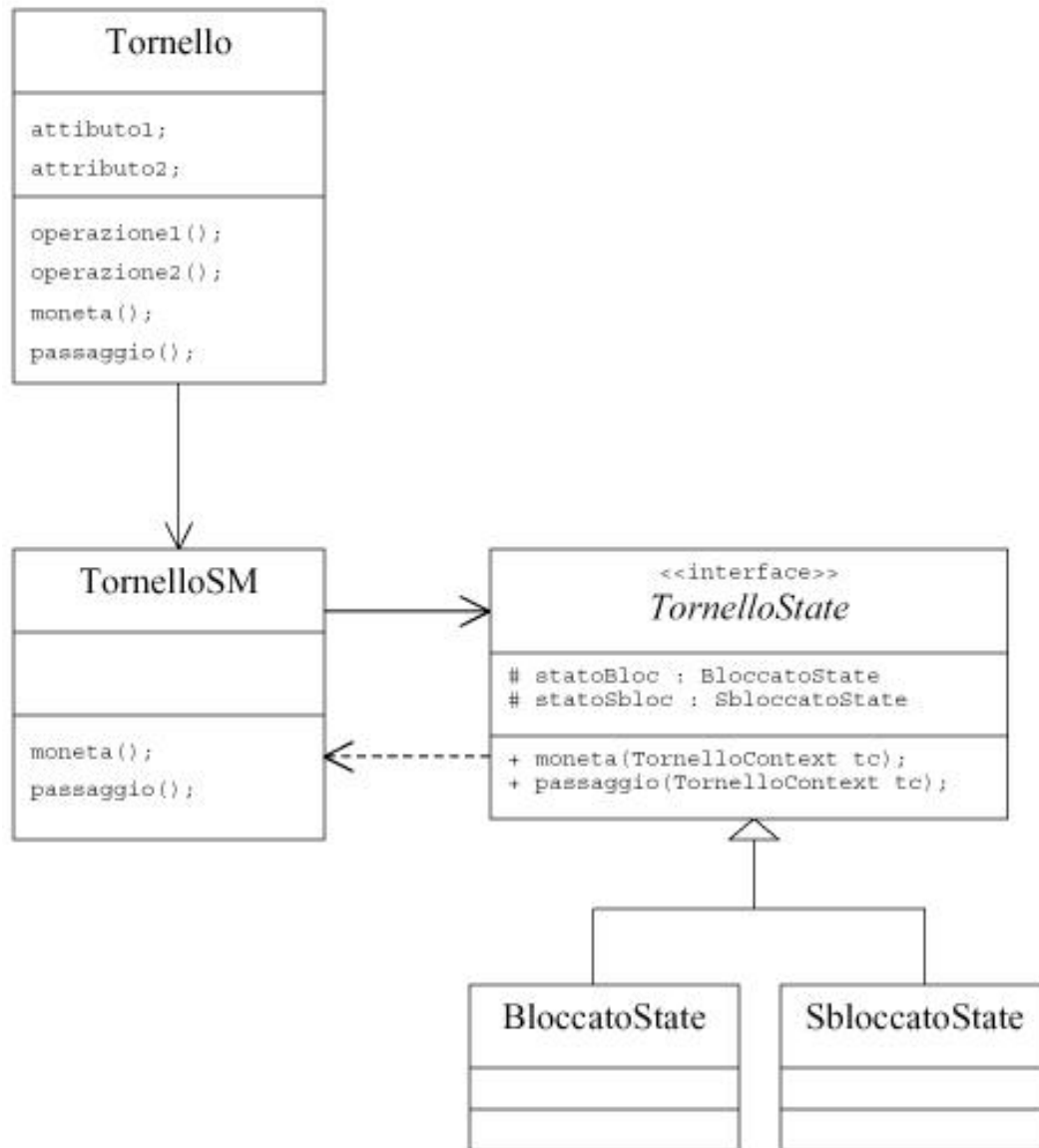
1. add a class “Turnstile” in the class diagram, adding to it its attributes and operations not directly regarding the dynamic behaviour.

2. select the class and add a state diagram with complete description of events, actions and transitions.
3. then the CASE tool will add events and actions as operations of “Turnstile” class in this way:
  - a. adding declarations of event to “Turnstile” class.
  - b. creating a new class “TurnstileSM” containing declarations of the events plus actions.

The code generation work in this way: starting from the “Turnstile” class and its statechart, we create many classes, *nameofState*, representing the states of statechart. These classes implement all events and actions operations declared before in “Turnstile” class and in “TurnstileSM” class.

Only the operations regarding that state can be called in such a *nameofState* class.

In our example, “Turnstile” class is derived in “TurnstileSM” class, that represent the behavioural context, then in the “TurnstileState” class (an interface) and from it we create the classes *nameofStat*, as illustrated next..



All static members in “TurnstilState” class make sure that will be one and only one instance of *nameofstate* classes in this context.

This architecture has many advantages: the dynamic behaviour of the turnstile is separates from the context of the whole system.

The code structure generated can be the next.

```

class Turnstile
{
    attributel;
    attribute2;

    public void op1();
    public void op2();
    public void coin();
    public void pass();
}

class TurnstileSM
{
    public void coin();
    public void pass();
}

abstract class TurnstileState
{
    static LockState _stlock =
                                new LockState();

    static UnlockState _stunlock =
                                new UnlockState();

    void SetState(TurnstileState);

    public void coin(TurnstileSM tc);
    public void pass(TurnstileSM tc);
}

class LockState extends TurnstileSM
{
    public void coin(TurnstileSM tc)
    {
        tc.SetState(_stunlock);
        tc.unlock();
    }

    public void pass(TurnstileSM tc)
    {
        tc.alarm();
    }
}

class UlockState extends TurnstileSM
{

```



```

    public void moneta(TurnstileSM tc)
    {
        tc.thanks();
    }

    public void passaggio(TurnstileSM tc)
    {
        tc.SetState(_stlock);
        tc.unlock();
    }
}

```

The code generated is open and any change doesn't have consequences on the whole system code implemented.

This structure is scalable and reusable: we add any behaviour while the system grows, maintaining these behaviour separated from the system model.

This work add these functionalities to ArgoUML: we can generate java code from the state diagram and class diagram.

## 5. ArgoUML code organization

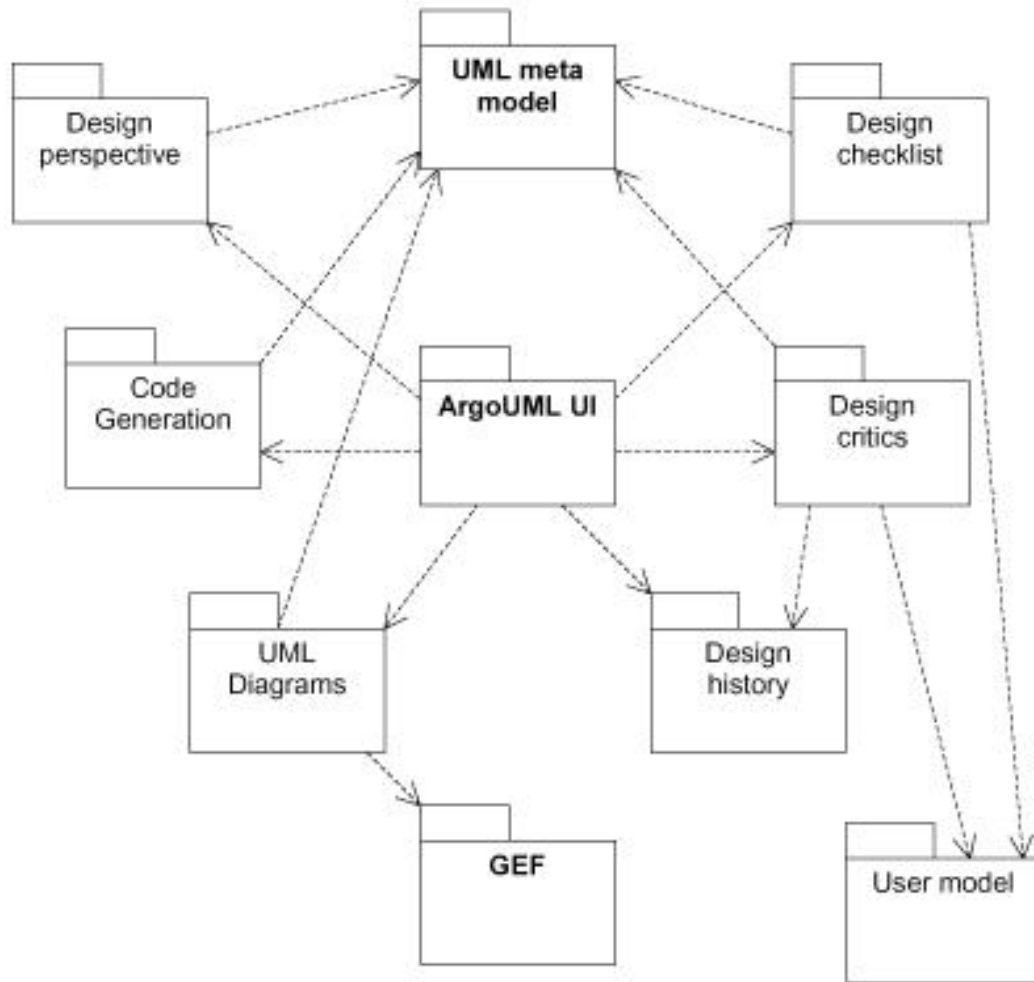
### 5.1. General architecture

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.

#### 5.1.1 Packages

ArgoUML itself breaks down into several components. The graphic does not reflect the true package names, neither the true dependencies of the packages.



There are three major libraries that make up the bulk of the Argo/UML source code. Each is made up of several Java packages. Argo/UML also contains several packages itself.

The numbers next to each package indicate the number of classes in that package. Each package is described below.

### ***GEF (170 classes)***

- `uci.gef` (105) -- reusable graph editing framework: shapes, selections, grids, editors, layers, commands
- `uci.gef.demo` (19) -- demonstrations how to use GEF
- `uci.gef.event` (2) -- events generated from GEF editors

- `uci.graph` (15) -- generic interface between GEF and graph representations, similar to Swing's `TreeModel`
  - `uci.util` (12) -- general purpose utilities
  - `uci.ui` (13) -- property sheet and other small user interface utilities
- `uci.beans.editors` (4) -- custom property editors used in property sheet

### ***UML meta-model (120 classes)***

- `org.argouml.uml.Foundation.Core` (26) -- from UML Semantic Specification
- `org.argouml.uml.Foundation.Data_Types` (22) -- from UML Semantic Specification
- `org.argouml.uml.Foundation.Extension_Mechanisms` (2) -- from UML Semantic Specification
- `org.argouml.uml.Behavioral_Elements.Common_Behavior` (16) -- from UML Semantic Specification
- `org.argouml.uml.Behavioral_Elements.State_Machines` (20) -- from UML Semantic Specification
- `org.argouml.uml.Behavioral_Elements.Use_Cases` (2) -- from UML Semantic Specification
- `org.argouml.uml.Behavioral_Elements.Collaborations` (6) -- from UML Semantic Specification
- `org.argouml.uml.Model_Management` (5) -- from UML Semantic Specification
- `org.argouml.uml.test.omg` (15) -- examples from UML Notation Guide
- `org.argouml.uml.generate` (3) -- placeholder for future code generation features

### ***Argo (38 classes)***

- `org.argouml.kernel` (34) -- reusable framework for critics, "to do" list, design history
- `org.argouml.cognitive.checklists` (4) -- framework for design checklists

### ***Argo/UML (191 classes)***

- `org.argouml.uml` (1) -- Main
- `org.argouml.uml.ui` (111) -- Argo/UML windows, dialogs, panels, navigational perspectives, property panels
- `org.argouml.uml.visual` (25) -- UML diagram definitions, UML node and arc definitions
- `org.argouml.uml.cognitive.critics` (44) -- design critics
- `org.argouml.uml.cognitive.critics.java` (1) -- design critics
- `org.argouml.uml.cognitive.critics.patterns` (3) -- design critics
- `org.argouml.uml.cognitive.checklist` (6) -- design checklists
- `org.argouml.Images` -- gif images for splash screen and toolbar icons

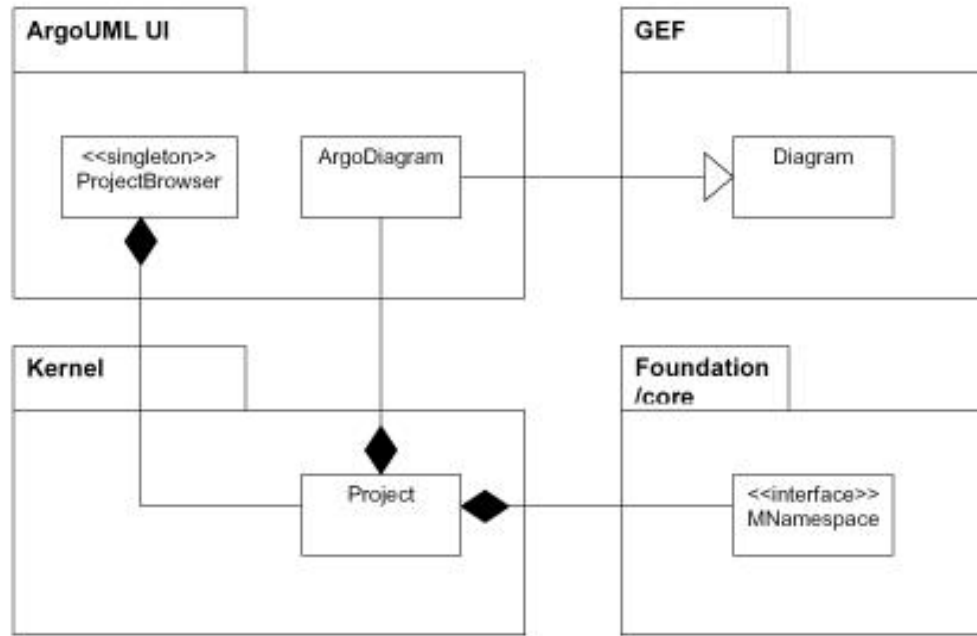
## 5.1.2 ArgoUML core

ArgoUML uses Ant as it's main build tool. The whole build process is controlled by the `build.xml` file in the main source directory.

The parser for Java reverse engineering is generated with Antlr from a customized Java grammar.

It all begins in `org.argouml.application.Main`: set up main application frame (`org.argouml.ui.ProjectBrowser`), the project (`org.argouml.kernel.Project`), numerous classes, and finally as a background thread: cognitive support (`org.argouml.cognitive.Designer`) and some more classes. The `ProjectBrowser` initializes the menu, toolbar, status bar and the four main areas:

- navigation pane (`org.argouml.ui.NavigatorPane`),
- editor pane (`org.argouml.ui.MultiEditorPane`),
- to do pane (`org.argouml.cognitive.ui.ToDoPane`),
- details pane (`org.argouml.ui.DetailsPane`).



All of the uml diagrams and models in a project are contained in a single Project instance. 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 persistence. 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 appropriate 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.

The ProjectMemberDiagram.save method creates an OCLExpander instance and generates a PGML document .

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.

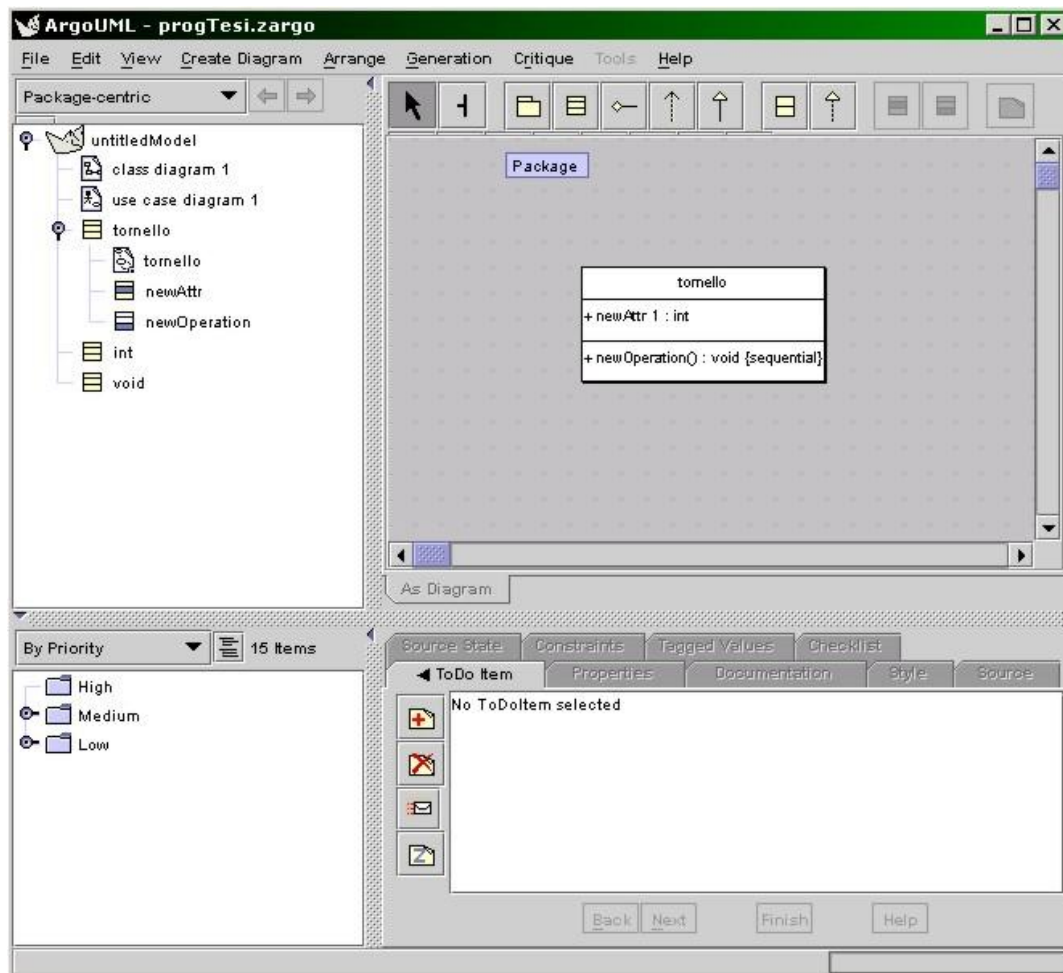
Restoring the state of a project from the XML documents is similar. An XMIREader is responsible for recreating the model, and OCLExpander instances recreate the project status and the diagrams.

### **5.1.3 ArgoUML main window**

Each of the classes are designed to be subclassed. In the links, there will be a hierarchy showing all of the classes that currently inherit from this class, and also what part of Argo/UML's main window they are associated with.

Throughout the code of there are groups of classes that share inheritance hierarchies, and play a key role in how the program works.

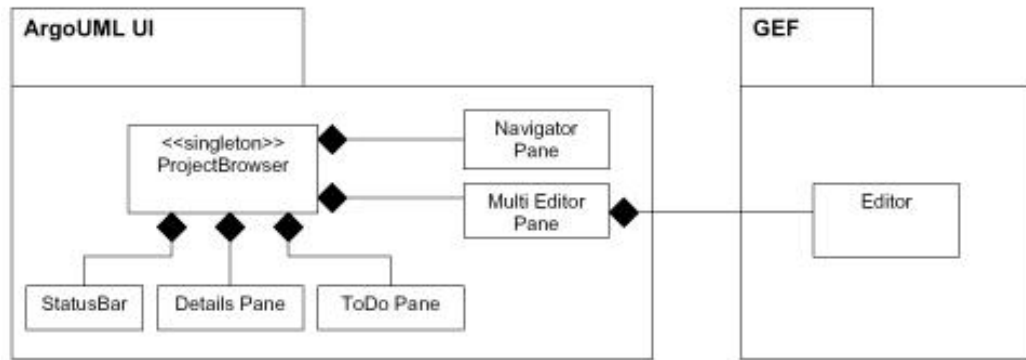
Below is the Argo/UML main window. In the window, there are four panes. The Navigation pane (top left), Editor pane (top right), To Do pane (bottom left), and the Details pane (bottom right).



The main ArgoUML window is the singleton ProjectBrowser, containing classes for each of the windows panes.

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.

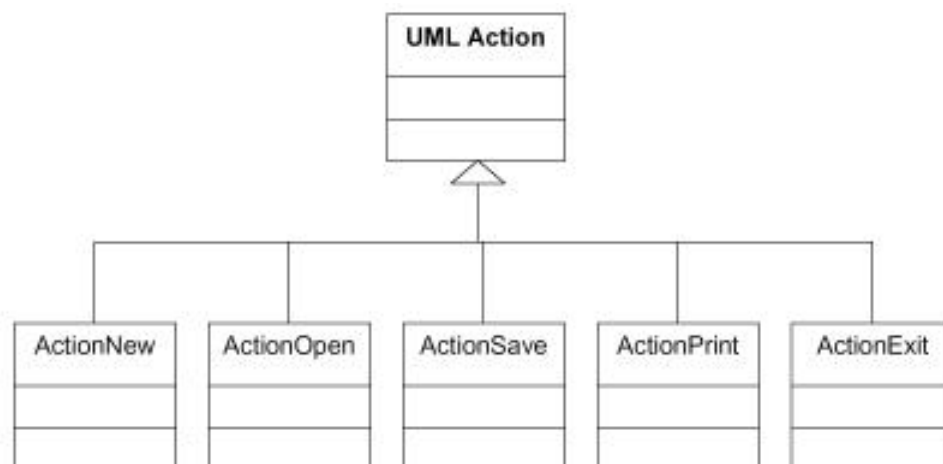




## 5.1.4 Menu

It is based on the swing class `JMenu`, which requires a class implementing `MenuModel` from which it gets the tree's data.

There are many more classes that extend `UMLAction`, but many have been omitted for the sake of clarity. Notice that the two images above are parallel because they represent the same actions. This diagram could be repeated for each of the menus.

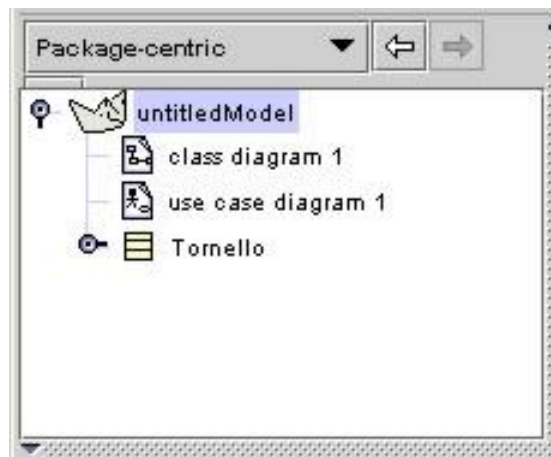


## 5.1.5 Navigator pane

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.

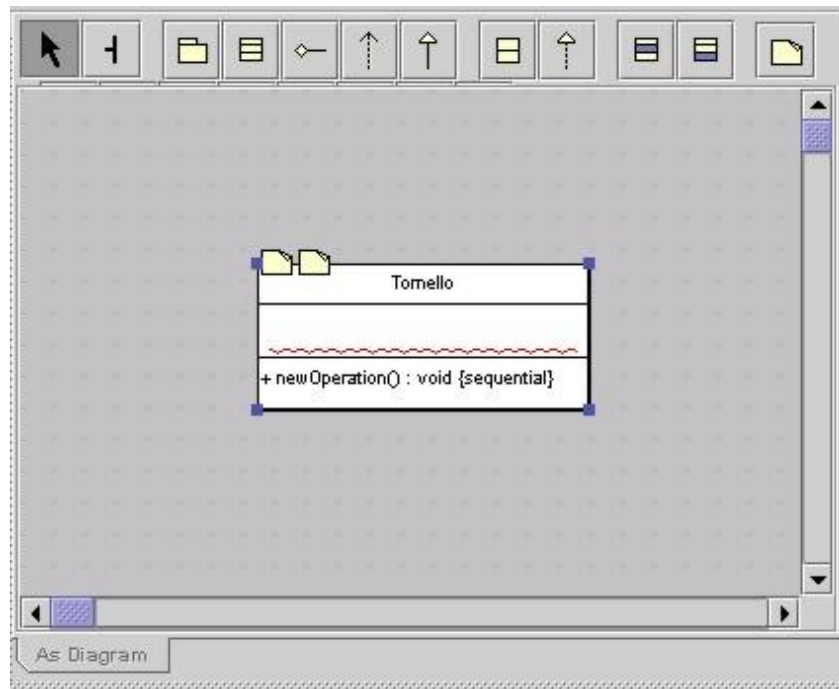
Since the user's model is a graph and not necessarily a tree, there are many ways (perspectives) to represent the model in a tree-like structure. One possibility is to start with the model, branch into the namespaces and packages (and subpackages), branching further into the individual classes and finally to the class attributes and operations (which are leaves in the tree).

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 `GoParentToChild` classes realizing the `TreeModelPrereqs` interface. All the rules known to the system are registered in the static vector `NavPerspective.rules`.



## 5.1.6 Editor

The type of diagram that the figure below is displaying is a class diagram. This class can be extended to create other types of diagrams that aren't currently implemented from the UML specification.



ArgoUML uses the GEF library for the various UML diagram types. 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.

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.

Actors and use-cases are considered nodes, since they can exist independently. Generalizations and dependencies are edges which connect the nodes. These figure classes represent the metaclasses graphically in a specific diagram.

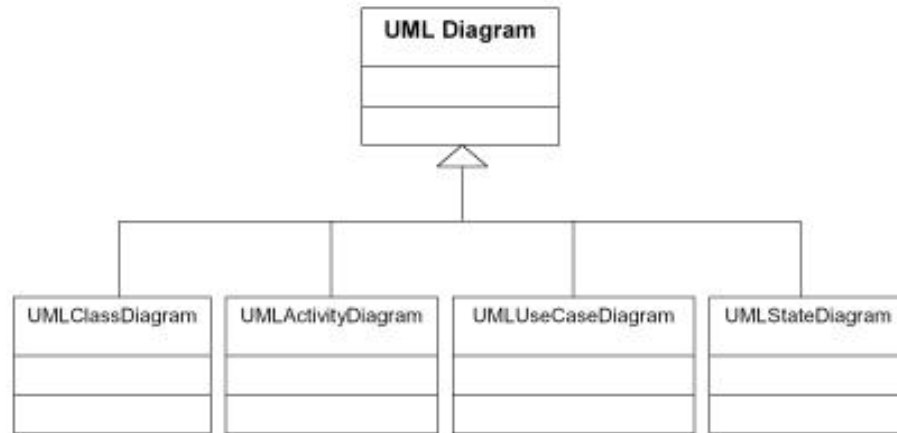
Implementing the various UML diagrams (static structure, use-case, sequence, etc.) is surprisingly straightforward. Due to the supporting framework, only a few classes for each diagram type are necessary. Each UML diagram type requires:

a class derived from Diagram. This class should contain member objects for each

command (usually Cmd-Subclasses) the user may trigger. ArgoUML supplies the classes ArgoDiagram which uses the observer pattern 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.

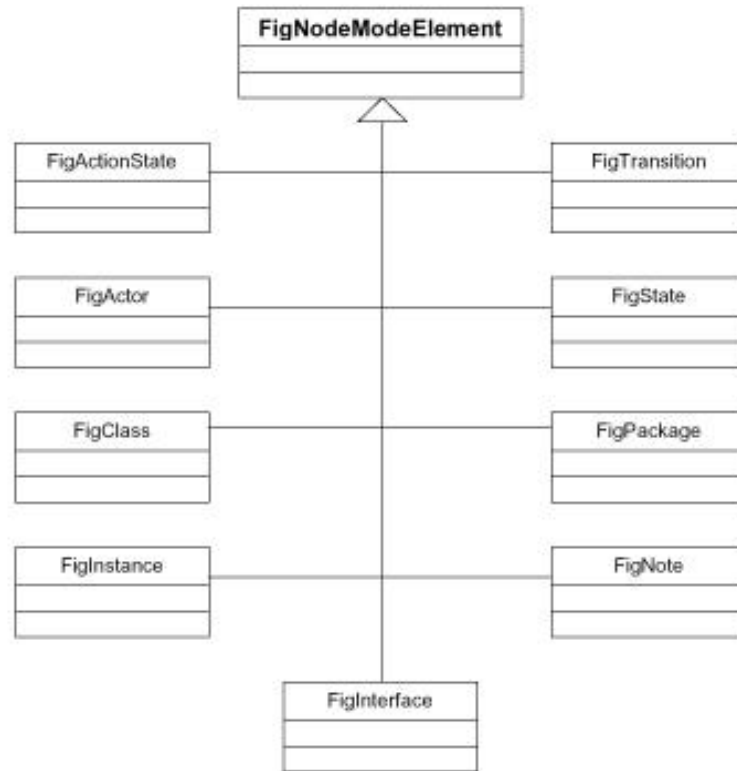
a displayable Fig subclass for each UML metamodel element which can be inserted in the diagram.

1. a factory class realizing the GraphEdgeRenderer and GraphNodeRenderer interfaces which can create the necessary Fig instances on demand.
2. a adapter class 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.



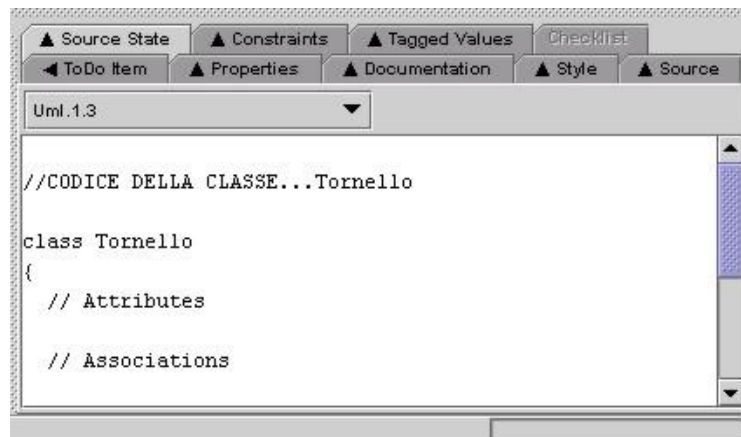
The associations (lines) drawn between the figures shown in the Editor pane of the window are subclasses of `FigEdgeModelElement`.

From looking at the window below, figures in the Editor pane are elements of this class. So to make a new diagram type, this class would need to be extended. In this particular diagram, there are four instances of `FigClass` (shown third from the left in the above hierarchy).



### 5.1.7 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 attributes and references of the currently selected model element. Its contents is dependent on the metaclass of the element { a class has different attributes 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.

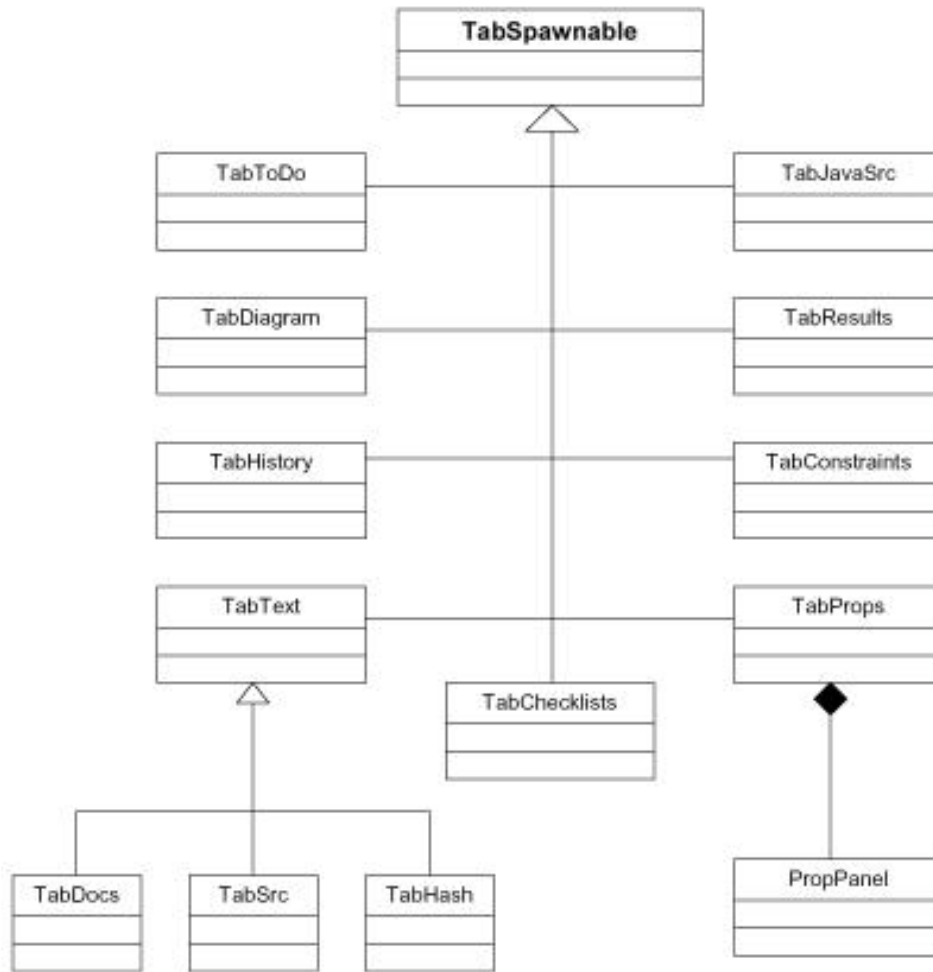
Source Tab lets the user enter implementation details directly into the model.

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.

The property tabs are located below in the Navigation pane. To add another tab, TabSpawnable would have to be inherited from.





## 6. ArgoUML libraries

### 6.1. GEF

GEF, the thing which is hidden in `gef.jar`, is responsible for displaying the graphs and which gives you the ability to move things around. GEF used to be closely interwoven with Argo, but is now more or less separate.

The goal of the GEF project is to build a graph editing library that can be used to construct many, high-quality graph editing applications. Some of GEF's features are:

Node-Port-Edge graph model that is powerful enough for the vast majority of connected graph applications.

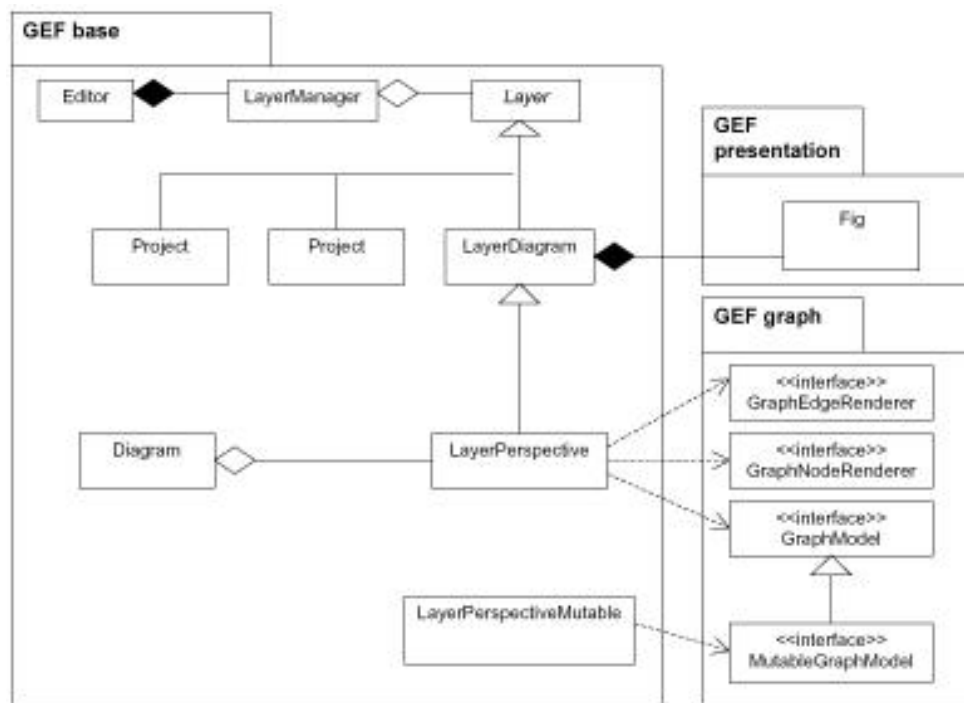
Model-View-Controller design based on the Swing Java UI library makes GEF able to act as a UI to existing data structures, and also minimizing learning time for developers familiar with Swing.

XML-based file formats based on the PGML standard (soon to support SVG).

#### 6.1.1 Characteristics

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 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.

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 connect ports with each other.



## 6.1.2 Layers

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 individually. `LayerDiagram` is a layer which contains a collection of view elements (class `Fig`).

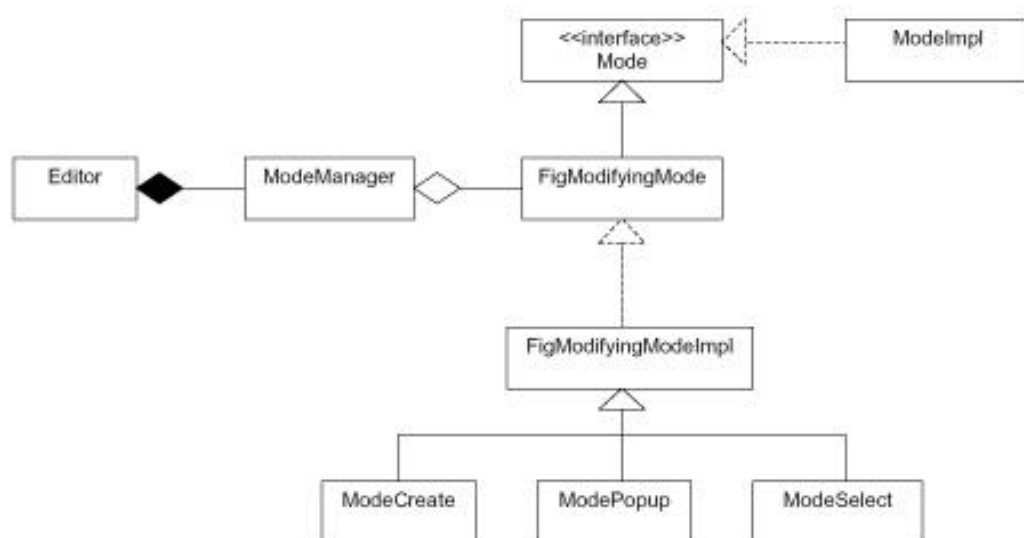
`LayerPerspective` extends on this by assuming that the model is a connected graph. Each view element represents a node or an edge in the model.

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.

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.

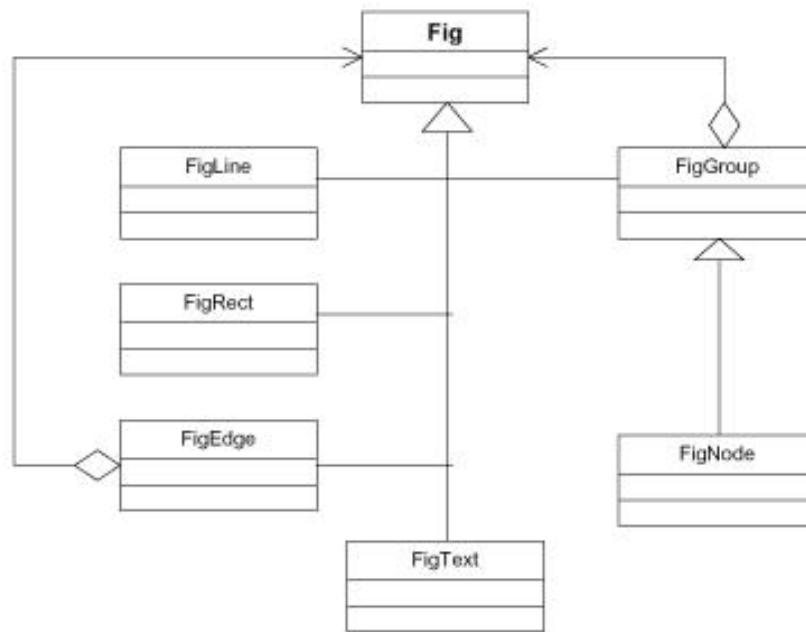
### 6.1.3 Editor

The `Editor` class acts as a mediator that holds the other pieces together and routes messages among them.



## 6.1.4 Figs

Figs (short for figures) are the primitive shapes; for example, `FigCircle` draws a circle and `FigText` draws text.



## 6.1.5 Selections

Selections keep track of which Figs are selected and the effect of each handle; for example, `SelectionResize` allows the bounding box of a `Fig` to be resized, while `SelectionReshape` allows individual points of a `FigLine` or `FigPoly` to be moved.

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 behavior 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

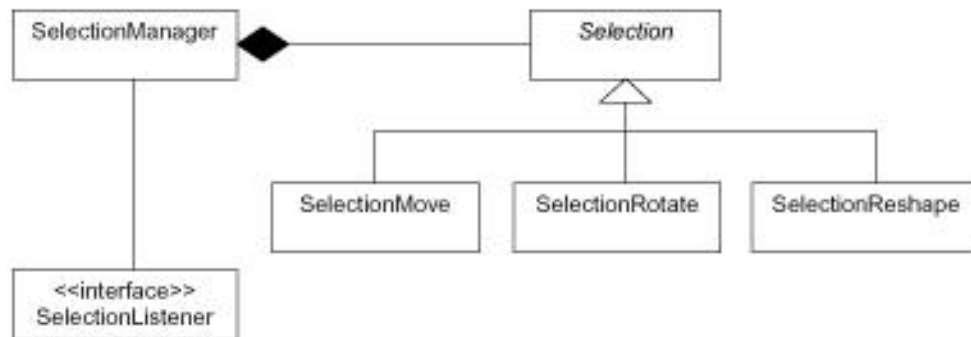
SelectionNoop does not allow the user to do manipulate the figure.

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 bounding box.

SelectionRotate allows the user to rotate the figure.



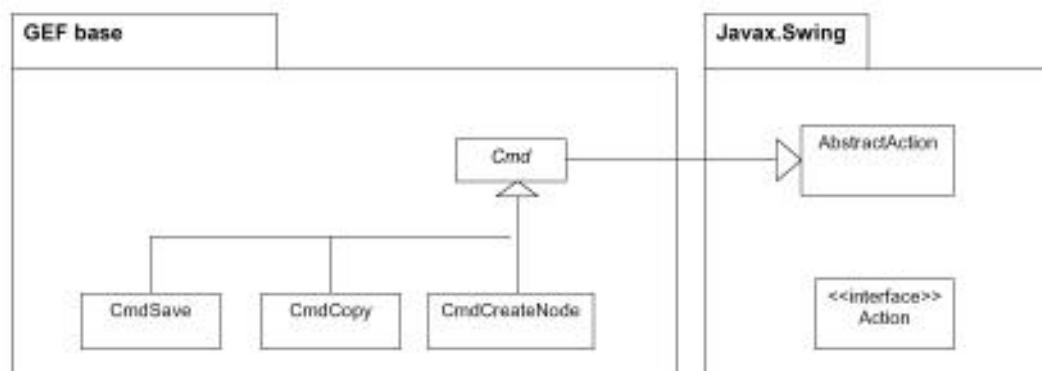
## 6.1.6 Commands

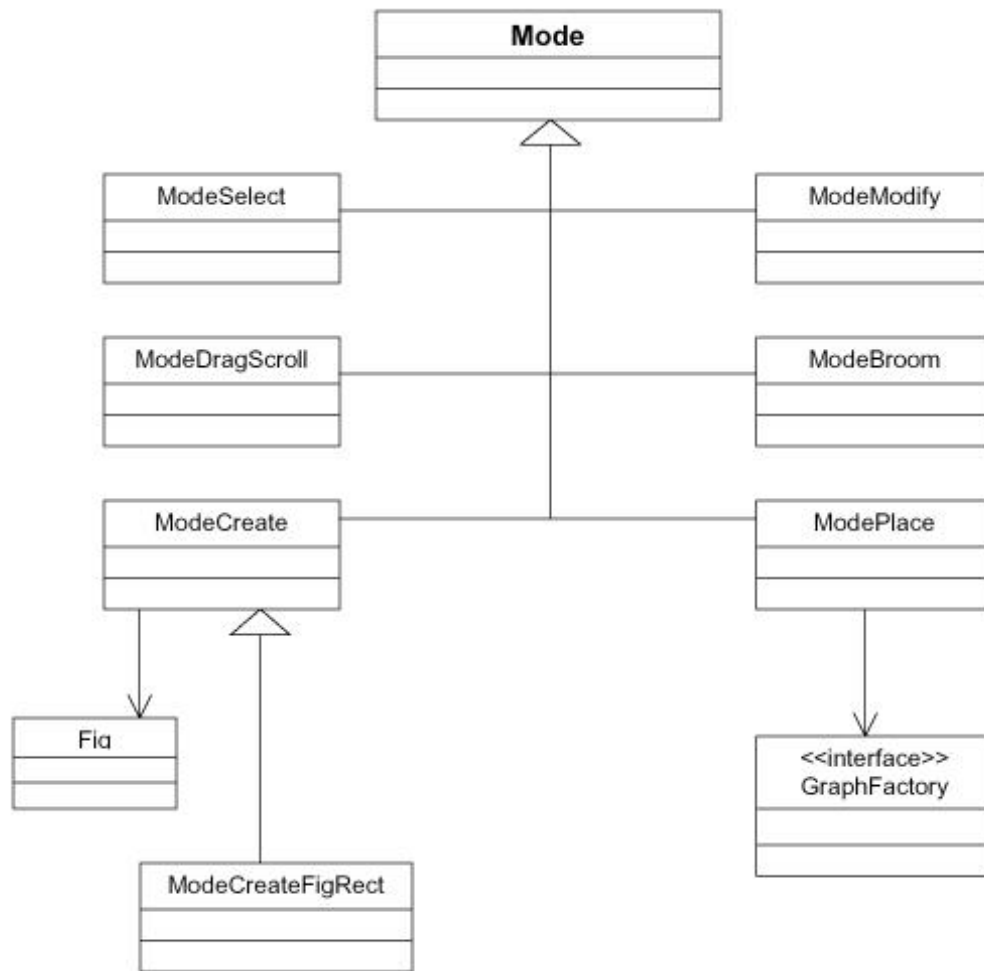
`Cmds` (short for commands) make modifications to the `Figs`; for example, `CmdGroup` removes the selected `Figs` from their `Layer` and adds a new `FigGroup` in their place.

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.

## 6.1.7 Modes

`Modes` are objects that process user input events (e.g., mouse movement and clicks) and execute `Cmds` to modify the `Figs`; for example, dragging in `ModeSelect` shows a selection rectangle, while dragging in `ModeModify` moves the selected objects. I have made central those concepts that are familiar to diagram editor users and avoided those that are unfamiliar or too abstract; for example, GEF does not use the decorator pattern (Gamma et al., 1995) or attempt to offer general purpose constraint solving (e.g., Sannella, 1994).





## 6.2. NSUML

### 6.2.1 UML Metamodel implementation

Novosoft metadata framework is based on JMI specification and generated classes that are required by JMI specification and also provides additional services like event notification, undo/redo support, XMI support. NSMDF is local in-memory implementation.

It also provides code generated from UML 1.4 metamodel. Which could be used for constructing applications based on UML 1.4.

NSMDF project is based upon NSUML 0.4.\*, which is still provided as separate download. NSUML 0.4.\* is being phased out.

## **6.2.2 Packages structure**

The complexity of the UML metamodel is managed by organizing it into logical packages. These packages group metaclasses that have strong cohesion with each other and loose coupling with metaclasses in other packages. The Foundation and Behavioural Elements packages are further decomposed.

## **6.2.3 Foundation**

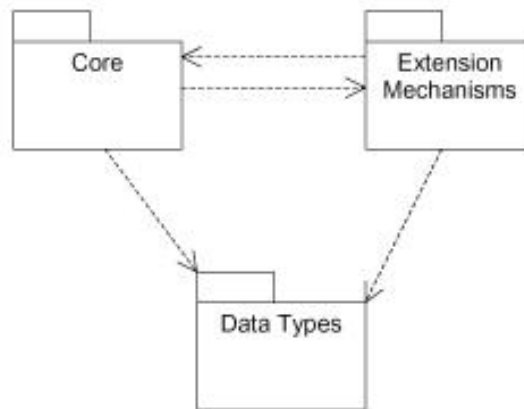
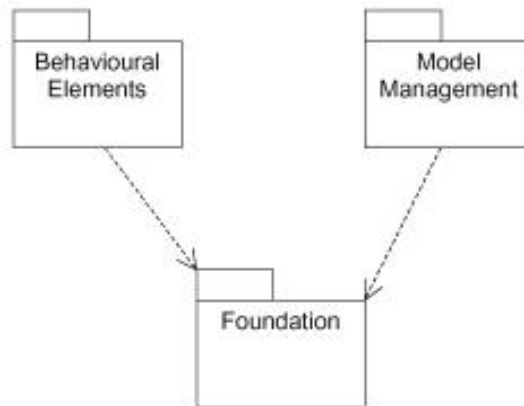
The Foundation package is the language infrastructure that specifies the static structure of models. The Foundation package is decomposed into subpackages: Core, Data types and Extension Mechanisms.

The Core package specifies the basic concepts required for an elementary metamodel and defines an architectural backbone for attaching additional language constructs, such as metaclasses, metaassociations and metaattributes.

The Extension Mechanisms package specifies how model elements are customized and extended with new semantics.

The Data types package defines basic data structures for the language.





## 6.2.4 Behavioural Elements

This Behavioural Elements package is the language superstructure that specifies dynamic behaviour or models. The behavioural Elements package is decomposed into the following subpackages:

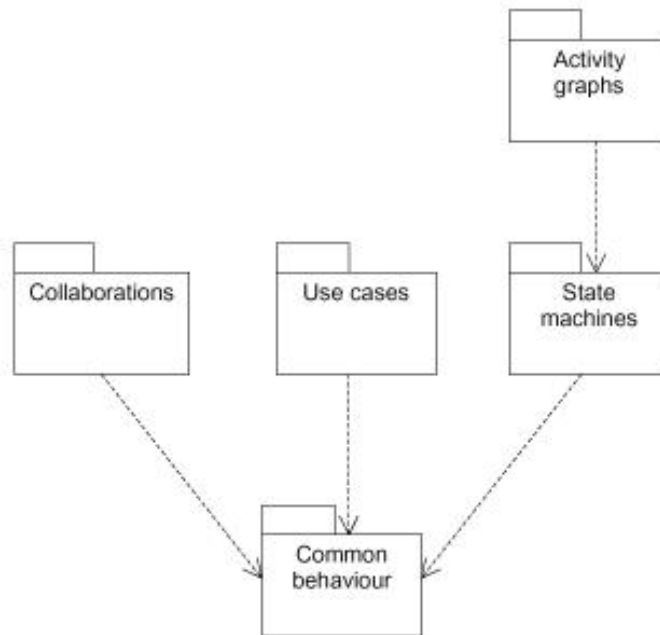
**Common Behaviour** It specifies the core concepts required for behavioural elements.

**Collaborations** It specifies a behavioural context for using model elements to accomplish a particular task.

**Use Cases** It specifies behaviour using actors and use cases.

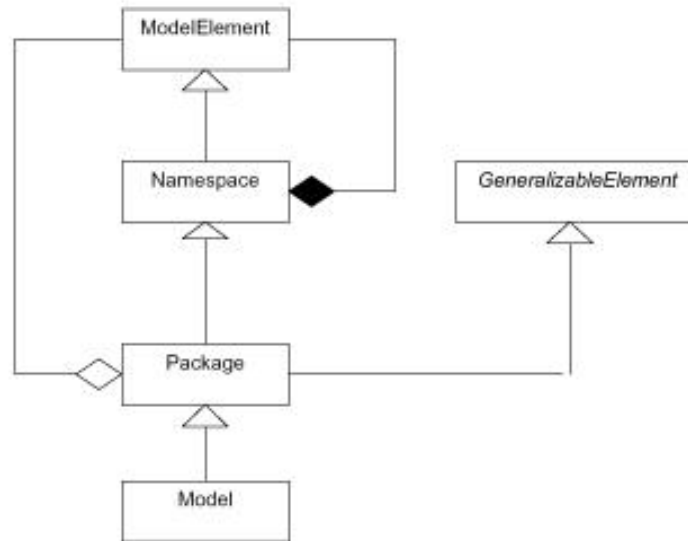
State Machines It defines behaviour using finite-state machine transition systems.

Activity Graphs It defines a special case of a state machine that is used to model processes



## 6.2.5 Model Management

Il pacchetto Model Management offre gli strumenti per organizzare un progetto, come, per esempio, i packages. La struttura è riportata in **Errore. L'origine riferimento non è stata trovata..**



## 6.2.6 API Description

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.

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

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

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

UML primitives	Java Type
Boolean	boolean
Name	String

Integer	int
UnlimitedInteger	int
LocationReference	String
Geometry	String

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

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.

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

For each enumeration literal, public static final instances are created on initialisation.

There will be exactly one (immutable) instance for each literal for the entire lifetime of the system<sup>5</sup>, ensuring that identity comparisons and equality comparisons will always have the same result.

Datatypes are UML data types with no stereotypes.

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 (accessible as `MMultiplicity.M0_1`, etc.).

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

**Tabella 1 – Mappatura del tipo “datatypes”**

Elements are UML metaclasses constituent of a UML model.

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.

all NSUML interfaces are derived from the interface MBase.

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

## 6.2.7 Accessing and modifying metaattributes

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

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!
```

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 (getAttribute or isAttribute ) and mutator (prefix setAttribute) methods.

The default operation are of three kind:

Get – returning the value or a collection of values.

Set – setting the values of a objec type. This doesn't return.

Is – querying type. This return a boolean that says if an attributes has a property.

MNameOfElement – these are the interfaces that define the methods to access to the attributes of the model ( its name is the same of the NameOfModelElement).

MNameOfModelElementImpl – these are classes that realize interfaces defined pervious. In these classes there are attributes defining properties of the element of the model that they represent (i.e. MOperationImpl class realize the MOperation interface and it realize all methods defining relevant attributes).

## **6.2.8 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 similarly to attributes with getRolename and setRolename methods.

Bag roles are unordered with a multiplicity different from 0..1 and 1. The getRoles 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 setRoles all the roles can be set at once. The addRole and removeRoll 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 `addRole (int, OppositeRole )`, `removeRole (int)`, `setRole (int, OppositeRole )` and `getRole (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 `setRole` ), the object first checks whether a link already exists. If this is the case, it tells the existing opposite end to remove the link with a `all` to `internalUnrefRole`. If the call to `setRole` 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 `internalRefRole`.

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 `internalUnrefRole` to



update the state at the opposite end of the association. Then any new links are added (again calling `internalRefRole` to update the opposite end).

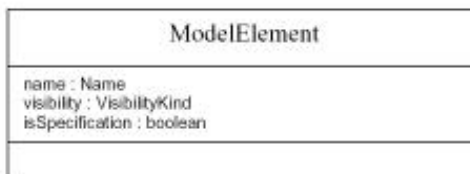
## **6.2.9 NSUML reflective API**

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.

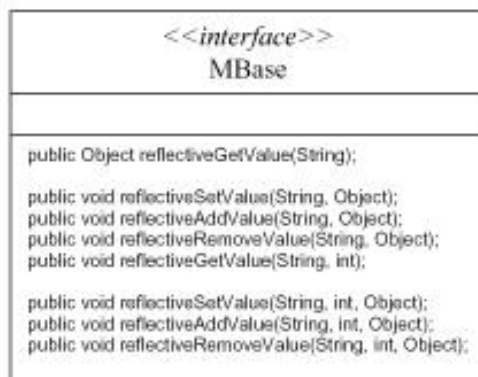
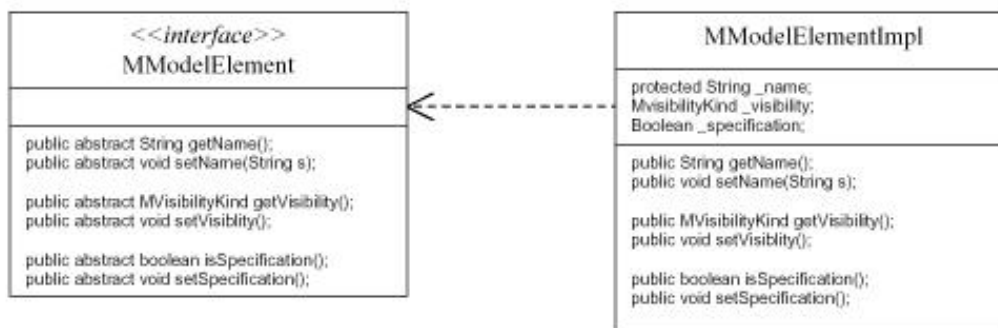
Depending on the type feature, not all reflective methods may be valid:

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`.

## Rappresentazione nel metamodello fisico UML



## Rappresentazione nel modello di NSUML



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

<code>Object reflectiveGetValue(String, int)</code>	only lists
<code>void reflectiveSetValue(String, int, Object)</code>	only lists
<code>void reflectiveAddValue(String, int, Object)</code>	only lists
<code>void reflectiveRemoveValue(String, int, Object)</code>	only lists

## 6.2.10 Other libraries

### OCL

OCL is handled by Frank Finger's OCL compiler, developed at the TU Dresden.

The purpose of the compiler is to generate code out of OCL expressions. The generated code evaluates the OCL constraint for a model instance, i.e. a program or data base, at runtime. Constraints formulated during analysis and design can then easily be checked in the implementation.

While the Java code generator that is being implemented makes certain restrictions to the OCL constructs that can be used, the other compiler modules are implemented as close to the OCL specification of UML 1.3 as possible.

## 7. Code generation

In this chapter we will describes the classes added to ArgoUML core for implementing the code generation feature.

The main work is due to the classs `GeneratorState`, that implements all the functions for generating code from the statechart.

### 7.1. Implementation

Code generation in Argo/UML is supported with a language independent abstract base class and Java-specific subclasses.

#### 7.1.1 Classe Generator

Class `Generator` is an abstract base class that is similar to the Visitor design pattern (Gamma et al., 1995). However, the logic to traverse the design representation is intermixed with node processing logic.

This class is the abstract super class that defines a code generation framework. It is basically a depth-first traversal of the UML model that generates strings as it goes. This framework should probably be redesigned to separate the traversal logic from the generation logic. See the Vistor design pattern in "Design Patterns", and the Demeter project.

Code generation strting from menu options need standard llibraries ( `javax.swing`), meanwhile the selected classifier is returned by the interface `org.argouml.generator.ui.TableModelClassChecks`.

### **7.1.2 Classe GeneratorJava**

Class `GeneratorJava` is a Java-specific subclass that generates Java source files. It is called by `ClassGeneratorDialog`.

Each of the Java-specific classes implements methods that generate source code for design elements of a given type. Since the code generation logic is coded in Java, the only way to customize it is by changing the code or by adding new code generation preferences.

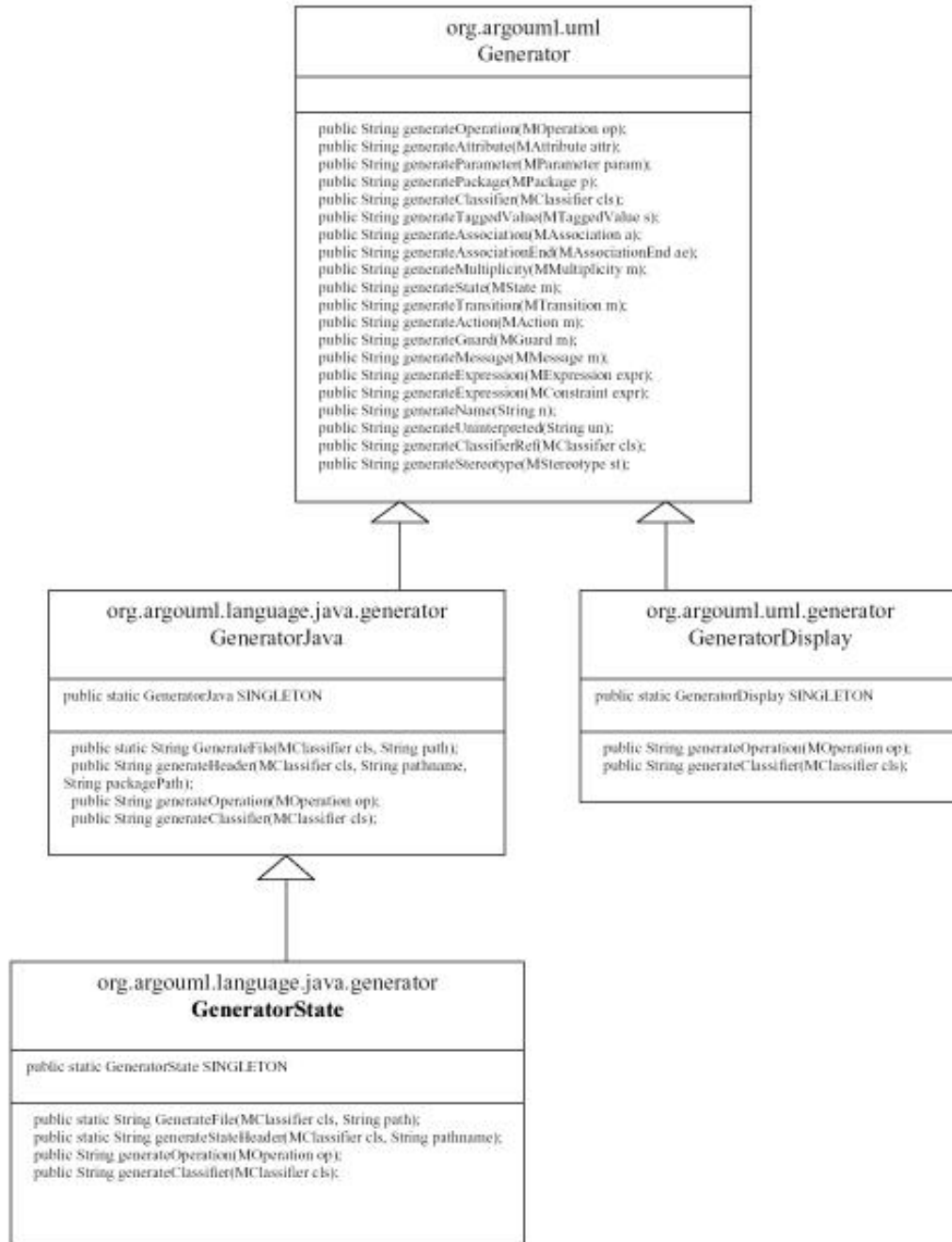
### **7.1.3 GeneratorDisplay**

Class `GeneratorDisplay` generates simplified Java code to be displayed in the "Source" tab and in the textual labels of UML class icons and other parts of UML diagrams.

### **7.1.4 GeneratorState**

I add a new class `GeneratorState` inherited from `GeneratorJava`.

This class is called by many classes.



For storing the main features of the state machine we created these variables:

```

static MStateMachine sm = null;
static Vector vstates = new Vector();
static Vector vevents = new Vector();
static Vector vactions = new Vector();
static Vector vtransitions = new Vector();
static Vector vguards = new Vector();
  
```

```
String dirpath = path + cls.getName() + "_state";
```

Each vector stores operations; more, we have variable to store the dirpath when we create the new classes of the structure.

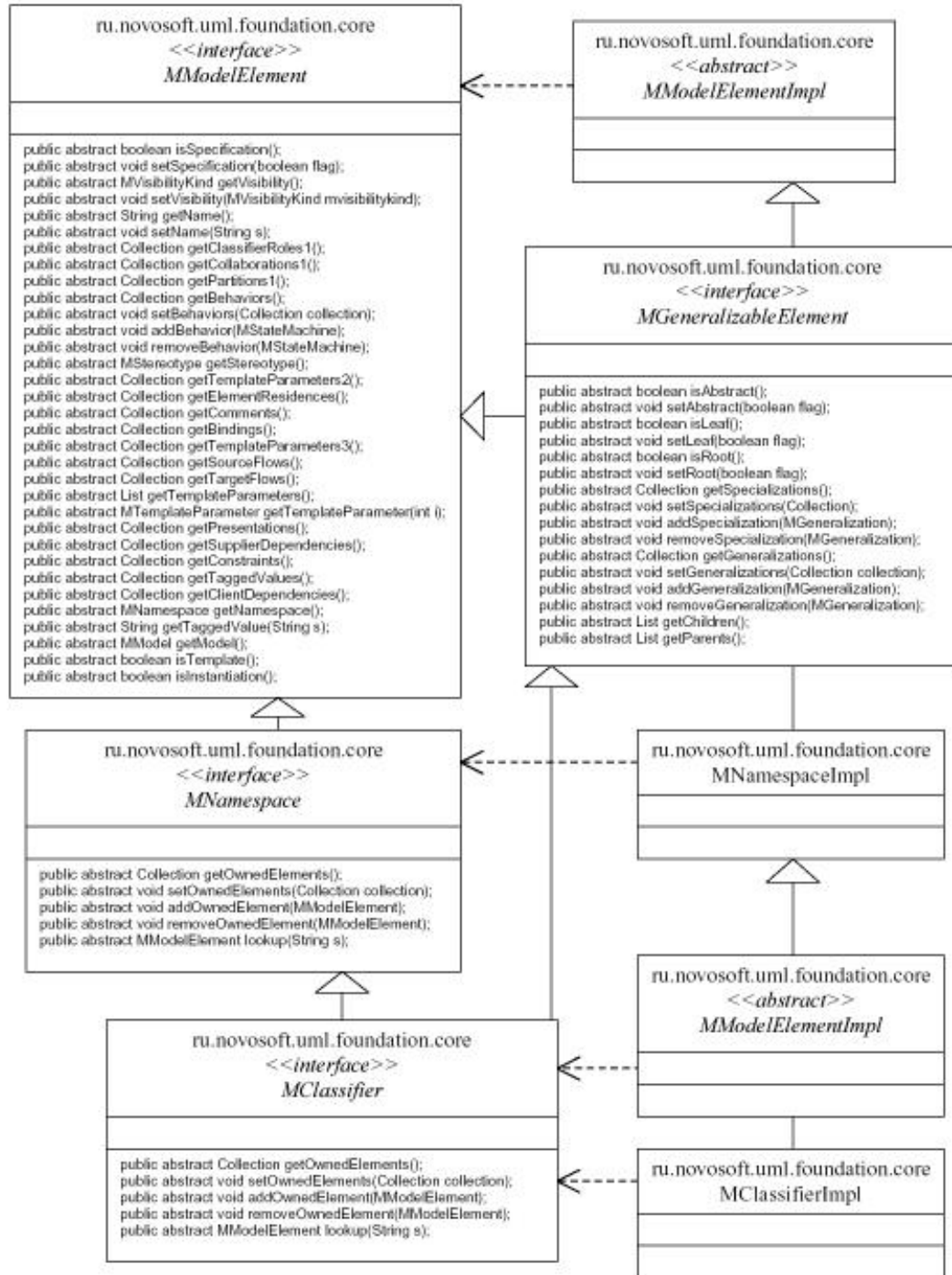
Main operations are:

- `String Generate(Object o)`
- `String GenerateFile(MClassifier cls, String path)`
- `String GenerateFSMFile(MClassifier cls, String path)`
- `boolean manageStateMachine(MClassifier cls)`
- `String generateHeader(MClassifier cls, String path)`
- `String generateClassifier(MClassifier cls)`
- `String generateStateHeader(String path)`
- `void findSMDData(String commonHeader, String path)`
- `void generateContext(MClassifier cls, String stateHeader, MClassifier srcClassifier, String path)`
- `void generateStateAbs(MClassifier cls, String stateHeader, String path)`
- `void generateStates(MClassifier cls, String stateHeader, String path)`

Fra i parametri gestiti da queste funzioni, `cls` rappresenta la classe attuale e il

## 7.1.5 Class properties

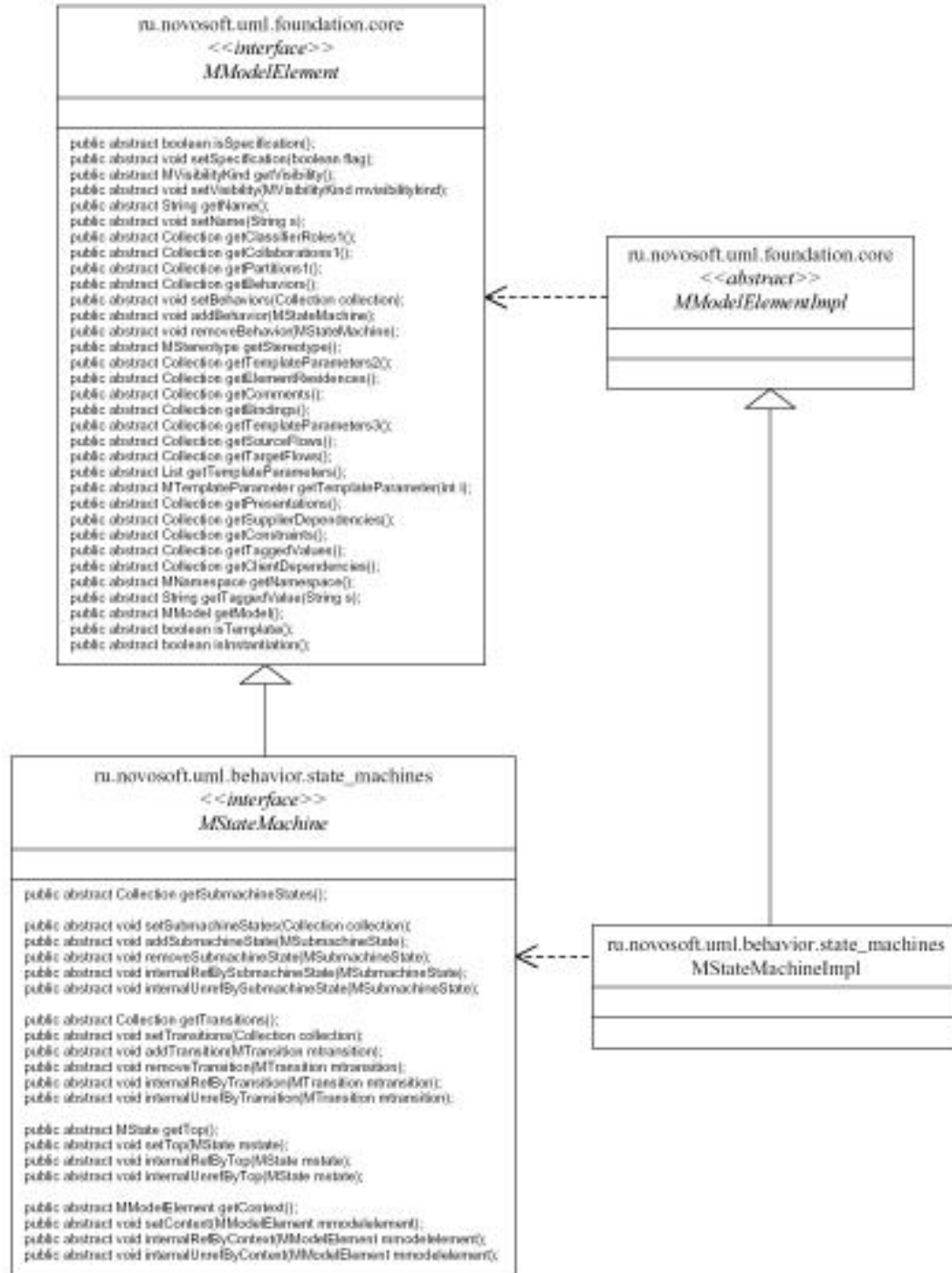
Then for classes and classifiers we refers to this hierarchy.





## 7.1.6 State machine properties

For state machines we refers to next class diagram.



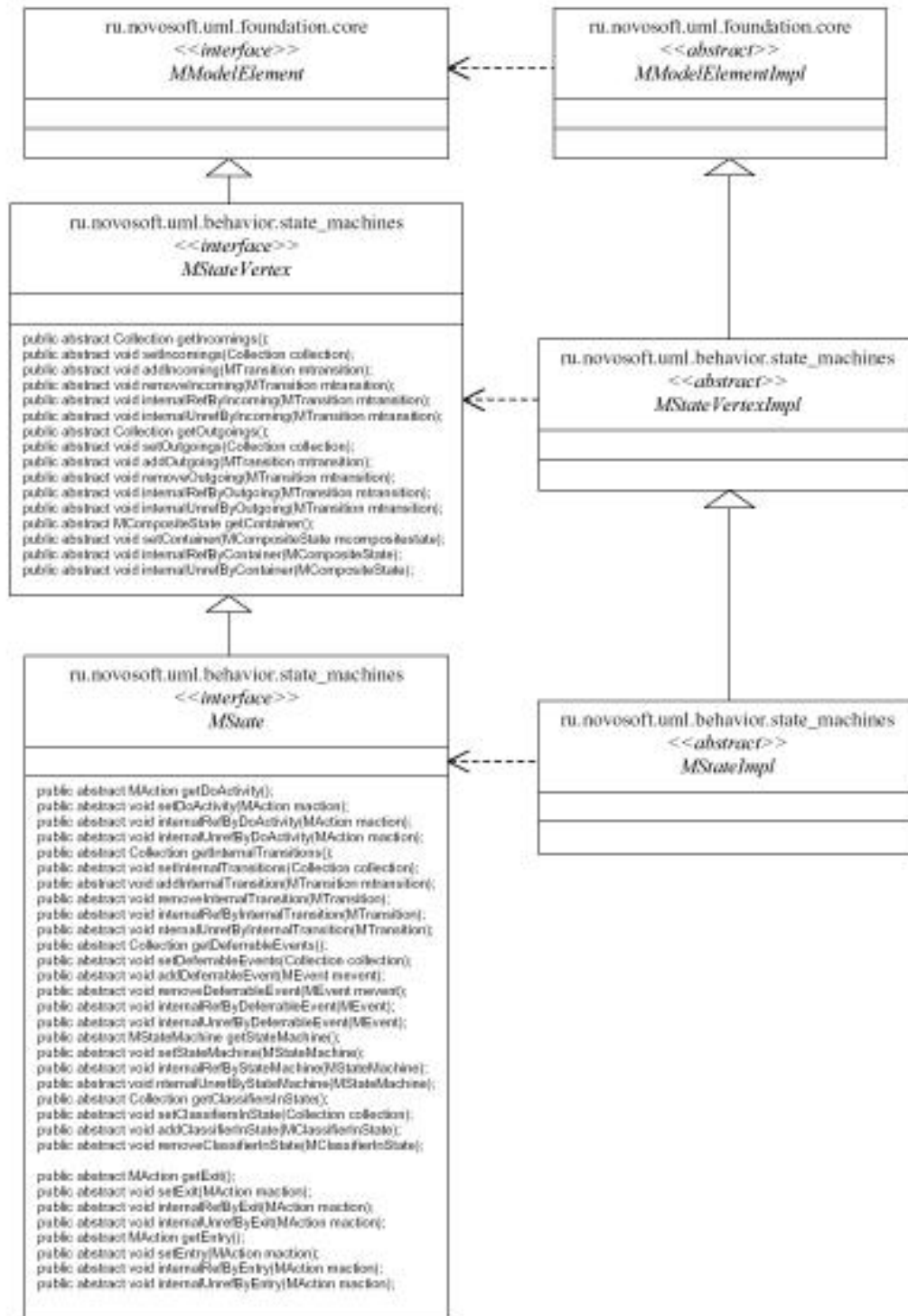
In GeneratorState I add some code for finding the statemachines associated to a class:

```
Collection behaviors = cls.getBehaviors();
Iterator it = behaviors.iterator();
sm = null;
while (it.hasNext())
{
    Object behavior = it.next();
    if ( behavior instanceof MStateMachine)
    {
        sm = (MStateMachine)behavior;
        break;
    }
}
if (sm instanceof MStateMachine)
    trovataSM = true;
```

So we can load state machine's components and generate the appropriate code.

All attributes of classifier is referred by MMUtil class.

For the states' properties we refer to Mstate class.



```

MState top = sm.getTop();
Collection states = (MCompositeState)top).getSubvertices();
Iterator itstate = states.iterator();
while (itstate.hasNext())
{

```

```

        Object obj = itstate.next();
        if (obj instanceof MState)
        {
            // Memorizzo gli stati
            vstates.addElement((MState)obj);
        }
    }
    Collection trans = sm.getTransitions();
    Iterator ittrans = trans.iterator();
    while (ittrans.hasNext())
    {
        //Memorizzo le transizioni
        MTransition t = (MTransition)ittrans.next();
        vtransitions.addElement(t);
        //Memorizzo gli eventi/trigger
        MEvent e = t.getTrigger();
        ...
        vevents.addElement(e);
        //Memorizzo le azioni/effects
        MAction a = t.getEffect();
        ...
        vactions.addElement(a);
        //Memorizzo le guardie
        MGuard g = t.getGuard();
        ...
        vguards.addElement(g);
    }
}

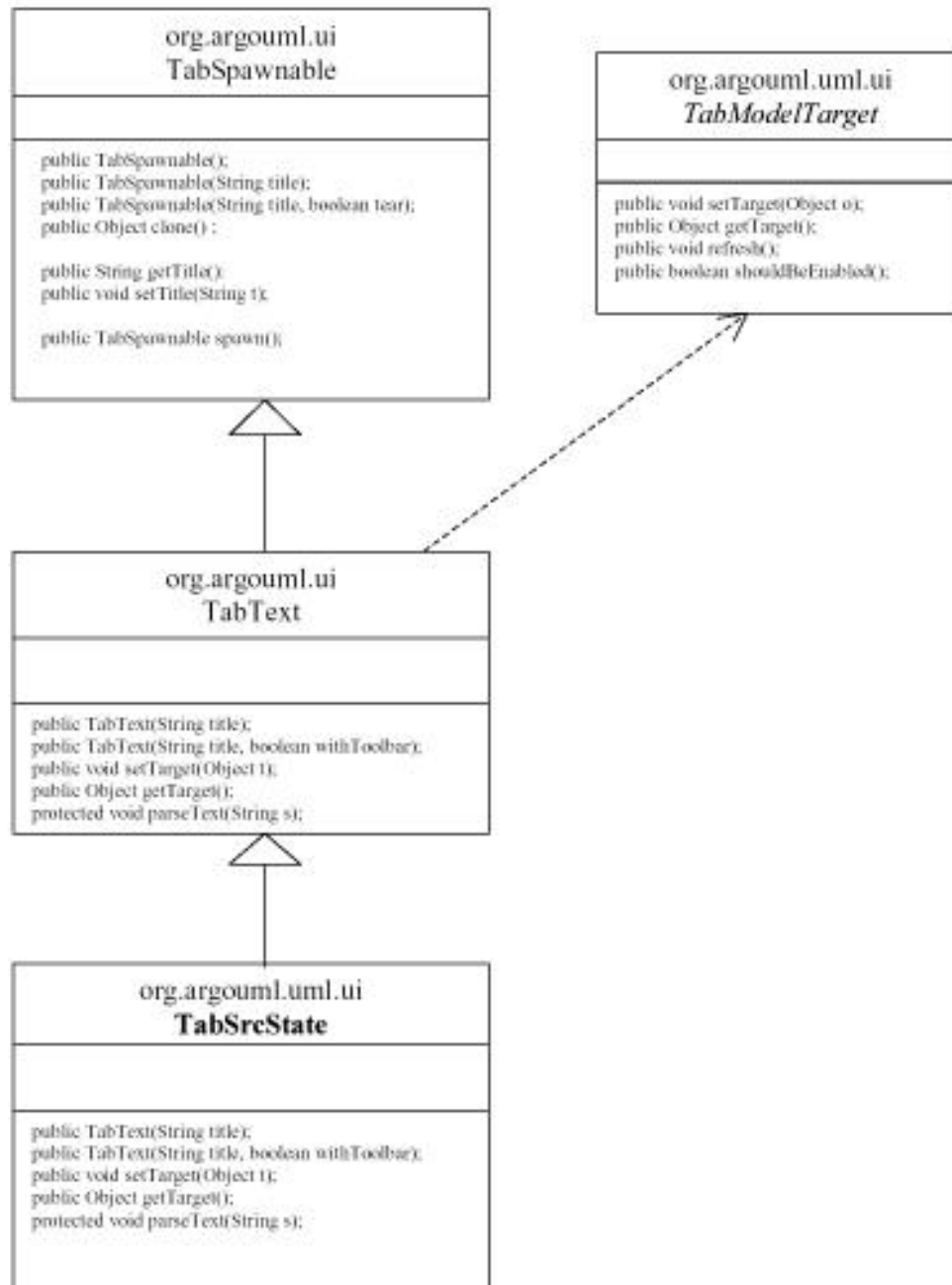
```

## 7.2. Generated code rendering

A subclass of `JPanel` that can act as a tab in the `DetailsPane` or `MultiEditorPane`.

When the tab is double-clicked, this `JPanel` will generate a separate window of the same size and with the same contents. This is almost like "tearing off" a tab.

Before viewing code we must select a class. This is done by `TabModelTarget`.



## 7.2.1 Classe TabModelTarget

```

package org.argouml.uuml.ui;

public interface TabModelTarget {
    public void setTarget(Object o);
    public Object getTarget();
}
  
```

```

    public void refresh();
    public boolean shouldBeEnabled();
}

```

This interface permits to select by mouse a class to traduce and then to obtain the code generated as a returned value.

So I start creating a new class called TabSrcState in this manner:

## 7.2.2 Classe TabText

The inherited class TabText permits to generate code from a selected object ( \_target) to a string ( \_text) that must be shown by tab strips.

Is the genText method that must be redefined to make a particular code generation code generation.

This class contains the code generated from the class Generator.

## 7.2.3 Classe TabSrcState

### ***Adding a tab in the Details Panel***

I created my TabSrcState class in org.argouml.uml.ui by copying from another TabYYY.java. Then I registered my TabSrcState in org/argouml/argo.ini by adding a line like:

```

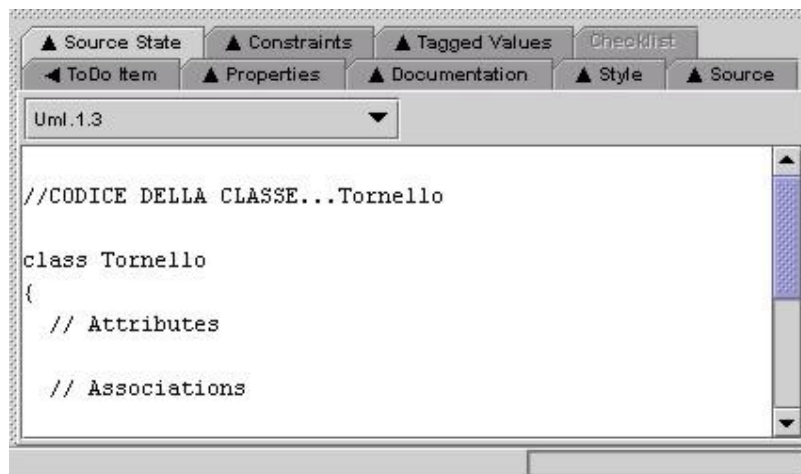
details:      TabSrcState

```

Immediate rendering of generated code needs GEF library. The Editor class is the first to be called: it allows to select method for processing events from ArgoUML's user. These events are propagate until Modes class that permits to find the code out of the libraries.

Referred code is in `org.argouml.uml.diagram.ui.TabDiagram`, in `selectionCahnged()` method.

```
protected String genText()
{
    Object modelObject = _target;
    if (_target instanceof FigNode)
        modelObject = ((FigNode)_target).getOwner();
    if (_target instanceof FigEdge)
        modelObject = ((FigEdge)_target).getOwner();
    if (modelObject == null) return null;
    return GeneratorState.Generate(modelObject);
}
```



Rendering fo generated code is created by calling GEF libraries: Editor class and the Modes class.

In particoular we call the class `TabDiagram`. By the method `selectionChanged()`, we pass the selected fig to `setDetailTarget()`, in `ProjectBrowser` class.

```
public class TabDiagram extends TabSpawnable
{
    public void selectionChanged(GraphSelectionEvent gse)
    {
        Vector sels = gse.getSelections();
        ProjectBrowser pb = ProjectBrowser.TheInstance;
        if (sels.size() == 1)
            pb.setDetailsTarget(sels.elementAt(0));
    }
}
```

```

        else
            pb.setDetailsTarget(null);
    }
}

```

The automatic load of the tabstrip is done by `org.argouml.util.ConfigLoader`.

## 7.3. Show generate code in “Source state” tab strip

We show you how code generated is shown in the “Source state” tab strip.

User clicks on Source state tab, then we have a call to `setTarget(Object)`:

```

public void setTarget(Object t) {
    //chiamata a setTarget di TabText (il padre)
    super.setTarget(t);
}

```

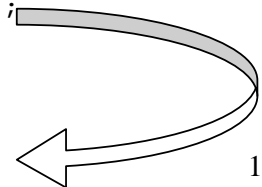
Then from TabText we have:

```

public void setTarget(Object t) {
    _target = t;
    _parseChanges = false;
    if (_target == null) {
        _text.setEnabled(false);
        _text.setText("Nothing selected");
        _shouldBeEnabled = false;
    }
    else {
        _text.setEnabled(true);
        String generatedText = genText();
    }
}

```

Then we return in TabSrcState calling `generateState`:





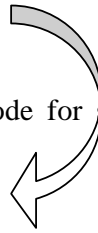
```

protected String genText() {
    Object modelObject = _target;
    if (_target instanceof FigNode)
        modelObject = ((FigNode)_target).getOwner();
    if (_target instanceof FigEdge)
        modelObject = ((FigEdge)_target).getOwner();
    if (modelObject == null) return null;

    return GeneratorState.Generate(modelObject)
}

```

This return is a call to `GenerateState`, class in which I create the code for state machine. This return a string to `TabSrcState`.



```

public static String Generate(Object o) {
    return SINGLETON.generate(o);
}

```

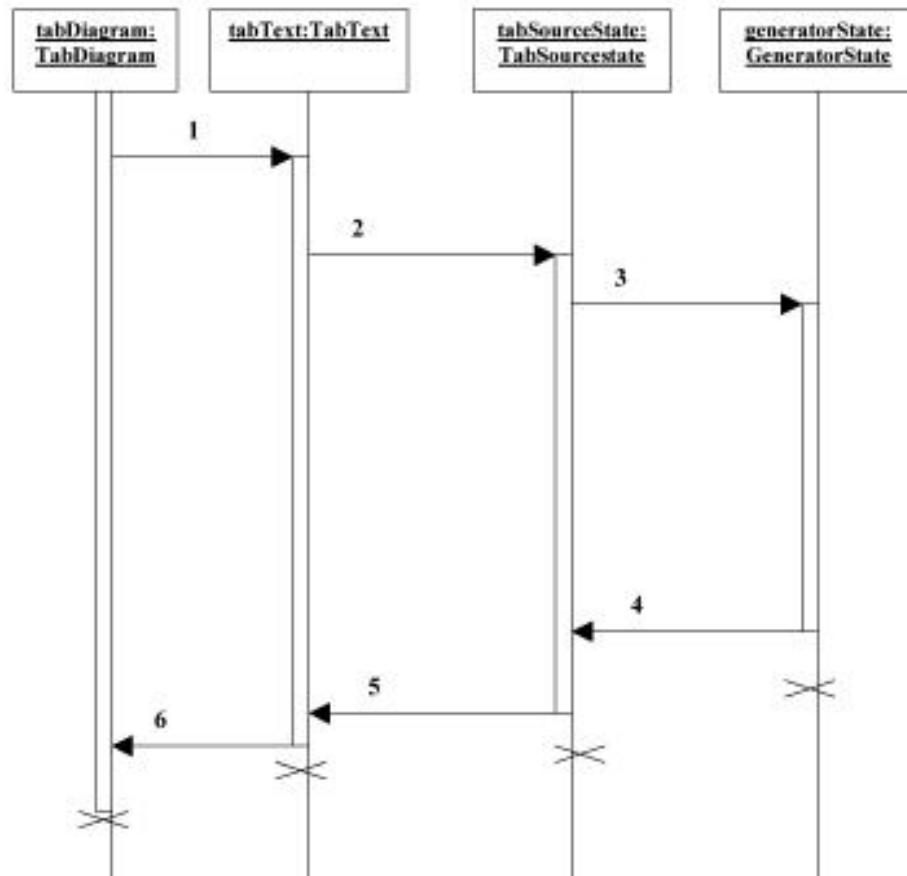
So in `TabSrcState` we return a string to `TabText`, which is calling, and it can show the generated text.

```

//public class TabSrcState
protected String genText() {
    ...
    return GeneratorState.Generate(modelObject);
}

//public class TabText
public void setTarget(Object t) {
    ...
    String generatedText = genText();
    if (generatedText != null) {
        _text.setText(generatedText);
    }
}

```



## 7.4. New menu: *Generation statecharts*

The new feature of code generation is added to ArgoUML by new menu called *Generation statecharts*, under the existing *Generation* menu.

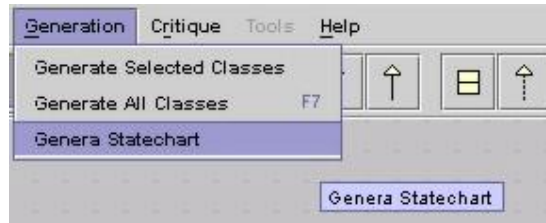
### 7.4.1 Class ProjectBrowser

#### *Class ProjectBrowser*

This class contains all declarations and definition of all menus in the interface. One particular menu is Generation( with two options: Generate selected classes and Generate All Classes).

This menu has three options:

- *Generate selected classes* – for generating java code from the selected class
- *Generate All classes* – for generating java code from all classes in the diagram
- *Generate statechart* – for generating java code from the statechart regarding the selected class

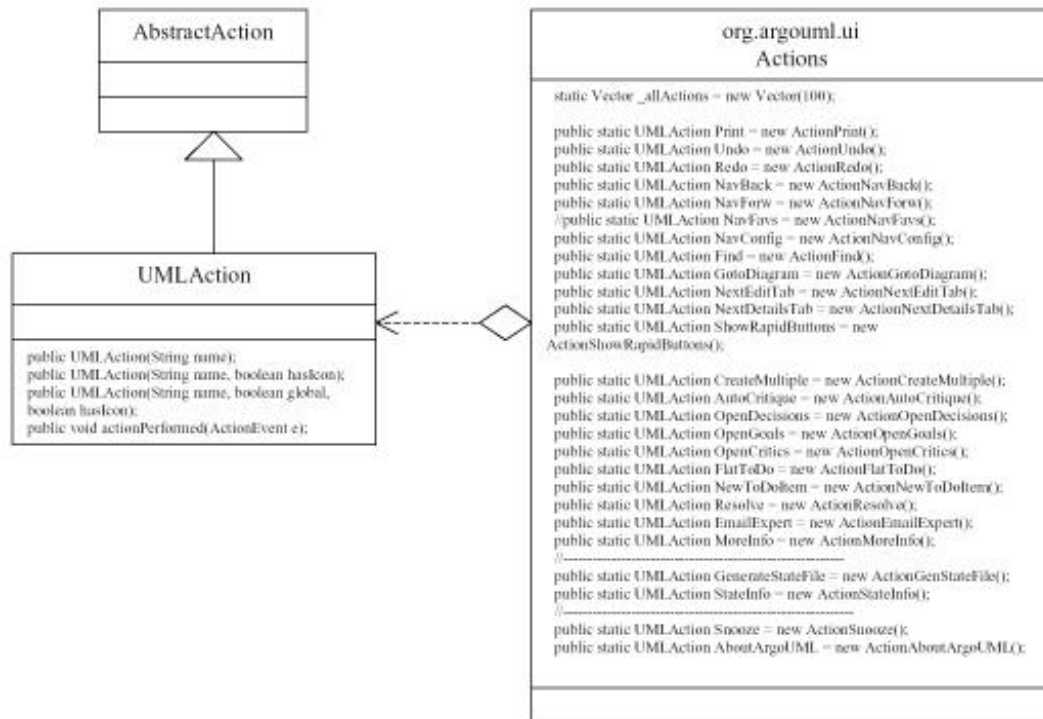


Menu creation consists in creating an `javax.swing.JMenu` element to which add options of Action class.

```
JMenu generate =
    (JMenu)_menuBar.add(new Menu(menuLocalize("Generation")));
setMnemonic(generate,"Generate",'G');
generate.addActionGenerateOne.SINGLETON);
JMenuItem genAllItem =
    generate.addActionGenerateAll.SINGLETON);
setAccelerator(genAllItem,F7);

//Genera file con il codice
JMenuItem genStatecharts =
    generate.addAction(Actions.GenerateStateFile);
```

`_menuBar` variable contains the menu. New menu `genStatecharts` calls the operation of the Action class.



## 7.4.2 Class UMLAction

These classes are important for performing action in the menu Generate.

## 7.4.3 Class Actions

This class contains all declarations of all actions invoked by methods and their management by `actionPerformed()` methods.

Actions are defined as extensions of `UMLAction` class.

I add the code for new actions, about generating code from state machines:

```

public static UMLAction GenerateStateFile =
    new
    ActionGenStateFile();
  
```

```

        public static UMLAction
        StateInfo = new
        ActionStateInfo();

```

## 7.4.4 Class ActionGenStateFile

I structured my classes in the same way as the menu Generate Selected classes calls the classesGenerationDialog,.

Selecting the new menu, a dialog window opens and here we find options implemented by ClassStateGenerationDialog class. This class is declared under Actions class and it implements code generation.

Calling the classStateGenerationDialog() class, we pass the management of code generation to this dialog window.

```

class ActionGenStateFile extends UMLAction
{
    public ActionGenStateFile()
    {
        super("Genera File con il codice",NO_ICON);
    }

    public void actionPerformed
        (java.awt.event.ActionEvent actionEvent)
    {
        ProjectBrowser pb = ProjectBrowser.TheInstance;
        Editor ce =
            org.tigris.gef.base.Globals.curEditor();
        Vector sels = ce.getSelectionManager().getFigs();
        java.util.Enumeration enum = sels.elements();
        Vector classes = new Vector();
        while (enum.hasMoreElements())
        {
            Fig f = (Fig) enum.nextElement();
            Object owner = f.getOwner();
            if (!(owner instanceof MClass) &&
                !(owner instanceof MInterface))
                continue;
            MClassifier cls = (MClassifier) owner;
            String name = cls.getName();
            if (name==null || name.length()==0)
                continue;

```

```

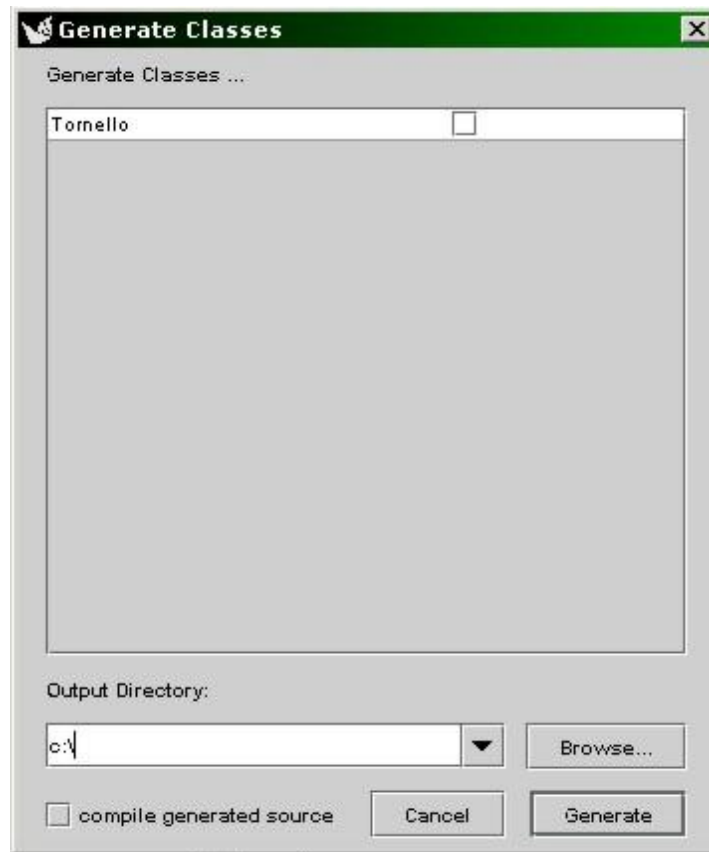
        classes.addElement(cls);
    }
    ClassStateGenerationDialog cgd =
        new ClassStateGenerationDialog(classes);
    cgd.show();
}

public boolean shouldBeEnabled()
{
    return true;
}
}

```

### 7.4.5 Class ClassStateGenerationDialog

As all the menu for generating code, the mine is the same: a dialog window is opened and the user have to elect the classes to implement.



### 7.4.6 Show the call stack from the menu

So the call stack from menu generates a file with the source code shown in TabSrcState.

From the menu *Generation statecharts* we call Action class:

```
class ActionGenStateFile extends UMLAction
{
    public void actionPerformed(ae)
    {
        Vector sels =
            ce.getSelectionManager().getFigs();
        ClassStateGenerationDialog cgd =
            new ClassStateGenerationDialog(classes);
        cgd.show();
    }
}
```

1. Then a dialog window is opened

```

public class ClassGenerationDialog
{
    protected JButton _generateButton =
        new JButton("Generate");

    public ClassGenerationDialog(Vector nodes);

    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == _generateButton)
        {
            String path = ((String)_dir.getModel()
                           .getSelectedItem()).trim();
            ProjectBrowser pb =
                ProjectBrowser.TheInstance;
            Project p = pb.getProject();
            p.getGenerationPrefs().setOutputDir(path);

            Vector nodes = _tableModel.getChecked();
            int size = nodes.size();
            String[] compileCmd=new String[size+1];
            for (int i = 0; i <size; i++)
            {
                Object node = nodes.elementAt(i);
                if (node instanceof MClassifier)
                    compileCmd[i+1] =
                        GeneratorState.GenerateFile(
                            (MClassifier) node, path);
            }
        }
    }
}

```

2. then, after click the generate button, we call the operation GenerateFile in

GenerateState class:

```

String GenerateFSMFile(MClassifier cls, String path)
{
    String nomefile = cls.getName()+ ".java";
    String pathname = path + nomefile;
    String dirpath = path +cls.getName() +"_state";
    trovataSM = manageStateMachine (cls);
    String commonHeader =
        SINGLETON.generateStateHeader(cls);
    String srcClassifier =
        SINGLETON.generateClassifier(cls);
    if (trovataSM)
    {
        File cartella = new File(dirpath);
        cartella.mkdir();
        String nomedir=

```



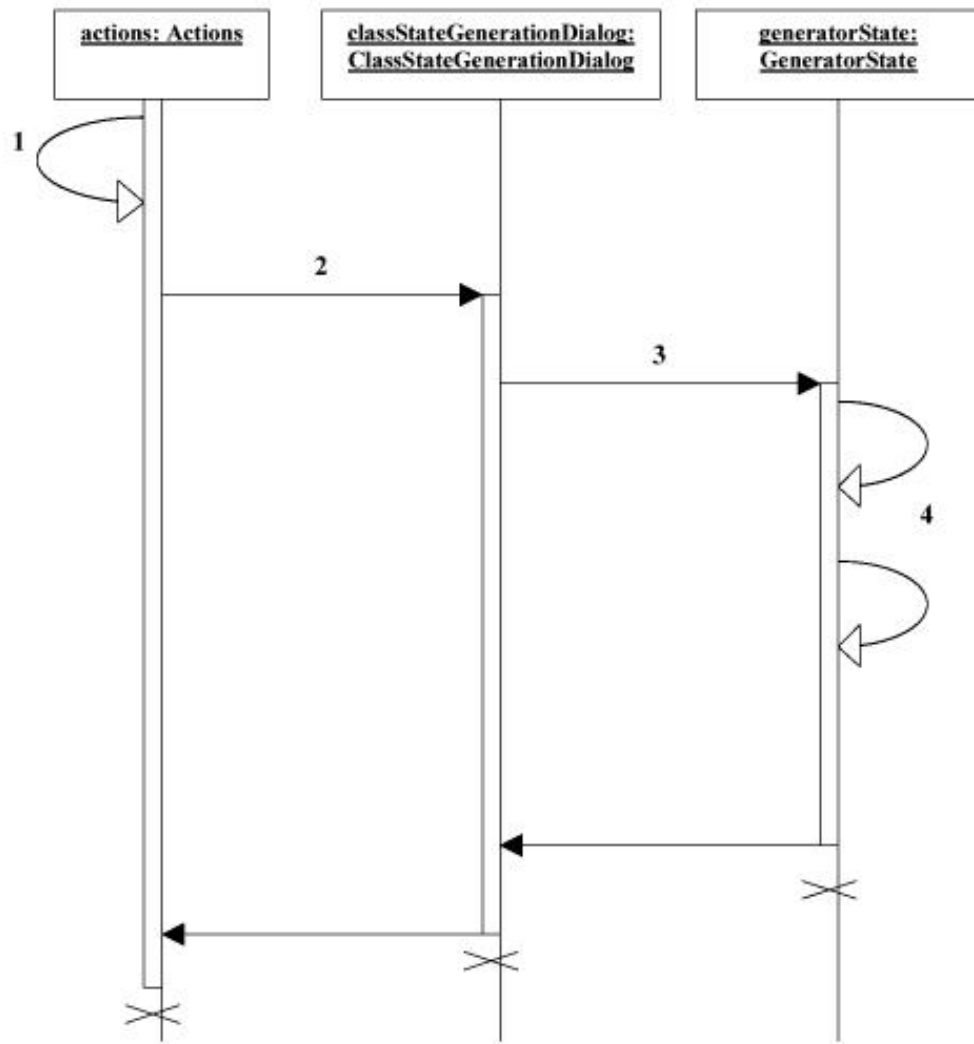
```

        cartella.getPath()+FILE_SEPARATOR;
String stateHeader =
    SINGLETON.generateStateHeader(nomedir);
SINGLETON.findSMDData(commonHeader, nomedir);
SINGLETON.generateContext
    (cls, stateHeader, srcClassifier, nomedir);
SINGLETON.generateStateAbs
    (cls, stateHeader, nomedir);
SINGLETON.generateStates
    (cls, stateHeader, nomedir);
}
File f = new File(pathname);
BufferedWriter fos = null;
try
{
    fos = new BufferedWriter(new FileWriter(f));
    fos.write(commonHeader);
    fos.write(srcClassifier);
}
catch (IOException exp) { }
finally
{
    try { if (fos != null) fos.close(); }
    catch (IOException exp)
    {
        System.out.println("FAILED:"+f.getPath());
    }
    System.out.println("-end generating all file -");
    return pathname;
}

```

3. The two functions, `generateStateHeader` and `generateClassifier`, creates the source code starting from the class of the model, according to the statechart defined. Initially, the state machine is read by the `manageStateMachine` operation, that store in vector variables the main features of the state machine, and finally it return a boolean `trovataSM`. Then we call operation that creates the files requested by State pattern:

- `generateContext` - generates the class “context”
- `generateStateAbs` - generate the class “abstract state”
- `generateStates` - generate the classes “concrete states”



## 8. Conclusions

This work add to ArgoUML the feature of code generation starting from statechart in the model. Code generated automatically control class behaviour according to the statechart.

This statechart have to be defined in UML and it have to have no composite states.

The work is totally scalable and reusable: both the structures of code generated and the code added to ArgoUML kernel. This last structure is modular.

From this work is possible generate a new release of ArgoUML or a plug-in with only the new features.

## 9. Thanks

I would thanks alla member of *dev-argouml* mailing list,about developing ArgoUML, that helped me much to learn the kernel architecture and the main mechanisms for generating realeases and plug-ins.I would thaks expecially ir. J.F.J. Branderhorst that helped me to developing a plug-in with my new feature.



## Bibliography

- [1] Armstrong Eric, *How to implement state-dependent behavior - Doing the State pattern in Java* (<http://www.javaworld.com>)
- [2] De Lamotte Florent, *Présentation d'ArgoUML*,  
([http://n00n.free.fr/argo\\_presentation/Presentation.html](http://n00n.free.fr/argo_presentation/Presentation.html))
- [3] Design Patterns in Java - Reference and Example Site  
(<http://www.fluffycat.com/java/patterns.html>)
- [4] Domenici Andrea, *Dispensa di Ingegneria del Software*
- [5] Domenici Andrea, Frosini Graziano, *Introduzione alla programmazione ed elementi di strutture dati con il linguaggio C++*, FrancoAngeli, 1996
- [6] Gamma Eric, Helm Richard, Johnson Ralph, and Vlissides John, *Design Patterns*, Addison-Wesley, 1995
- [7] GEF - The Java Graph Editing Framework (<http://gef.tigris.org>)
- [8] Gomaa Hassan, *Designing concurrent, distributed, and real-time applications with UML*, Addison-Wesley 2000
- [9] Holub Allen, *OODesign Workshop - A UML Reference*  
([http://www.holub.com/cat/oo\\_design\\_workshop.html](http://www.holub.com/cat/oo_design_workshop.html))
- [10] Lindholm Tim, Yellin Frank. *The Java Virtual Machine specification*, Addison-Wesley, 1999 (<http://java.sun.com/docs/books/vmspec>)

- [11] Martin Robert C., *UML Tutorial: Finite State Machines*, Engineering Notebook Column, C++ Report, June 98.
- [12] NSUML - The Novosoft UML API ([http:// nsuml.sourceforge.net](http://nsuml.sourceforge.net))
- [13] Object Management Group, *OMG Unified Modeling Language Specification* v.1.3 e v.1.4 (<http://www.omg.org>)
- [14] Ramirez Alejandro, Vanpeperstraete Philippe, Rueckert Andreas, Odutola Kunle e Bennett Jeremy, *ArgoUML User Manual - A tutorial and reference description of ArgoUML*, (<http://argouml.tigris.org>)
- [15] Robbins Jason, Redmiles David, Tolke Linus e altri, *The ArgoUML case tool Project* (<http://argouml.tigris.org>)
- [16] Skinner Martin, *Enhancing an Open Source UML Editor by Context-Based Constraints for Components*, University of Berlin, Thesis, December 2001
- [17] Sun Microsystems, *Forte for Java CE Guide and Tutorial*, August 2001 (<http://www.sun.com/forte/ffj/overview>)
- [18] Tarquini Massimiliano, *Java mattone dopo mattone* (<http://www.java-net.tv>)
- [19] TogetherSoft, Inc., *Practical UML - A Hands-On Introduction for Developers* Copyright © 2001 (<http://www.toghetersoft.com>)
- [20] Tolke Linus, Klink Markus, *Cookbook for developers of ArgoUML* (<http://argouml.tigris.org>)