ATLAS muon reconstruction from a C++ perspective: a road to the Higgs
Hendriks, P.J.

Citation for published version (APA):
Hendriks, P. J. (2000). ATLAS muon reconstruction from a C++ perspective: a road to the Higgs

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
A.1 Unified Modelling Language

To communicate efficiently one needs a common language and software design is no exception to that rule. From the late 1980's onwards many object-oriented analysis and design methods were introduced for exactly that reason, but sadly their proliferation negated their usefulness. Fortunately, out of this wave of methods a recent standard has arisen: the Unified Modelling Language or UML [64]. It defines a set of diagrams each representing a different part of, or view on the analysis or design. Only three of these diagrams are used in this thesis, viz. the package, class and interaction diagrams. Their basic syntax is explained in the next sections, but first some general concepts are presented.

![Diagram](image)

**Figure A.1** General-purpose concepts in the UML.

An entity such as a package or a class is usually represented by a box of some sort, containing its name and an optional stereotype. A stereotype is a general attribute either defined by the UML or by the user. Examples of attributes used in this thesis are `<interface>` (all methods of the class are abstract), `<abstract>` (some of the methods are abstract) and `<external>`.

---

1. A method consists, in principle, of both a modelling language and a software process. The language is the (mainly graphical) notation, while the process is its advice on what steps to take in creating a design (see e.g. [24]).
A dependency between two entities is always represented by an arrow. The single-sided arrow shown in figure A.1 depicts a unidirectional relationship, but bidirectional ones are also possible. When a dependency line is crossed by a slash ('/') it means that the connection is a derived one, i.e. it depicts a relationship existing somewhere else, e.g. between two base classes.

### A.1.1 Package Diagrams

A package diagram is a high-level diagram showing groups of classes (the packages) and the dependencies among them. A package can be opaque, meaning that its internals are not visible from the outside, or it can be transparent. In the latter case, nested packages or even individual classes can be shown on the diagram. Dependencies can then not only point to the package itself, but also to an entity within it.

![Package Diagram Syntax](image)

**Figure A.2** Package diagram syntax.

One of the possible stereotypes of a package is `<<global>>`, which means that all other packages (can) reference it.

### A.1.2 Class Diagrams

A class diagram can be used to show the static structure of either concepts, types or classes:

- In its conceptual view the diagram can be used to depict the way users think about the world, but also to show an overall, high level view of the design;
- In its interface view it shows the interfaces of the software components;
- And finally in its implementation view the actual classes are displayed.

In all three cases the syntax is identical, which means that each diagram must be accompanied by an explicit statement of its type. However, unless otherwise noted, all class diagrams in this thesis depict an implementation view.

---

2. This subdivision is not explicitly defined by the UML.
The basic syntax of a class diagram is shown in figure A.3. In it a class is depicted by a rectangular box that may contain in addition to its name and stereotype a listing of attributes and operations. Each of these is preceded by an access specifier: a ‘+’ means public, i.e. part of the class’ interface, while a ‘-’ denotes a private, and a ‘#’ a protected member. When a method is printed in italics it is abstract, which means that no implementation is provided by the class itself, but that instead its derived classes must supply one. When the name of the class is italicized, it is an interface, meaning all its methods are abstract.

Specialization or inheritance is depicted by an arrow pointing from the derived class to its super- or base class. A hollow arrow means an exclusive inheritance, while a filled arrow depicts an inclusive- or virtual-inheritance relationship. When the same base class appears multiple times in a class hierarchy and the inheritance is inclusive, all these base instances are merged into one. On the other hand, when the inheritance is exclusive, each appearance of the base object generates its own methods and attributes.

In figure A.4 the syntax for the associations between classes, interfaces and types is presented. The three different association types are:

**Basic association**

The semantic relationship between two or more entities that involves connections among their instances.

---

3. When a member is protected it can, in addition to the class to which it belongs, only be seen by instances of derived classes.
Association (bidirectional):

\[
\begin{array}{c}
\text{Class A} \quad \text{role B} \\
\downarrow \quad \downarrow \\
\text{Class B} \quad \text{role A}
\end{array}
\]

Multiplicity:

\[
\begin{array}{c}
1 \quad \text{Class} \\
\downarrow \\
\text{exactly one}
\end{array}
\]

Aggregation:

\[
\begin{array}{c}
\text{Class A} \\
\downarrow \\
\text{Class B}
\end{array}
\]

Composition:

\[
\begin{array}{c}
\text{Class A} \\
\downarrow \\
\text{Class B}
\end{array}
\]

Figure A.4 Class diagram associations syntax.

Aggregation

A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.

Composition

A form of aggregation with strong ownership, and coincident lifetimes of the whole and the parts. Parts with non-fixed multiplicity may be created after the composite itself, but once created they share its lifetime.

As can be seen from these definitions, the three types of relationships form a continuous range, and the distinction between them is not always clear.

A.1.3 Interaction Diagrams

Interaction diagrams show how several objects interact with each other. The UML defines three different types of interaction diagrams, viz. sequence, collaboration and activity diagrams. They basically show the same information, but in a different format. Of the three, only the sequence diagrams are used in this thesis (see figure A.5).

Each vertical line represents the life span of an object with time flowing from top to bottom. The object’s identifier consists of two parts, separated by a colon. The first part represents its name or identity, while the second part denotes the name of the class of which the object is an instance. When the object’s name is omitted, the box represents an anonymous object, i.e. one whose identity is not relevant to the scope of the interaction diagram.

A message is represented by an arrow between the lifelines of two objects. The order in which these messages occur is shown from top to bottom. A message is always labelled with the message name, but the message’s arguments and some control information can also be included.
The first type of control information used in this thesis is the \textit{condition}, which indicates when a message is sent: A message is only dispatched when the condition evaluates to true. The second useful type of control info is the \textit{iteration marker}, which shows that a message is sent many times to one or multiple receiver objects. Its syntax is \texttt{*[type of iteration]}.

Finally, the end of a method can be shown as a dashed arrow, possibly accompanied by a return value.

\section*{A.2 Dataview Diagrams}

In addition to the diagrams defined by the UML, component diagrams are used to represent dataviews and their interactions. A component is defined as a software entity, which completely decouples its interface from its implementation (see also section 3.4), and a dataview fulfils that requirement: It can be perceived as a back box, completely identified by its inputs and outputs.

A dataview is depicted by a box containing its name and optionally its type. Its inputs are arranged on the left and top edges of the box, its outputs on the right and bottom ones. A

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{dataview.png}
\caption{Dataview diagram syntax [65].}
\end{figure}
connection between two dataviews is represented by a line, with the data flowing from the output of the first dataview to the input of the second one. The type of the transferred data can be listed alongside the line. Commands to change the state of a dataview flow in the direction opposite the data, i.e. from an input to an output.

By default, each connection is independent, which means that multiple links to the same output retrieve their value and change their state independent of each other. Only when two or more connections share a common line, does the connected output (and its dataview) have the same state.