Supplementary Notes

FuGE UML Primer

The Unified Modeling Language (UML) from the OMG (http://www.omg.org/) is a specification defining a graphical language for visualizing, specifying, constructing, and documenting the artifacts of object-based software, from simple applications to enterprise-level distributed computing systems. For the purposes of easy interpretability, FuGE uses a much-restricted subset of the UML standard for specifying the data model. Specifically, the model utilizes only one type of UML diagram, the Class Diagram, which represents a static view of the data structures. This restriction is enforced because the primary purpose of the model is to design an exchange format and facilitate the storage of the data (i.e. database design). Class diagrams can also be mapped relatively easily to other platform-specific implementations, such as basic toolkits in Java, Perl etc. Other features of UML, such as state, activity and interaction diagrams are used for developing complete software systems, which is not part of the scope of the FuGE project, and they are significantly more difficult to map to multiple platforms. The elements of a Class Diagram are discussed below.

Class Diagram

A real-world concept represented in the model is called a Class. Class diagrams, as the name implies, contain Classes and the relationships between Classes, called Associations. They can also contain notes that help a developer understand the diagram, and hence understand the relationships between the Classes present in the diagram. A Class can appear more than once in a diagram, and can appear in more than one diagram.

Developers should be careful to note that Class Diagrams are not the definitive source for defining the data model (the definitive source is the entire UML model, represented in an underlying format). Class Diagrams are used to convey a specific concept and, as such, may not represent all of the structure and relationships of a Class, for instance to focus on a subset of relationships that represent a specific part of the domain. Typically, the more complex a data model is, the more Class Diagrams are needed to explain a set of relationships. For instance, the `Protocol` package of FuGE represents the single most important piece of functionality, as the notion of protocols is generally applicable across any technology type and analysis routine. The FuGE `Protocol` package contains five Class Diagrams, and could conceivably contain more to clarify certain key points of this important part of the model.

The next sections will discuss the concepts found in Class Diagrams, Classes and Associations.

Class

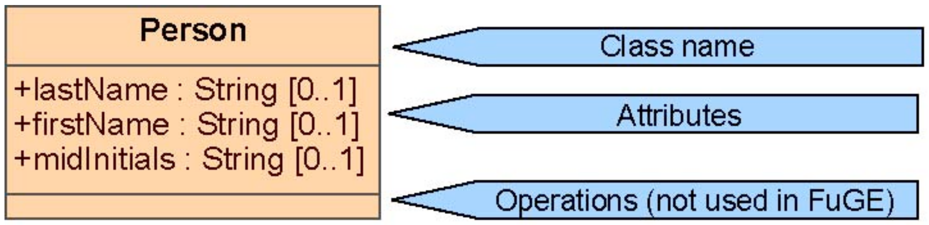
Person
+lastName : String [0..1]
+firstName : String [0..1]
+midInitials : String [0..1]
Class name
Attributes
Operations (not used in FuGE)

Figure 1 UML Class with compartments outlined

Classes are diagrammatically represented by a rectangle that is subdivided into three compartments (Fig. 1). The top compartment contains the class name and other intrinsic properties of the entire class. One type of intrinsic property used in the FuGE model is a Class Stereotype (Fig. 2). In UML, Stereotypes define the intended role a Class plays. For FuGE in particular, Stereotypes are used to drive the AndroMDA template engine to decide what type of document to produce for any given model element. For instance, almost all classes in the FuGE model have the “XmlSchemaType” stereotype, which signals to AndroMDA to convert a UML Class to an XML element in the FuGE XML schema. A very small number of FuGE classes, instead, have the “Enumeration” stereotype as shown in Figure 2b, for representing a list of possible values to be used in conjunction with another Class. In most cases, the class stereotype information is not shown in the diagrams.

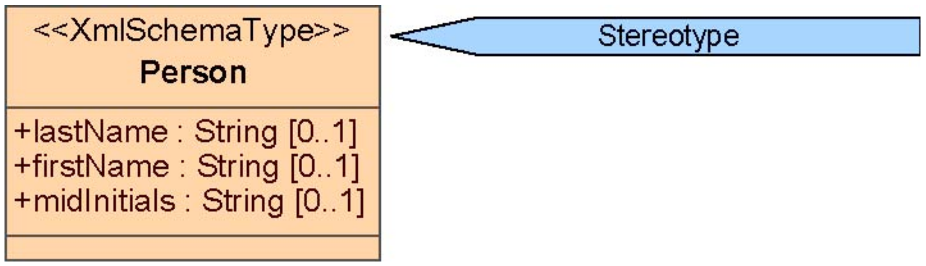
<<XmlSchemaType>>
Person
+lastName : String [0..1]
+firstName : String [0..1]
+midInitials : String [0..1]
Stereotype

Figure 2a UML Class Stereotype “XMLSchemaType” used on almost all elements in FuGE.

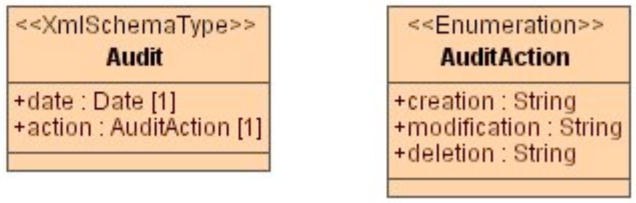

Figure 2b UML Class Stereotype “Enumeration”. The attributes in `AuditAction` can be used as an enumeration (a list of all the allowable values) for the attribute “action” on the Class `Audit`.

The middle compartment contains a list of attributes. For FuGE, except for enumerations, all attributes are singular simple data types, such as integers, Booleans, strings, etc. Attributes in the FuGE data model are defined as optional ("0..1") or mandatory ("1") by the cardinality specified next to the attribute name. UML in general can define a much broader set of cardinalities on class attributes than discussed here, but the FuGE project has so far restricted itself to using these two types to facilitate the mapping to XML. Attributes in UML can have the same types and constraints discussed in the section on Association cardinalities.

Finally, the bottom compartment of a UML class contains a listing of operations that are performed by the class. Since FuGE is a data model and does not contain operations, this compartment is not used. In certain FuGE diagrams, the bottom compartment is not shown for this reason.

UML Relationships

There are two general types of relationships between classes in FuGE: Associations and Generalizations.

Associations

Associations are the simplest type of relationship in UML, providing access between the Classes at either end of the Association (an Association End). Associations are diagrammatically represented as a line between two classes (Fig. 3) and have several components: Labels, Directionality, Cardinality, and Containment.

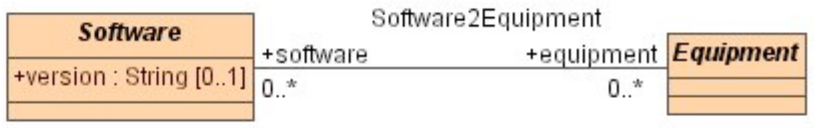

Figure 3 An Association between two Classes.

Labels

An Association can potentially have up to three labels, one for the entire Association (e.g. "Software2Equipment", and one label for each Association End ("software" and "equipment"). Of interest to FuGE developers are the Association End labels since these are used to traverse the Association from one class to another. Specifically, Classes gain the label at the opposite Association End to use for traversal of the Association. In the XML schema translation of FuGE, this corresponds to a sub-element that acts as a container for elements representing the class at the other Association End. For Java, this translates to the name of the collection object that is used to contain or reference the other class.

Directionality

Directionality defines whether one Association End is accessible from the other. Associations are either bi-directional, meaning each Association End is accessible from the other (hence both classes can traverse the relation), or unidirectional, meaning only one Association End is accessible and the relation can only be traversed in one direction. In Figure 4, the association is navigable from `Protocol` to `Software` but not *vice versa*. Uni-directional Associations are denoted by an arrow located at the Association End that is accessible. Lack of an arrow on either Association End denotes bi-directionality, as in Figure 3.

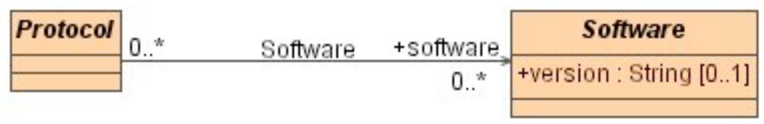

Figure 4 Unidirectional association between Protocol and Software

Cardinality

Each Association End contains a specification for its cardinality, or how many instances of a Class can (or must) be reachable at the Association End. Figure 4 demonstrates that a `Protocol` can be associated with multiple instances of `Software` (0..* on the `Software` end), and that each instance of `Software` can be used in conjunction with multiple `Protocols` (0..* on `Protocol` end).

	up to one	One	zero or more	more than one
up to one	`0..1 — 0..1`	`0..1 — 1`	`0..1 — 0..*`	`0..1 — 1..*`
One	`1 — 0..1`	`1 — 1`	`1 — 0..*`	`1 — 1..*`
zero or more	`0..* — 0..1`	`0..* — 1`	`0..* — 0..*`	`0..* — 1..*`
at least one	`1..* — 0..1`	`1..* — 1`	`1..* — 0..*`	`1..* — 1..*`

Table 1 A listing of possible Association End cardinality constraints and their implications. The numeric ranges are UML notation in the Class Diagrams; table headings represent the semantic meaning of the numeric ranges; the m-dashes represent an Association

Containment

It is often necessary to denote that a Class cannot exist independently from another class. For instance, the `Description` class would be useless without a reference from the class that it describes. Composition is denoted by a diamond shape at the Association End of the owning Class. The Class located at the contained (e.g. no diamond shape) Association End cannot exist independently of an owner (Fig. 5a).

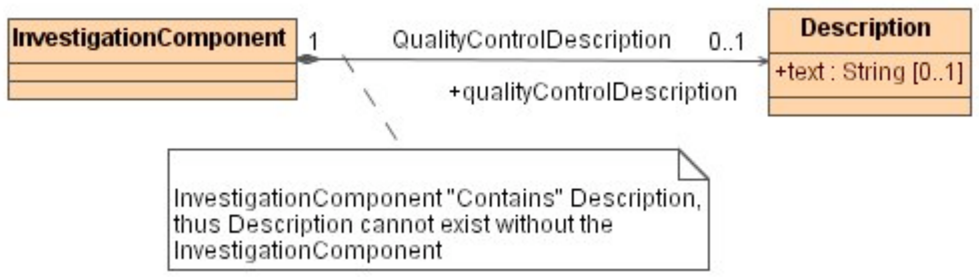
InvestigationComponent
1
QualityControlDescription
0..1
+qualityControlDescription
Description
+text : String [0..1]
InvestigationComponent "Contains" Description, thus Description cannot exist without the InvestigationComponent

Figure 5a A black diamond represents complete ownership of a Class instance: the instance of `Description` cannot exist independently of `InvestigationComponent`.

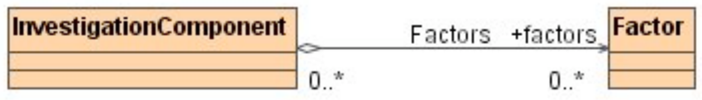

Figure 5b An open diamond represents shared ownership of a Class instance i.e. `Factors` are shared across `InvestigationComponents`.

There are two possible containment constraints on Classes: Composition and Aggregation. For the most part, the FuGE model uses the more restrictive Composition containment constraint, where the owning Class instance completely controls the existence of the contained Class instance(s). This is denoted by a closed, or black, diamond, as in Figure 5a.

Aggregate composition entails that an instance of a Class can be shared across separate instances of the parent, or even across instances of multiple parent types. Figure 5b illustrates this point using the FuGE classes `InvestigationComponent` and `Factor`. Briefly, `InvestigationComponent` represents a particular experimental technology platform, such as microarrays or tandem mass spectrometry, and `Factor` represents the principal comparators of an experiment, such as drug dosages or gene knock-outs. One can imagine a study that combines multiple high-throughput technologies, such as transcriptomics and proteomics, for studying the same condition, for instance the effect of COX inhibitors on cardiovascular tissue examined using microarrays and LC-MS/MS. The `Factor` (dosage) does not have meaning outside the scope of these experiments and should not exist without at least one of the `InvesitgationalComponents`; in effect the `Factors` are jointly owned between `InvestigationComponents`.

Generalizations

A Generalization represents an inheritance relationship between Classes, such that one Class is the descendant (child) of another (the parent). Inheritance implies that a descendent acquires all of the properties of all of its ancestors. FuGE utilizes a simple inheritance model where each Class is restricted to having only one direct parent, thus there is a direct line of descent, although a parent class may have multiple children. (Fig.

6). Multiple inheritance is difficult to map to different software platforms and rarely conveys any significant advantage to the data model.

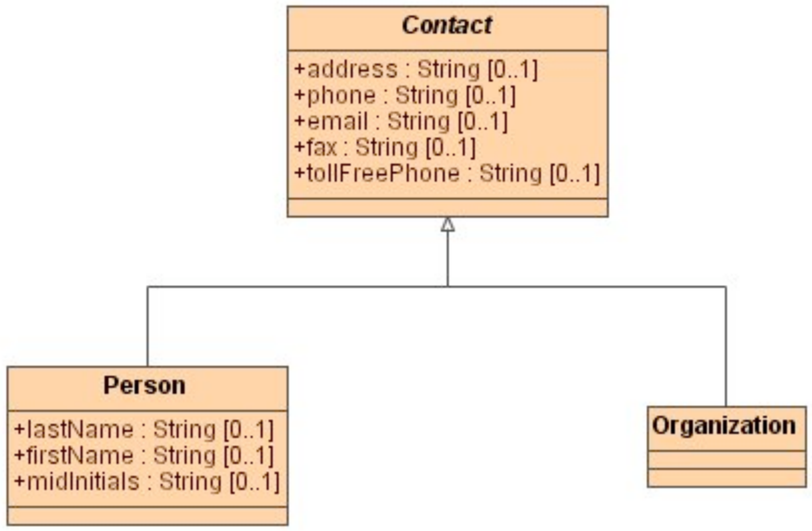

Figure 6 Generalization relationship between Classes. The child Classes (`Person` and `Organization`) point to the parent Class (Contact).

Abstraction

Classes can be set as abstract, denoted by the Class name in *Italic font*, meaning that the Class cannot be instantiated, and must be used in conjunction with a child class that is not abstract. Figure 6 demonstrates why this may be an advantageous design. Both `Person` and `Organization` share a number of common attributes, which have been assigned to the parent `Contact` Class. `Contact` is set to be abstract, because in the actual format all instances of `Contact` must be either a `Person` or an `Organization`; `Contact` itself cannot be instantiated.