

# COMN 1.1 Reference

---

Revision 1.1, 2017-03-30

by Theodore S. Hills, thills@acm.org. Copyright © 2015-2016

## Contents

|       |   |    |
|-------|---|----|
| 1     | Introduction.....                                   | 2  |
| 1.1   | Release 1.1 .....                                   | 3  |
| 1.2   | Release 1.0 .....                                   | 3  |
| 1.3   | Release 0.4 .....                                   | 3  |
| 1.4   | Release 0.3 .....                                   | 3  |
| 1.5   | Release 0.2 .....                                   | 3  |
| 2     | General Symbology Principles .....                  | 4  |
| 2.1   | Basic Polygons .....                                | 4  |
| 2.2   | Rectangles Depicting Types and Classes .....        | 5  |
| 2.2.1 | Typographical Conventions .....                     | 5  |
| 2.2.2 | Subdividing Rectangles .....                        | 5  |
| 2.2.3 | Component Typographical Conventions .....           | 7  |
| 2.2.4 | Multiplicity .....                                  | 7  |
| 2.2.5 | Optionality .....                                   | 7  |
| 2.2.6 | Unknown.....  | 8  |
| 2.2.7 | Key Notation.....                                   | 9  |
| 2.2.8 | Foreign Key Notation .....                          | 10 |
| 2.3   | Hexagons Depicting Variables and Objects.....       | 11 |
| 2.3.1 | Typographical Conventions .....                     | 11 |
| 2.3.2 | Subdividing Hexagons .....                          | 11 |
| 2.4   | Rounded Rectangles Depicting Values and States..... | 12 |
| 2.4.1 | Typographical Conventions .....                     | 12 |
| 2.4.2 | Subdividing Rounded Rectangles.....                 | 12 |
| 2.5   | Worlds .....  | 12 |
| 3     | Relationships.....                                  | 13 |
| 3.1   | Multiplicity and Optionality.....                   | 15 |

|   |    |
|---|----|
| COMN 1.1 Reference  | 2  |
| 3.2 Completeness .....  | 15 |
| 4 Hardware.....   | 16 |
| 4.1 Simple Hardware Objects, Classes, and Collections.....    | 16 |
| 4.2 Composite Hardware Objects, Classes, and Collections..... | 17 |
| 5 Software .....  | 19 |
| 6 Types and Variables.....                                    | 21 |
| 6.1 Simple Variables and Types.....                           | 21 |
| 6.2 Composite Variables, Types, and Collections .....         | 22 |
| 7 Real-World Objects and Concepts.....                        | 24 |
| 8 Relationships from Descriptors to That Described.....       | 24 |
| 8.1 Relationship between Type, Variable, and Value .....      | 24 |
| 8.2 Relationship between Class, Object, and State .....       | 25 |
| 9 Representation Relationships.....                           | 26 |
| 10 Derivation Relationships.....                              | 29 |
| 10.1 Extension Relationship.....                              | 29 |
| 10.2 Restriction Relationship .....                           | 32 |
| 11 Reference Relationships.....                               | 34 |
| 11.1 Containment.....   | 34 |
| 11.2 Collection.....  | 35 |
| 12 Composition Relationships .....                            | 36 |
| 12.1 Blending Relationship .....                              | 36 |
| 12.2 Aggregation Relationship.....                            | 36 |
| 12.3 Assembly Relationship.....                               | 37 |
| 13 Role Boxes .....   | 38 |

# 1 Introduction

This document provides a complete reference for the Concept and Object Modeling Notation (COMN, pronounced “common”), release 1.1. The book *NoSQL and SQL Data Modeling* ([Technics Publications](#), 2016) reflects release 1.0 of COMN.

This is a reference, not a tutorial. This document is designed to support a quick check of how to draw or notate something in COMN.

After the section on general symbology principles, the notation is presented starting with the most concrete things, namely hardware objects and their classes and states, then moves to

software objects and their classes and states. The reference then moves to the conceptual, namely variables and their types and values, and finally to real-world objects and their classes and states.

## 1.1 Release 1.1

Notation added to represent a view. “expr” preceding a type or class name indicates that the type or class is defined as an expression, such as a SQL view.

## 1.2 Release 1.0

A double-arrowhead symbol has been added to indicate composition by blending.

The form of the open arrow has been changed to be more similar to arrow styles available in many drawing tools.

The style of relationship lines, dashed or solid, has been changed to indicate whether the relationship is conceptual or physical, respectively. This is in keeping with the style of polygon outlines.

There is recognition that composite types may have methods. This does not change the symbology; it only changes its description and usage.

When a polygon is divided into sections, it is now stipulated that the name of the thing represented by the polygon be centered in the top section.

Nomenclature for referencing keys has been added.

Role boxes have been added.

## 1.3 Release 0.4

Multiplicity is now given following a component’s type’s name, since it changes the type (to an array).

A collection of variables, concepts, objects or states is now indicated by shadowing behind the polygon.

## 1.4 Release 0.3

Cardinality is now called multiplicity. The term cardinality is reserved for the actual count of items in a set. Multiplicity gives the allowed values for cardinality. Multiplicity is now indicated by integer expressions surrounded by curly braces, the asterisk, and the plus sign—exactly those symbols used in extended regular expressions.

## 1.5 Release 0.2

Cardinality is now indicated by expressions surrounded by square brackets.

A colon separates a component’s name from the type or class name which may follow it.

## 2 General Symbology Principles

COMN strives to use a relatively small number of symbols that can be combined in many ways to express the varieties of entities, attributes, and relationships found in the real world, in data, and in objects. This approach is one of *intrinsic simplicity* but *combinatorial complexity*.

COMN's graphic symbols include:

- 3 basic polygons (rectangle, hexagon, rounded rectangle) representing types/classes, variables/objects, and concepts/states
- a pentagon to show restriction (subtyping)
  - optional X for exclusive subtyping
- a triangle to show extension
  - optional X for exclusive extension
- the line for relationships
  - arrowheads: none, open arrow, closed arrow, double closed arrow, closed circle
  - tails: none, open circle, X
- multiplicity specified by number ranges and symbols (+, \*) on both relationship lines and components
- optionality and unknown-ness represented distinctly by '?' in different positions

### 2.1 Basic Polygons

There are three kinds of polygons, used as shown in Table 1 below. The name of the thing represented is typically centered in the polygon. Typographical conventions are as shown in the table.

The names of logical things (types, variables, values) and of real-world things follows general English rules. More particular rules are given in following sections.

The names of implementation things (software classes and objects) are given following the generally strict rules imposed by implementation contexts. In most cases these names may only use alphabetic letters, numbers, and the underscore character (“\_”). Some implementation contexts are less restrictive; some are more restrictive.




| Shape             | Example   | Represents  |
|-------------------|---|---|
| rectangle         |  | a type or a class<br>name centered in rectangle, Title Case   |
| hexagon           |  | a variable or an object<br>name centered in hexagon<br>Title Case (proper noun) if specific instance identified<br>lower case with preceding “a” or “an” if typical instance identified |
| rounded rectangle |  | a concept (including a value or a meaning) or a state<br>name centered in rounded rectangle<br>no particular letter case dictated   |

Table 1. COMN Polygons

## 2.2 Rectangles Depicting Types and Classes

### 2.2.1 Typographical Conventions

A type or class name is usually given in Title Case.

If a composite type or class extends another type or class, this may be indicated by including the name of the base type or class, enclosed in guillemets, following the type or class name, as in “«base class»”. Alternatively, this may be indicated with an extension symbol. See the section on Extension Relationship.

If a composite type or class is defined as an expression, such as an SQL view, the type or class name should be preceded by “expr”.

### 2.2.2 Subdividing Rectangles

A rectangle, depicting a type or class, may be divided into three sections by two horizontal lines. The topmost section contains the name of the type or class, centered, possibly including a base class enclosed in guillemets ( « » ). The middle section lists the components of the type or class, left justified. If the thing represented can’t have components (such as a class interface), the middle section is shown with diagonal crossed lines through it. The bottom section lists the methods of the type or class. If the thing represented can’t have methods (such as an unencapsulated composite type), the bottom section is shown with diagonal crossed lines through it. See Table 2 below for illustrations.

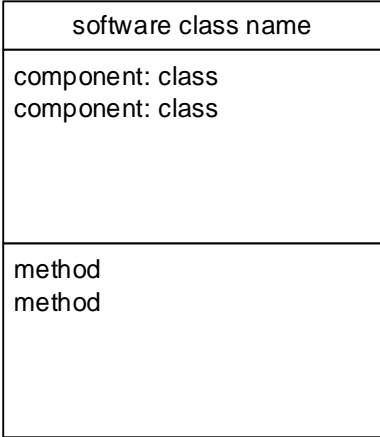
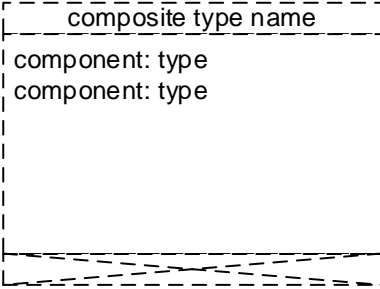
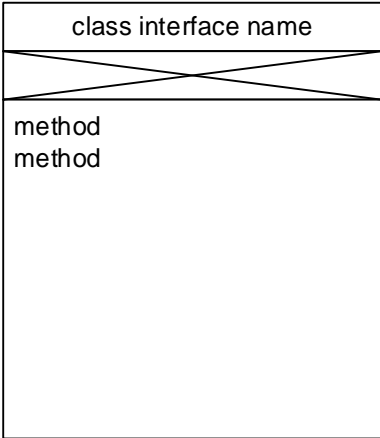
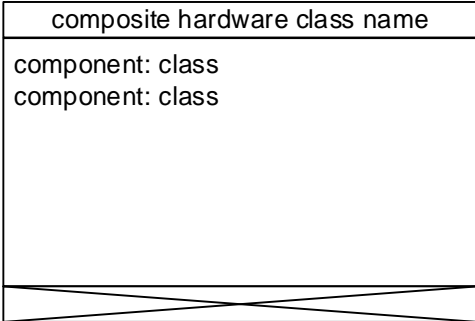
| Shape  | Example   | Represents                              |
|--|---|---|
| divided solid rectangle                              |    | a software class                        |
| divided dashed rectangle, bottom section crossed out |   | a type with no methods                  |
| divided solid rectangle, middle section crossed out  |  | a class interface (no components)       |
| divided rectangle, bottom section crossed out        |  | a composite hardware class (no methods) |

Table 2. Divided Composite Type and Class Rectangles

### 2.2.3 Component Typographical Conventions

In a divided polygon, the name of the thing represented is centered in the top section, component names, if any, are left-justified in the middle section, and method names, if any, are left-justified in the bottom section. If a component's type or class is shown, it follows the component name and is separated from the component name by a colon.

### 2.2.4 Multiplicity

In a type or class, a component's type may be followed by expressions that indicate number or optionality, as follows<sup>1</sup>:

- no suffix: the component must appear exactly once in every instance of the type or class; for example, "String"
- a plus sign, indicating that one to any integral number of elements may occur; for example, "PhoneNumber+"
- an asterisk, indicating that zero to any integral number of elements may occur; for example, "String\*"
- integer expressions enclosed in a pair of curly braces("{ " and "}") giving the possible numbers of instances of the component. This defines the component as an array. The expressions can take the following forms.
  - a single positive integer, indicating exactly that many elements will occur; for example, "String{3}"
  - a range of integers specified as two non-negative integers separated by a hyphen; for example, "Address{0-2}"
  - a comma-separated list of non-negative integers giving allowable numbers of elements; for example, "Rational{0, 2, 4, 6}"
  - any combination of number ranges and non-negative integers; for example, "ArgType{0, 2-5, 9}"

These symbols may also be used on the end of a relationship line, indicating the multiplicity of occurrences of the thing at that end of the line.

Any multiplicity other than 1 indicates that the component's type is an array whose element type is given by the type preceding the expression.

### 2.2.5 Optionality

In a type or class, a component's name may be preceded by a question mark ("?"), indicating that the component is optional, and may appear zero or one times in any instance of the type or class; for example, "?MiddleName"

- This is not the same as potentially unknown, which is signified by a question mark following the component's type or class name. See below.

---

<sup>1</sup> Some readers might recognize that these expressions are used in extended regular expressions.

- This is not the same as “{0-1}” or “{0,1}”, which define the component as an array of zero or one elements, and is generally deprecated.

The question mark may also be used on the end of a relationship lines, indicating the optionality of occurrence of the thing at that end of the line.

An optional component is implicit extension. The presence of an optional component in a type or class implicitly defines two types or classes: a base type or class without the optional component, and an extending type or class that adds the optional component.

## 2.2.6 Unknown

A component’s type’s name may be followed by a question mark; for example, “BirthDate: date?”. This indicates that the type is extended to include an additional value that represents that the actual value of the component is unknown.

- This is not the same as an optional component, which is signified by a question mark preceding the component’s name. See above.

Multiplicity, optionality, and unknown-ness may be combined in a single component.

### 2.2.6.1 Example

See Figure 1 below for an example illustrating component number and optionality.

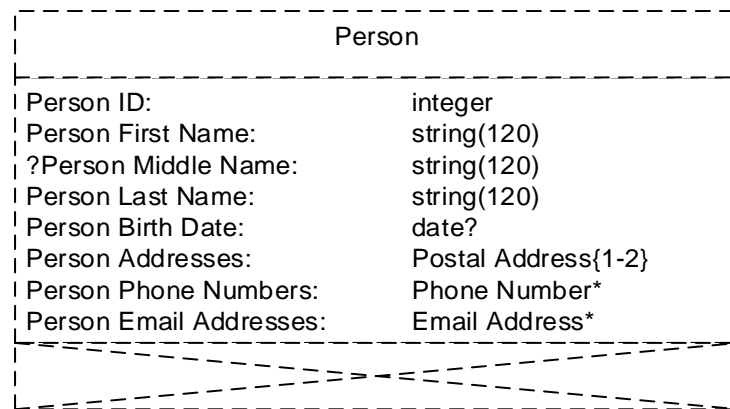


Figure 1. Illustration of Component Number and Optionality

- Person First Name, Person Last Name, Person Birth Date, and at least one Person Address must exist in every instance of this type.
- Person Middle Name may or may not exist in any instance of this type.
- Although Person Birth Date must exist in every instance, its value may indicate that the actual date is unknown. This reflects the reality that every person has a birth date, though it is not always known.
- The components Person Addresses, Person Phone Numbers, and Person Email Addresses are arrays whose element types are given to the right of the component names.
- There must be at least one, and might be as many as two, Person Addresses in any instance of this type.



- There can be any number of Person Phone Numbers and Person Email Addresses, including zero, in any instance of this type.

### 2.2.7 Key Notation

When a composite type or class is a type or class of a collection of variables or objects, the collection may have a key. A key is a component or set of components whose values or states are always unique in any set of variables or objects of the type or class. A key with a single component is a simple key; a key with multiple components is a composite key. A collection may have any number of keys.

There are three types of keys:

- candidate key (abbreviated as CK): Any component or set of components that are a key are properly referred to as a candidate key.
- primary key (abbreviated as PK): In some contexts, a particular CK might be elevated to the status of a primary key. A collection may have no more than one PK. At the physical level, a CK might be chosen as the PK because access to the collection will be most frequently via the PK, and therefore access via the PK should be optimized (usually by building a PK index or by organizing the collection in PK order). At a logical level, there is rarely good reason to identify one key from the set of CKs as a PK. Notwithstanding, it is almost a universal practice to talk of a collection's PK even when it is the collection's only key or when there is insufficient justification to select a CK as the PK.
- alternate key (abbreviated as AK): When one CK of a collection is selected to be its PK, then the remaining keys may be referred to as AKs.

The participation of a component in a key is indicated by following the component's name with CK, PK, or AK in parentheses. If the component's type or class is included, then the CK, PK, or AK precedes the colon that separates the component name from its type or class. This is appropriate as a key designation, like a component's name, indicates its role and not its type.

When there is more than one CK or AK, the CK/AK letters are followed by a natural number to distinguish them. When there is more than one component to a key, a suffix of a period (".") and a natural number is used to distinguish the components of the key.

#### 2.2.7.1 Examples:

A collection with a single simple key.

Person ID (CK): integer

OR

Person ID (PK): integer

A collection with a single composite key:

Order ID (CK.1): integer

Coffee Shop ID (CK.2): integer

OR

Order ID (PK.1): integer

Coffee Shop ID (PK.2): integer

A collection with two simple keys:

User ID (CK1): string{1-255}

Employee ID (CK2): integer

OR

User ID (PK): string{1-255}

Employee ID (AK): integer

A collection with a simple key and a composite key:

Employee ID (CK1): integer

User ID Domain Name (CK2.1): string{1-255}

User ID (CK2.2): string{1-255}

OR

Employee ID (PK): integer

User ID Domain Name (AK.1): string{1-255}

User ID (AK.2): string{1-255}

### 2.2.8 Foreign Key Notation

When a composite type or class contains a component or components whose values or states reference a key of some collection, those components are called a **foreign key**. A foreign key is indicated as the type of the component or components. The type is given as FK followed by an expression in parentheses that identify the collection referenced. If the collection referenced has more than one key, then the reference to the collection is followed by a comma and the collection's key designation as described in the previous section.

If the foreign key referenced is a composite key, then either:

1. The foreign key component is a composite component, and the FK type expression is given as described above; or
2. The foreign key is provided as multiple components, one per referenced component, and each component's FK type expression references a single key component, using the designations described in the previous section.

#### 2.2.8.1 Examples:

A foreign key reference to a collection with a single simple key.

Supervisor Person ID: FK(Persons)

A foreign key reference to a collection with a single composite key:

Coffee Shop Order: FK(Orders)

OR

Coffee Shop Order Order ID: FK(Orders, CK.1)

Coffee Shop Order Coffee Shop ID: FK(Orders, CK.2)

A foreign key reference to a collection with two simple keys:

User ID: FK(Users, CK1)

OR

User Employee ID: FK(Users, CK2)

A foreign key reference to a collection with a simple key and a composite key:

User Employee ID: FK(Users, CK1)

OR

User ID: FK(Users, CK2)

OR

User ID Domain Name: FK(Users, CK2.1)

User ID: FK(Users, CK2.2)

## 2.3 Hexagons Depicting Variables and Objects

### 2.3.1 Typographical Conventions

If a variable or object is meant to depict a typical instance, then its name should be all in lower case, preceded by an indefinite article (“a” or “an”); for example, “a person”, “an order”.

If a variable or object is meant to depict a particular instance, then its name is a proper noun and it should be in Title Case, preceded by the definite article (“The”) if appropriate; for example, “The White House”, “The Magna Carta”, “IBM Corporation”.

### 2.3.2 Subdividing Hexagons

A hexagon representing a composite variable or object may be divided into two sections by a horizontal line in order to show the values or states of its components. The line is placed near the top of the symbol. The topmost section contains the name of the variable or object, centered, possibly including the type or class of the variable or object enclosed in guillemets ( « » ). In the space below the line, component names are listed along with an equal sign and an expression giving the component’s value in the instance represented by the hexagon. See Figure 2 below for an example. Conventions for component names, multiplicity, and optionality are the same as for components of types and classes. See the section above, Subdividing Rectangles.

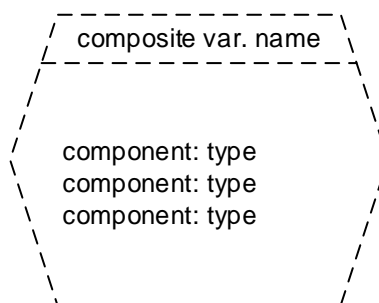


Figure 2. Subdivided Hexagon

## 2.4 Rounded Rectangles Depicting Values and States

### 2.4.1 Typographical Conventions

No typographical conventions have been established for the names of values and states.

### 2.4.2 Subdividing Rounded Rectangles

A rounded rectangle representing a composite value or state may be divided into two sections by a horizontal line in order to show the values or states of its components. The line is placed near the top of the symbol. The topmost section contains the name of the value or state, centered. In the space below the line, component names are listed along with an equal sign and an expression giving the component's value or state in the composite value or state represented by the rounded rectangle. See Figure 3 below for an example.

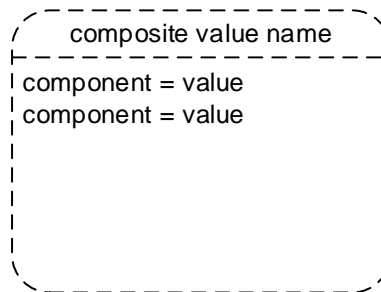


Figure 3. Subdivided Rounded Rectangle

## 2.5 Worlds

Polygon outlines can be solid or dashed, normal or bold, as shown in Table 3 below.


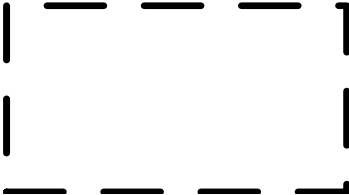

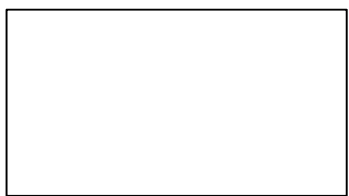


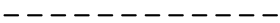

| Outline                     | Example  | Represents  |
|-----------------------------|--|---|
| bold solid line<br>(3 pt.)  |   | a real-world class, object (hexagon), or state (rounded rectangle)  |
| bold dashed line<br>(3 pt.) |   | a type, concept (hexagon), or value (rounded rectangle) of something that is not symbolic or lexical (though its representation might necessarily be symbolic or lexical) |
| dashed line<br>(0.72 pt.)   |   | a type, variable (hexagon), or value (rounded rectangle) of something that is symbolic or lexical. Such a type is often called a logical type or logical data type        |
| solid line<br>(0.72 pt.)    |  | a software class, object (hexagon), or state (rounded rectangle)  |

Table 3. Worlds Indicated by Line Style

### 3 Relationships

A relationship between any two things is represented as a line connecting the symbols that represent those things. The line may be straight, curved, or angled at one or more points. The shape of the line does not alter its meaning.

Lines representing relationships indicate the “world” in the same way as the outline of polygons indicate the world, as show in Table 4 below.

| Outline                     | Example   | Represents  |
|-----------------------------|---|---|
| bold solid line<br>(3 pt.)  |  | a physical relationship between two real-world objects  |
| bold dashed line<br>(3 pt.) |  | a conceptual relationship between two things where at least one thing is not symbolic or lexical            |
| dashed line<br>(0.72 pt.)   |  | a conceptual relationship between two things where at least one thing is symbolic or lexical                |
| solid line<br>(0.72 pt.)    |  | a physical relationship between two computer objects; that is, between two hardware and/or software objects |

**Table 4. Relationship Worlds Indicated by Line Style**

An unadorned line—that is, a line without arrowheads or other decorations at either end—indicates a relationship generically. If a relationship line connects polygons of different shape, the line does not need to be annotated to indicate the nature of the relationship, as only certain kinds of relationships can exist between the things represented by those polygons, as Table 5 below shows.

| polygon 1                 | polygon 2                        | meaning of unadorned relationship line  |
|---------------------------|----------------------------------|---|
| type (dashed rectangle)   | variable (dashed hexagon)        | the variable has the type               |
| type (dashed rectangle)   | value (dashed rounded rectangle) | the type includes the value             |
| variable (dashed hexagon) | value (dashed rounded rectangle) | the variable is bound to the value      |
| class (solid rectangle)   | object (solid hexagon)           | the object is an instance of the class  |
| class (solid rectangle)   | state (solid rounded rectangle)  | objects of the class may have the state |
| object (solid hexagon)    | state (solid rounded rectangle)  | the object has the state                |

**Table 5. Relationships between Dissimilar Polygons**

If an unadorned line connects polygons of the same shape, then the particular nature of the relationship must be spelled out by a verb or verb phrase, in the present tense, placed next to the line approximately in the center of the line, which can be used to form a phrase using the names of the thing at each end of the line as subject and object. For example, a line from a symbol named A to a symbol named B which represents an implementation relationship should have the verb “implements” next to the line near its center, and can be read, “A *implements* B”. If the phrase cannot be read left-to-right, either because the two symbols are not positioned in a left-to-

right relationship, or because they are but in right-to-left order, either or both of two alternatives may be employed:

1. An arrow may be placed at the beginning or end of the verb phrase to indicate which way to read the relationship. For example, if symbol A is to the right of symbol B, then the word “implements” can be preceded by an arrow, as in “←implements”, to indicate that A implements B and not the other way around.
2. The verb phrase may be reworded to indicate the reverse relationship when reading left-to-right. In the example above, the verb phrase may be altered to “is implemented by”, to indicate, in a left-to-right reading, that B is implemented by A.

A relationship line may be adorned with either or both verb phrases, each with an arrow to indicate the reading direction, in order to supply the intended wording for the relationship in both directions. For example, “←implements” and “is implemented by→”. See Figure 4 below.

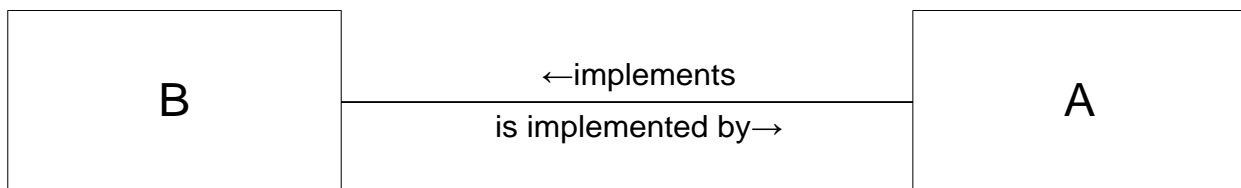


Figure 4. Labeling Unadorned Relationship Lines

There are certain relationship types that occur frequently and/or have special significance. For these relationships, there are particular symbols used at the beginnings and/or ends of each relationship line. If these symbols are arrowheads or are similar to arrowheads, then, by convention, these arrowheads indicate **direction of reference**, which indicates which notational element references another. The directionality of such arrowheads should not be interpreted as indicating a direction of data flow or anything else, but only direction of reference. The significance of various arrow heads and tails on relationship lines is described in the sections on the various kinds of relationships.

### 3.1 Multiplicity and Optionality

In the absence of other markings, a relationship line indicates that exactly one instance of the thing at one end of the line is related to exactly one instance of the thing at the other end of the line. Multiplicity and optionality is indicated using the same expressions as used within a list of components in a type or class rectangle. See the section Multiplicity under Rectangles Depicting Types and Classes, above.

### 3.2 Completeness

Given any two symbols on the same page of a diagram, *every relationship must be shown* between the two symbols that exist in the model. This convention enables a diagram to indicate whether two things represented are in fact *not* directly related. Without this convention, a diagram could never be interpreted as being complete in the scope of the symbols in the diagram.

This leads to a strategy of diagram management where symbols are added to or removed from a diagram in order to keep the diagram comprehensible, given the number of relationship lines that must appear. A diagram becomes focused on a “subject area”, and becomes a “package” or subset of an overall model. It also leads to a strategy where primitive things are kept off most

diagrams. For instance, the symbols for widely used types or classes, such as the Integer type or the ComputerObject class, will rarely be shown on any diagram, since the number of relationship lines to such symbols would be too great to be informative.

## 4 Hardware

Computers are composed of material objects that are stateful. When the state of one of these stateful material objects can be read and/or altered by a computer's instructions, we call it a hardware object.

### 4.1 Simple Hardware Objects, Classes, and Collections

An individual, simple hardware object is represented by an irregular hexagon, with an X through it to indicate that it has no components, and the name of the object in the center of the symbol. See Figure 5 below.

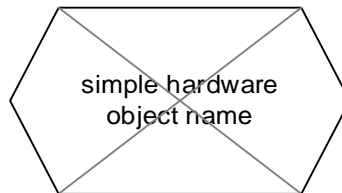


Figure 5. A Simple Hardware Object

A description of potential or actual simple hardware objects is called a simple hardware class. A simple hardware class is represented by a rectangle with an X through it to indicate that it has no components. See Figure 6 below. Since simple hardware objects have no components, and hardware classes have no methods, the symbol for a simple hardware class is never divided into sections.

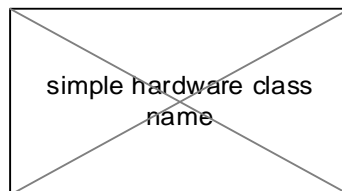


Figure 6. A Simple Hardware Class

A collection of simple hardware objects is represented by a simple hardware object symbol with a shadow. See Figure 7 below. Rather than the name of a particular object, the name of the collection is in the middle of the symbol.



Figure 7. A Simple Hardware Object Collection



A particular state of a simple hardware object is represented by a rectangle with rounded corners, with an X through it, and the name of the state in the center of the symbol. See Figure 8 below.

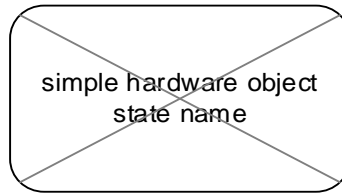


Figure 8. A Simple Hardware Object State

A set of states of a simple hardware object is represented by a simple hardware object state symbol with a shadow. See Figure 9 below. Rather than the name of a particular state, the name of the set of states is in the middle of the symbol.

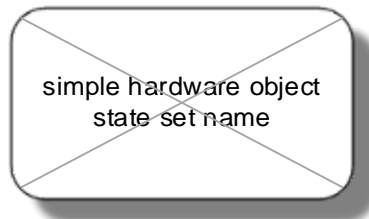


Figure 9. A Simple Hardware Object State Set

## 4.2 Composite Hardware Objects, Classes, and Collections

An individual, composite hardware object is represented by an irregular hexagon, with the name of the object in the center of the symbol. See Figure 10 below. (This is the same symbol as is used for a software object.)

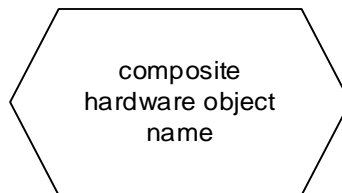
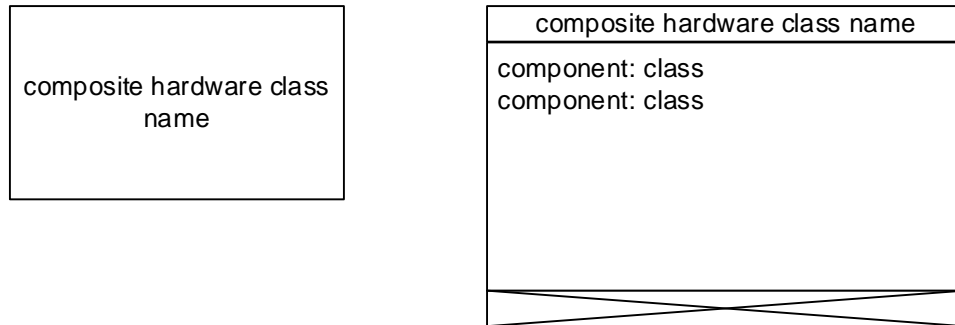


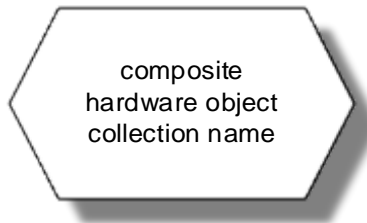
Figure 10. A Composite Hardware Object

A description of potential or actual composite hardware objects is called a composite hardware class. A composite hardware class is represented by a rectangle, which may or may not be divided into three sections. See Figure 11 below. The undivided rectangle is the same symbol that is used for software classes. Note the X through the section of the divided symbol normally reserved for methods. Since a hardware class cannot encapsulate, because it is not a software artifact, it cannot have methods which have exclusive access to the components.



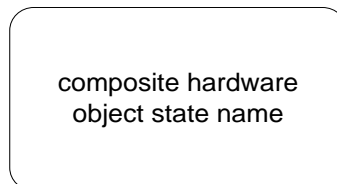
**Figure 11. A Composite Hardware Class**

A collection of composite hardware objects is represented by a composite hardware object symbol with a shadow. See Figure 12 below. (This is the same symbol as is used for a collection of software objects.) Rather than the name of a particular object, the name of the collection is in the middle of the symbol.



**Figure 12. A Composite Hardware Object Collection**

A particular state of a composite hardware object is represented by a rectangle with rounded corners, with the name of the state in the center of the symbol. See Figure 13 below. (This is the same symbol as is used for the state of a software object.)



**Figure 13. A Composite Hardware Object State**

The states of the components of a composite hardware object state may be illustrated as shown in Figure 14 below.

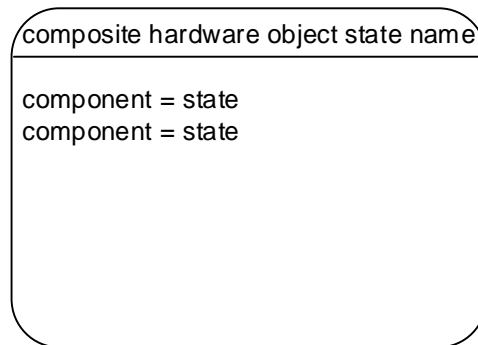


Figure 14. A Composite Hardware Object State with Component States

A set of states of a composite hardware object is represented by a composite hardware object state symbol with a shadow. See Figure 15 below. (This is the same symbol as is used for a set of states of a software object.) Rather than the name of a particular state, the name of the set of states is in the middle of the symbol.

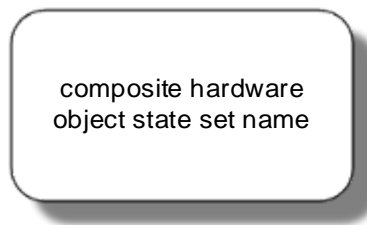


Figure 15. A Composite Hardware Object State Set

## 5 Software

All software objects, their classes, and their states are composite. The symbology for software objects reflects this.

An individual software object is represented by an irregular hexagon, with the name of the object in the center of the symbol. See Figure 16 below.

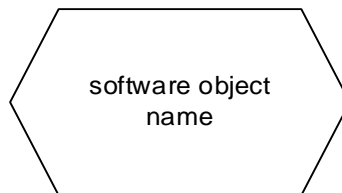


Figure 16. A Software Object

A description of potential or actual software objects is called a software class. A software class is represented by a rectangle, which may or may not be divided into three sections. See Figure 17 below.

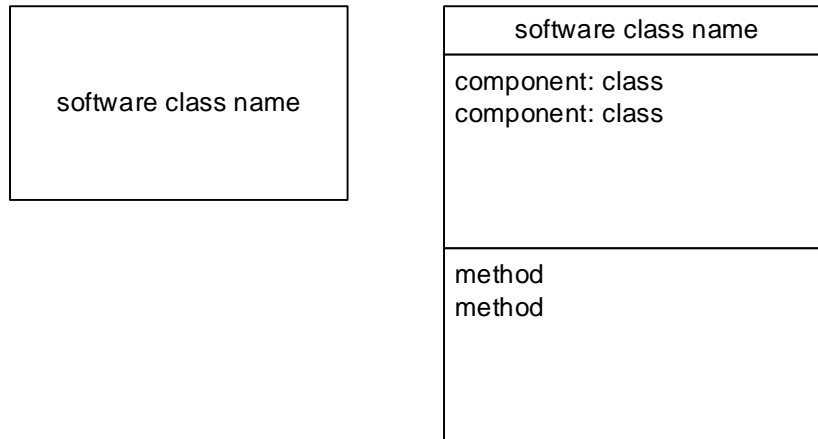


Figure 17. A Software Class

A collection of software objects is represented as a software object symbol with a shadow. See Figure 18 below. Rather than the name of a particular object, the name of the collection is in the middle of the symbol.

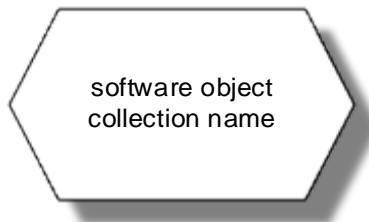


Figure 18. A Software Object Collection

A particular state of a software object is represented by a rectangle with rounded corners, with the name of the state in the center of the symbol. See Figure 19 below.

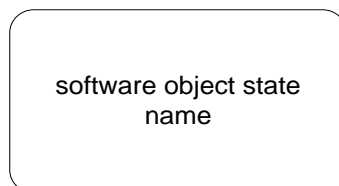


Figure 19. A Software Object State

The states of the components of a software object state may be illustrated as shown in Figure 20 below.

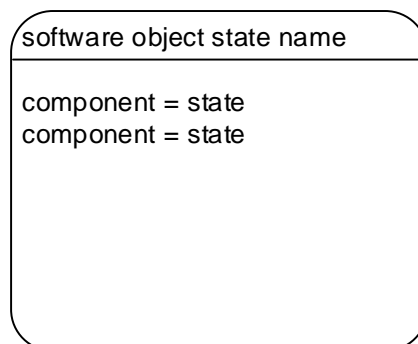


Figure 20. A Software Object State with Component States

A set of states of a software object is represented by a software object state symbol with a shadow. See Figure 21 below. Rather than the name of a particular state, the name of the set of states is in the middle of the symbol.

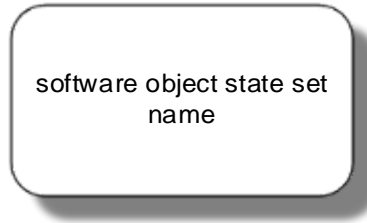


Figure 21. A Software Object State Set

## 6 Types and Variables

### 6.1 Simple Variables and Types

An individual, simple variable is represented by a dashed irregular hexagon, with an X through it to indicate that it has no components, and the name of the variable in the center of the symbol. See Figure 22 below.

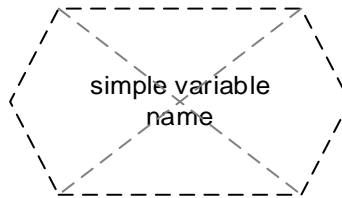


Figure 22. A Simple Variable

The type of a simple variable is a simple type, which is represented by a dashed rectangle with an X through it to indicate that it has no components. See Figure 23 below. Since simple types have no components and no methods, the symbol for a simple type is never divided into sections.

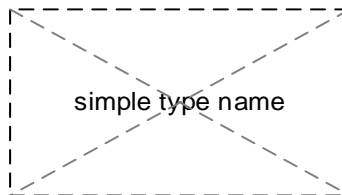


Figure 23. A Simple Type

A collection of simple variables is represented by a simple variable symbol with a shadow. See Figure 18 below. Rather than the name of a particular variable, the name of the collection is in the middle of the symbol.

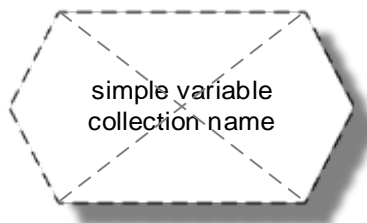


Figure 24. A Simple Variable Collection

A particular value of a simple type is represented by a dashed rectangle with rounded corners, with an X through it, and the name of the value in the center of the symbol. See Figure 25 below.

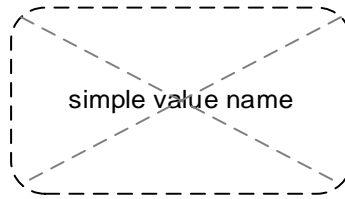


Figure 25. A Simple Value

A set of values of a simple type is represented by a simple type value symbol with a shadow. See Figure 26 below. Rather than the name of a particular value, the name of the set of values is in the middle of the symbol.

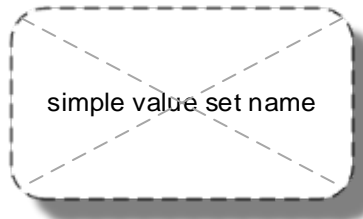


Figure 26. A Simple Value Set

## 6.2 Composite Variables, Types, and Collections

An individual, composite variable is represented by a dashed irregular hexagon, with the name of the variable in the center of the symbol. See Figure 27 below.

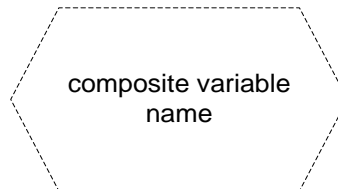


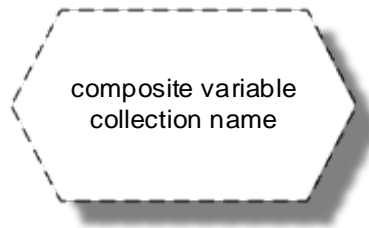
Figure 27. A Composite Variable

A description of potential or actual composite variables is called a scheme. A scheme is represented by a rectangle, which may or may not be divided into three sections. See Figure 28 below.



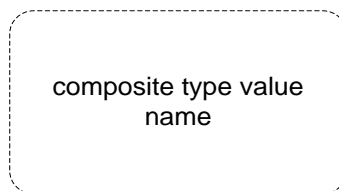
Figure 28. A Composite Type

A collection of composite variables is represented by a composite variable symbol with a shadow. See Figure 29 below. Rather than the name of a particular variable, the name of the collection is in the middle of the symbol.



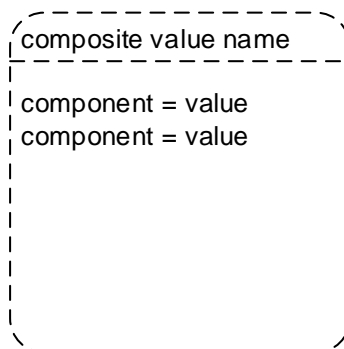
**Figure 29. A Composite Variable Collection**

A particular value of a composite type is represented by a dashed rectangle with rounded corners, with the name of the value in the center of the symbol. See Figure 30 below.



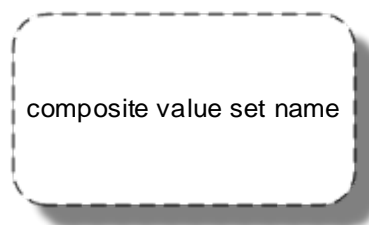
**Figure 30. A Composite Type Value**

The values of the components of a composite value may be illustrated as shown in Figure 31 below.



**Figure 31. A Composite Value with Component Values**

A set of values of a composite type is represented by a composite type value symbol with a shadow. See Figure 32 below. Rather than the name of a particular value, the name of the set of values is in the middle of the symbol.



**Figure 32. A Composite Value Set**

## 7 Real-World Objects and Concepts

Physical objects may exist outside any computer, in what we call “the real world”. When depicting real-world physical objects, their classes, collections of such objects, and their states, we use the same solid-outline symbols as for software objects, except that we use heavier (bold) lines for their outlines and internal lines.

Variables, their types, collections of variables, and their values are always conceptual, so there is no need to distinguish whether they exist in the real world or inside a computer: they always exist only conceptually. Concepts that are not related to data (examples: roles played by persons, the concept of a law) are depicted with the same dashed-line symbols as for variables, types, and values, except that we use heavier (bold) lines for their outlines and internal lines

## 8 Relationships from Descriptors to That Described

### 8.1 Relationship between Type, Variable, and Value

The relationships between types, variables, and values may only be as shown in Figure 33 below, so there is no need to label these relationships explicitly, nor is there any need for arrowheads or other adornments. These relationships hold for both simple and composite types, variables, and values.



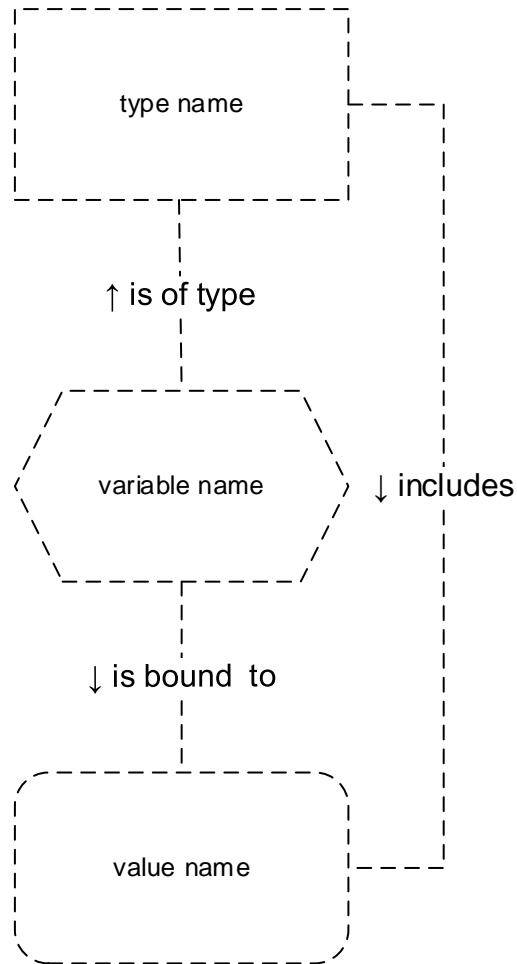


Figure 33. Relationships between Type, Variable, and Value (Labels Unnecessary)

## 8.2 Relationship between Class, Object, and State

The relationships between classes, objects, and states may only be as shown in Figure 34 below, so there is no need to label these relationships explicitly, nor is there any need for arrowheads or other adornments. These relationships hold for both simple and composite classes, objects, and states.

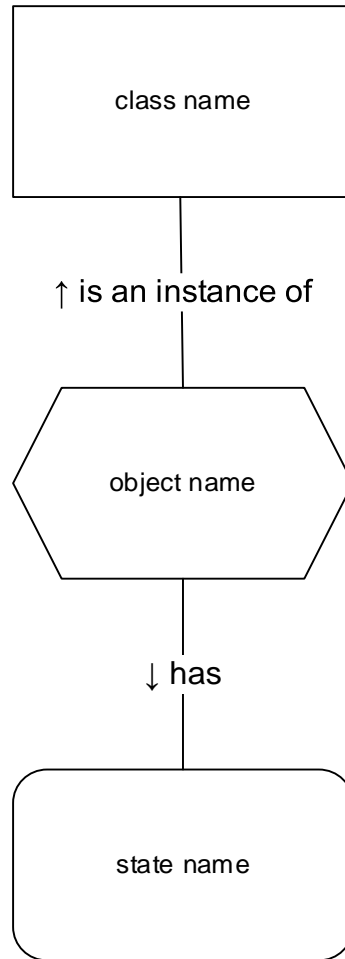


Figure 34. Relationships between Class, Object, and State (Labels Unnecessary)

## 9 Representation Relationships

A representation relationship may exist between many combinations of types, classes, variables, objects, values, and states. A relationship line is adorned with a solid circle at the end next to the thing being represented by the thing at the other end of the line. Representation is always conceptual; therefore, a representation line is always drawn dashed. See Figure 35 below for an example of how classes may represent types, objects may represent variables, and states may represent values. The words “represents” are not necessary, as the solid circle indicates that that is the meaning of the relationship line.

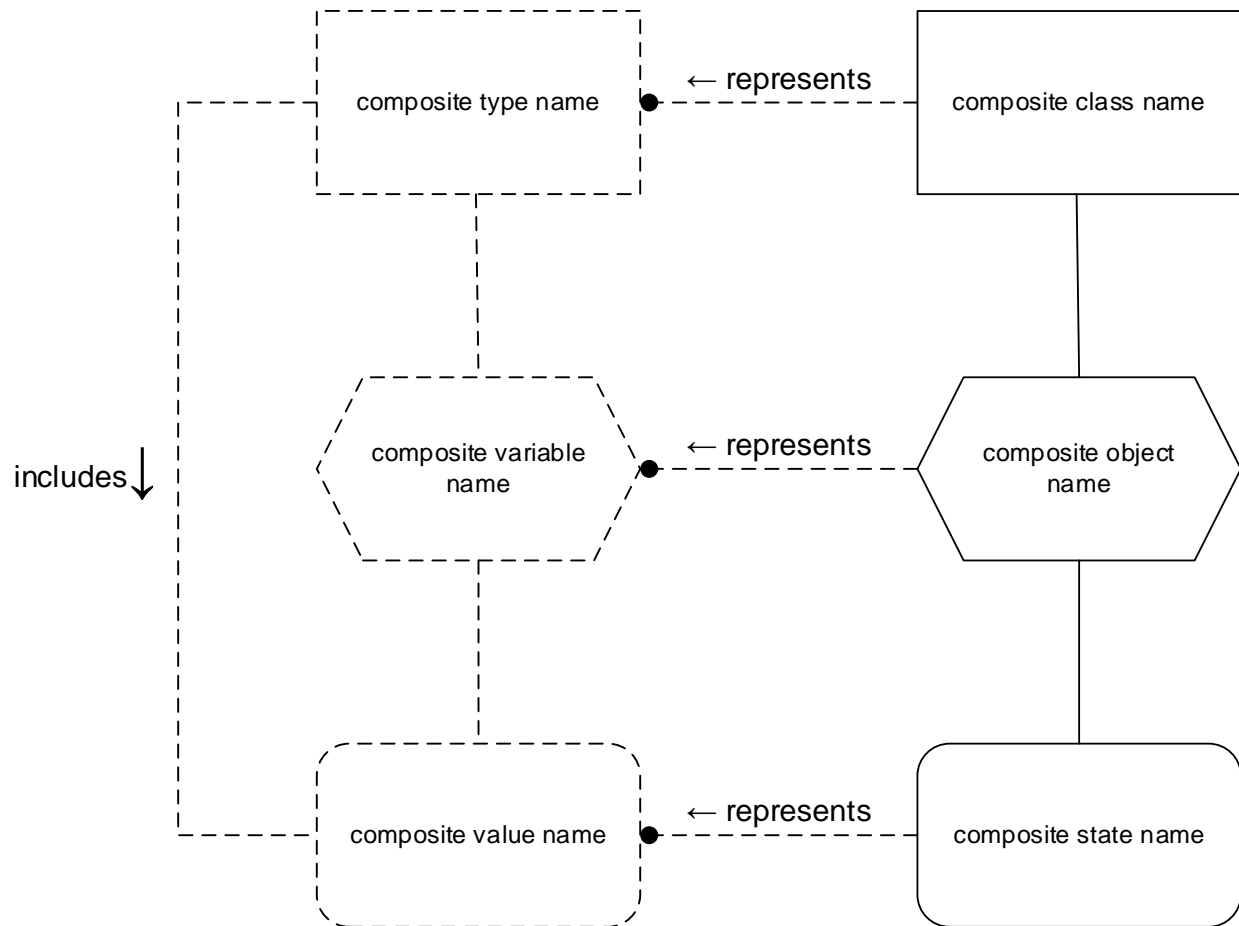
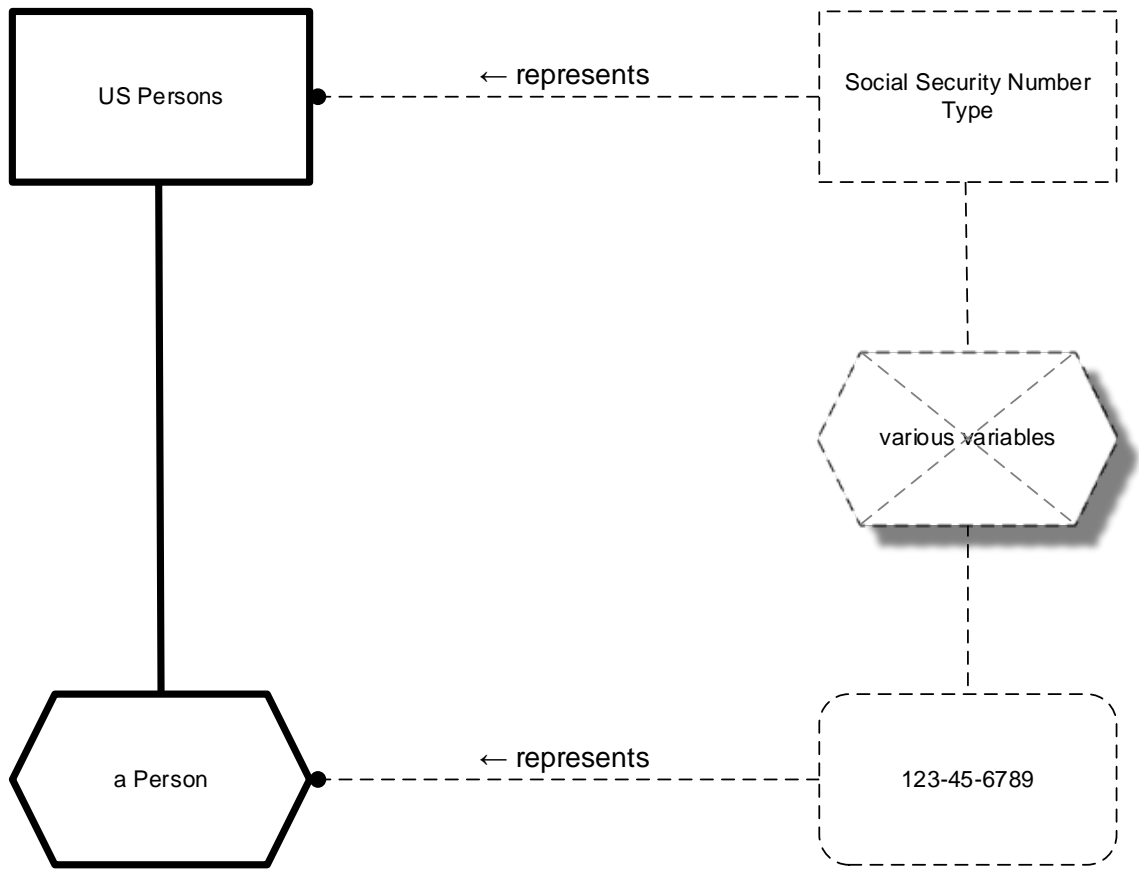


Figure 35. Representation of Logical Things by Physical Things

A common case of representation is for values to represent objects. We call this **identification**. See Figure 36 below for an example of identification.



**Figure 36. Identification**

Three levels of abstraction of data modeling can be depicted as in Figure 37 below. It is important to notice that there is no conceptual level, as there is in entity-relationship modeling. Instead, there is a real-world level, depicting things in the real world. The conceptual level is nothing more than a simplification of the logical level, by suppressing the display of attributes.

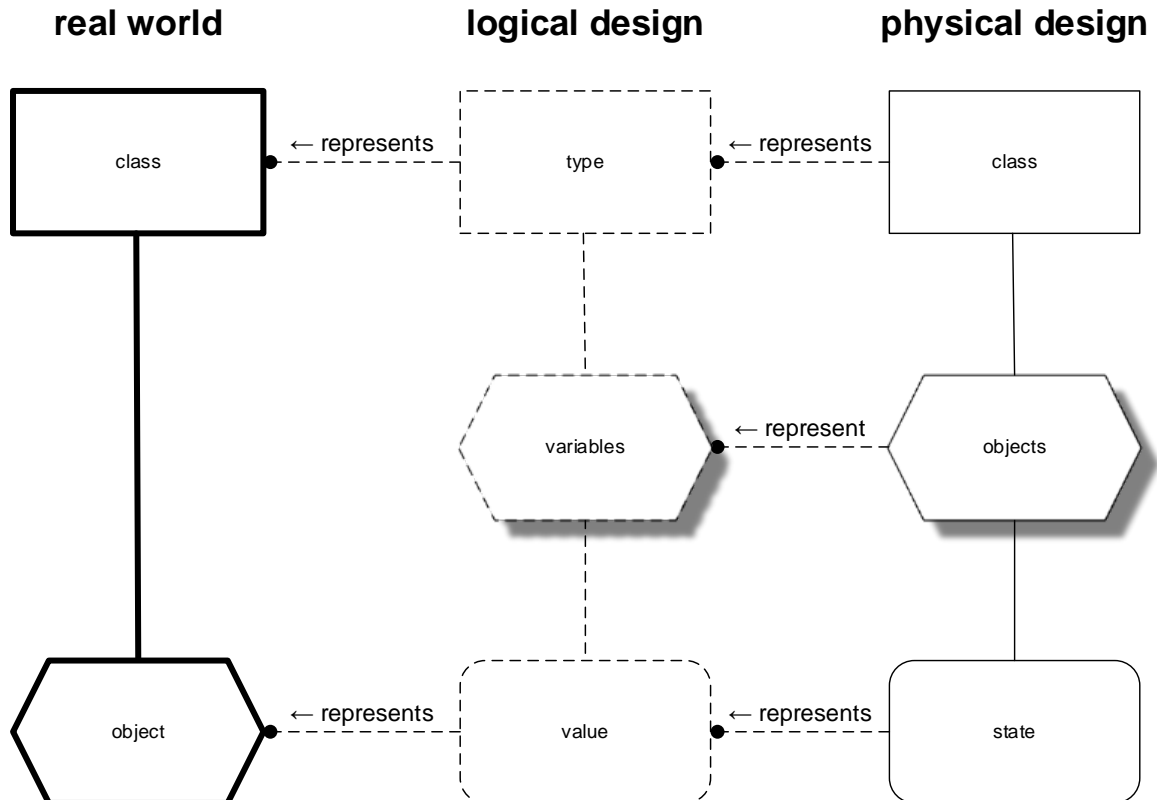


Figure 37. Three Model Levels

## 10 Derivation Relationships

A descriptor—that is, a type or a class—may be defined in terms of another descriptor of the same kind. One descriptor in the relationship is called the **base descriptor**, and the other descriptor, which references it in order to derive its own definition, is called the **derived descriptor**. The derived descriptor often has a special name based on the type of derivation.

### 10.1 Extension Relationship

An **extension** relationship is a relationship between two composite descriptors of the same kind, where one descriptor (the **extending descriptor**) adds components to the other descriptor (the **base descriptor**).

An extension relationship is represented by a line from the extending descriptor to the base of a triangle, and a line from the vertex opposite the base of the triangle to the base descriptor. See Figure 38 below. The wide side of the triangle is towards the descriptor which has more components, namely the extending descriptor. The direction of reference is assumed to be from the extending descriptor to the base descriptor.

An extension relationship can exist between any two composite descriptors of the same kind. The style of the triangle and the relationship lines should follow that of the descriptors, in weight and solid/dashed.

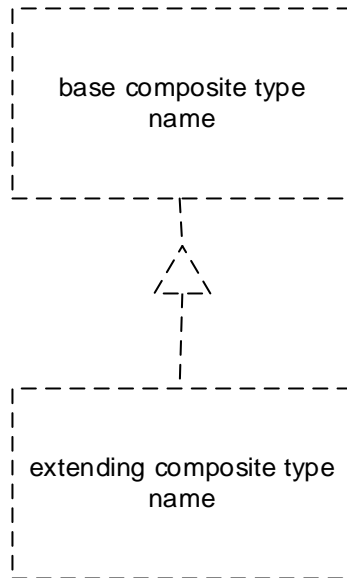


Figure 38. Extension

The inverse of extension is **projection**, where one descriptor (the **projected** descriptor) is defined in terms of a subset of the components of a base descriptor. This is indicated by an arrowhead on the relationship pointing to the base descriptor, reversing the assumed direction of reference. See Figure 39 below. The wide side of the triangle is still towards the descriptor which has more components.

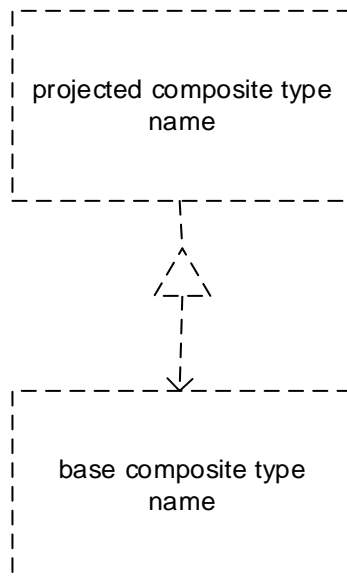


Figure 39. Projection

If more than one composite descriptor extends the same base composite descriptor, and they do not exclude each other, then the relationship lines may pass through the same triangle or there may be a triangle per extending descriptor. See Figure 40 below for an example of the latter convention. The former convention can be useful if a total number of occurrences of the extending descriptors is to be specified on the line near the base descriptor.

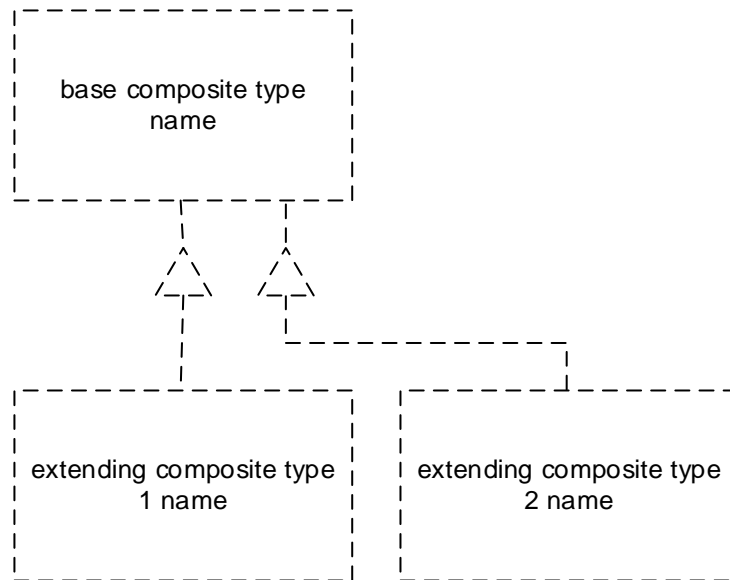


Figure 40. An Example of Two Non-Mutually-Exclusive Extending Relationships

Mutually exclusive extension means that, for a given instance of a base type or class, an instance of only one of the possible exclusively extending types or classes may exist. If there is a set of extending descriptors which are mutually exclusive, then:

- a) the extension relationship lines must pass through the same triangle; and
- b) the triangle must have an X in its center.

See Figure 41 below for an example.

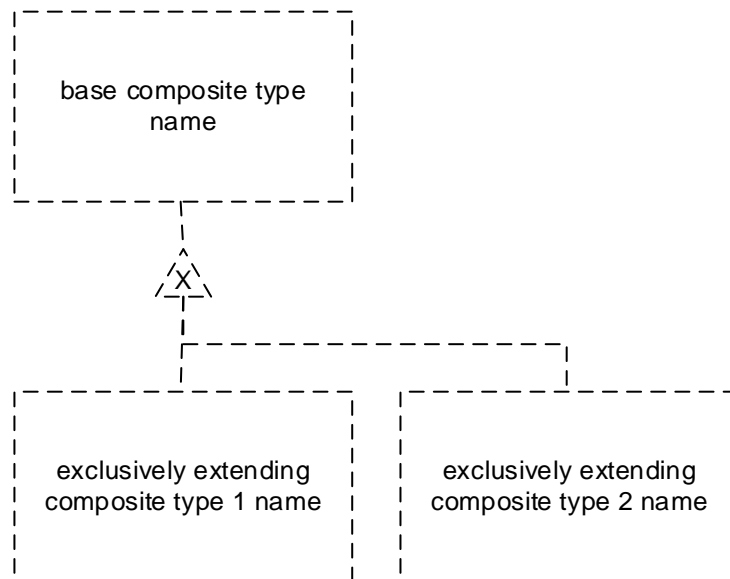


Figure 41. An Example of Two Mutually-Exclusive Extending Relationships

The completeness principle of symbology requires that, if any exclusively extending descriptors are shown on a diagram, then all exclusively extending descriptors must be shown.

## 10.2 Restriction Relationship

A **restriction** relationship is a relationship between two simple or composite descriptors of the same kind, where one descriptor (the **restricting** descriptor) designates fewer values or states than the other descriptor (the **base** descriptor). When a restriction relationship is defined between two types, the restricting type is a **subtype** of the base type, and the base type is the **supertype** of the restricting type.

A restriction relationship is represented by a line from the restricting descriptor to the vertex of an irregular pentagon, and a line from the base of the pentagon to the base descriptor. See Figure 42 below. The wide side of the pentagon is towards the descriptor which designates more values or states, namely the base descriptor. The direction of reference is assumed to be from the restricting descriptor to the base descriptor.

A restriction relationship can exist between any two simple or composite descriptors of the same kind. The style of the pentagon and the relationship lines should follow that of the descriptors, in weight and solid/dashed.

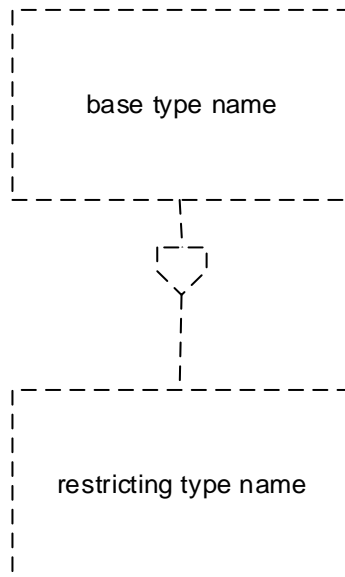


Figure 42. Restriction

The inverse of restriction is **inclusion**, where one descriptor (the **including** descriptor) is defined as having all of the values or states of the base descriptor, plus more. This is indicated by an arrowhead on the relationship pointing to the base descriptor, reversing the assumed direction of reference. See Figure 43 below. The wide side of the pentagon is still towards the descriptor which designates more values or states.



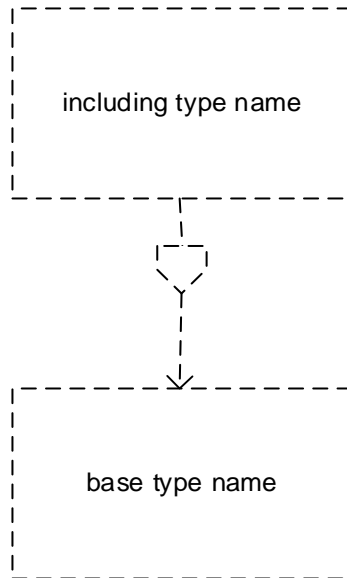


Figure 43. Inclusion

If more than one descriptor restricts the same base descriptor, and they do not exclude each other, then the relationship lines may pass through the same pentagon or there may be a pentagon per restricting descriptor. See Figure 44 below for an example of the latter convention.

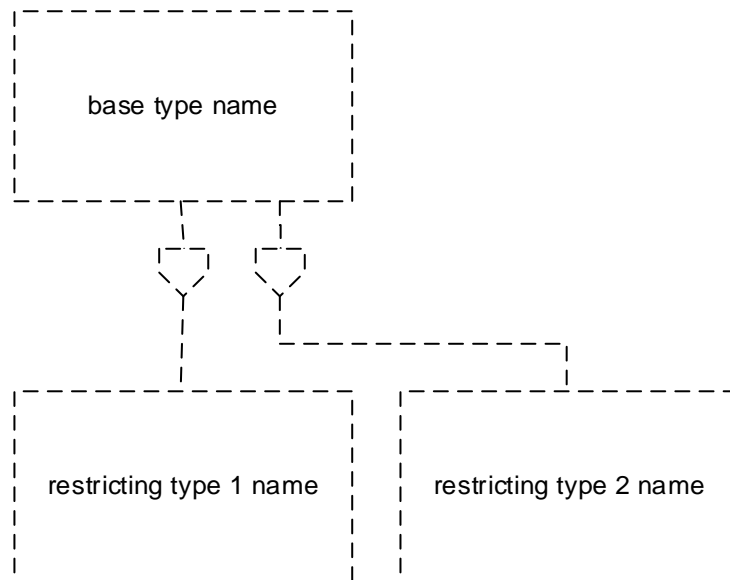


Figure 44. An Example of Two Non-Mutually-Exclusive Restricting Relationships

Mutually exclusive restriction means that the restricting descriptors have no values or states in common. If there is a set of restricting descriptors which are mutually exclusive, then:

- a) the restriction relationship lines must pass through the same pentagon; and
- b) the pentagon must have an X in its center.

See Figure 45 below for an example.

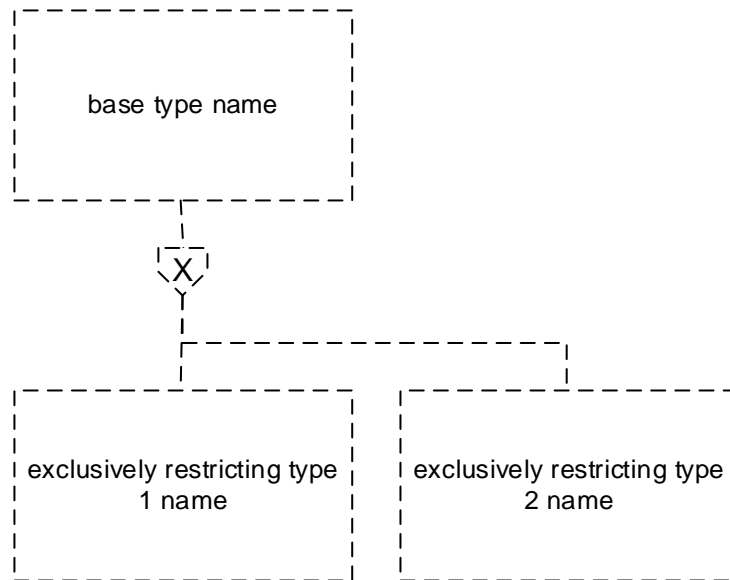


Figure 45. An Example of Two Mutually-Exclusive Restricting Relationships

The completeness principle of symbology requires that, if any exclusively restricting descriptors are shown on a diagram, then all exclusively restricting descriptors must be shown.

## 11 Reference Relationships

Reference without composition is indicated by a relationship line with an open arrowhead. See Figure 46 below for an example of reference between two types. A reference relationship may also exist between two variables, two classes, and two objects. The relationship line may be annotated with the exact nature of the reference.

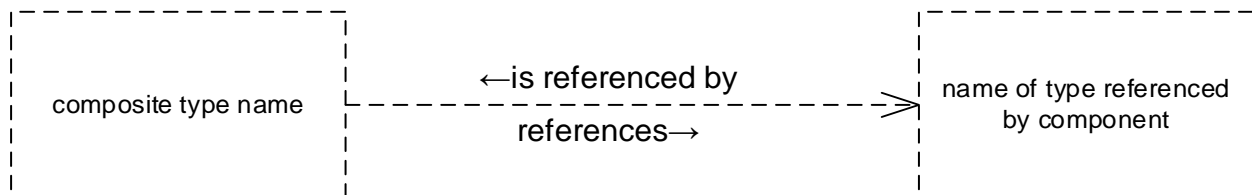


Figure 46. Reference between Two Types

A relational foreign key relationship is indicated by a relationship line.

### 11.1 Containment

Containment is a relationship between an object called a **container** and some number of objects called its **contents**. Containment is always between objects. Containment is by exclusive reference. An X at the referencing end of the relationship line indicates exclusive reference. See Figure 47 below for an example of containment.

A container is not considered to be composed of its contents. Some or all of the contents of a container may be removed from a container, and additional objects may be added to a container, without changing the composition of the container. However, these operations change the *state*

of the container, from empty to non-empty and possibly to full (if the container has a “full” state).

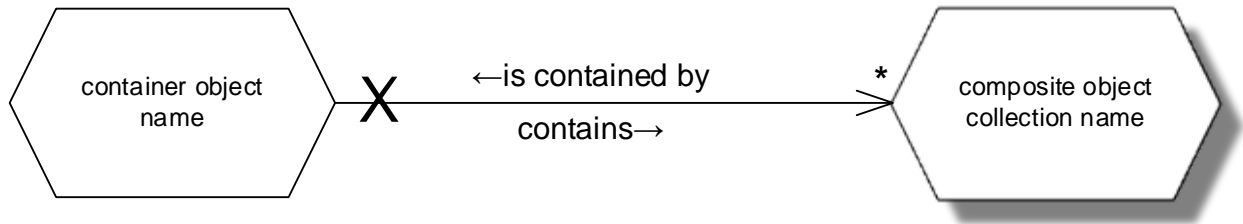


Figure 47. Containment

A class of container objects can also be depicted in terms of the class of objects it may contain. See Figure 48 below for an example of a container class and its content class.

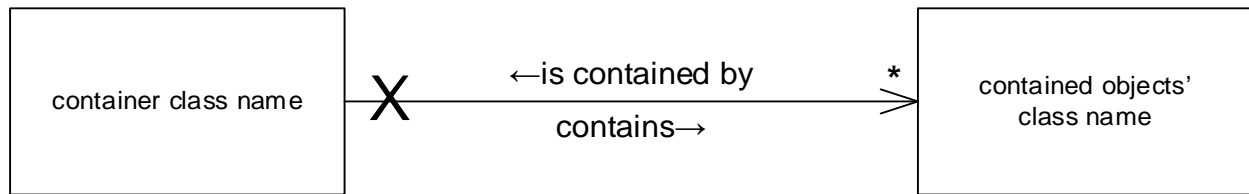


Figure 48. Container Class

## 11.2 Collection

Collection is a concept, though a collection may consist of objects or concepts. A collection has an exclusive relationship to the things in the collection, meaning that they may not simultaneously belong (directly) to more than one collection. See Figure 49 below for an example of a real-world collection of objects.

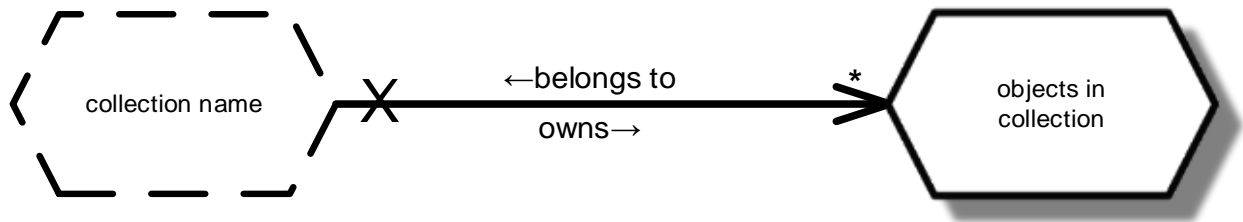


Figure 49. Real-World Collection of Objects

A collection type may be defined in terms of the class or type of its members. See Figure 50 below for an example.

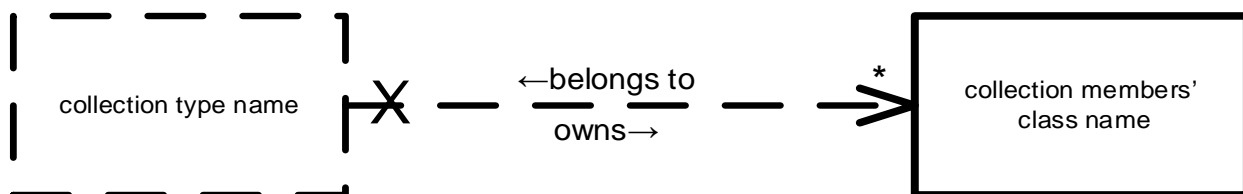


Figure 50. A Collection Type

## 12 Composition Relationships

Composition is indicated by an arrow with a solid arrowhead. Any composite descriptor (a composite type or class) incorporates its components either by aggregation (that is, inline) or by assembly (that is, by reference).

### 12.1 Blending Relationship

Blending is a relationship from a composite to one of its components, such that the integrity of the component is lost. Composition by blending can be illustrated between two types, two variables, two values, two classes, two objects, and two states.

A blending relationship can exist between a composite descriptor and a simple or composite descriptor representing the descriptor of one of its components.

A blending relationship is represented by a line from the composite descriptor to the descriptor of a component, with a solid arrowhead at the component descriptor end. If the composite descriptor is a type, then the line must be dashed. See Figure 52 below for an example of an aggregation relationship between two classes.



Figure 51. Blending between Two Classes

### 12.2 Aggregation Relationship

Aggregation is a relationship from a composite to one of its components, such that the integrity of the component is preserved but it is difficult or impossible to separate the component from the composite. Composition by aggregation can be illustrated between two types, two variables, two values, two classes, two objects, and two states.

An aggregation relationship can exist between a composite descriptor and a simple or composite descriptor representing the descriptor of one of its components. A composite descriptor which specifies that a component be constructed in-line with other components has aggregated the component.

An aggregation relationship is represented by a line from the composite descriptor to the descriptor of a component, with a solid arrowhead at the component descriptor end. If the composite descriptor is a type, then the line must be dashed. See Figure 52 below for an example of an aggregation relationship between two classes.



Figure 52. Aggregation between Two Classes

An aggregation relationship exists between a composite variable or object and one of its component variables or objects when that composite variable or object's descriptor has described aggregation. See Figure 53 below for an example of an aggregation relationship between two objects.



Figure 53. Aggregation between Two Objects

## 12.3 Assembly Relationship

Assembly is a relationship from a composite to one of its components, such that the integrity of the component is preserved and the component may be separated from the composite (which destroys the composite). Composition by assembly can be illustrated between two types, two variables, two classes, and two objects.

An assembly relationship can exist between a composite descriptor and a simple or composite descriptor representing the descriptor of one of its components. A composite descriptor which specifies that its component is to be held by reference has assembled that component into itself. (This is not containment. See the discussion below of containment.)

An assembly relationship is represented by a line from the composite descriptor to the descriptor of a component, with an open circle at the composite descriptor end and a solid arrowhead at the component descriptor end. If the composite descriptor is a type, then the line must be dashed. See Figure 54 below for an example of an assembly relationship between two classes.



Figure 54. Assembly between Two Classes

An assembly relationship exists between a composite variable or object and one of its component variables or objects when the composite variable or object's descriptor has described assembly. See Figure 55 below for an example of an assembly relationship between two objects.

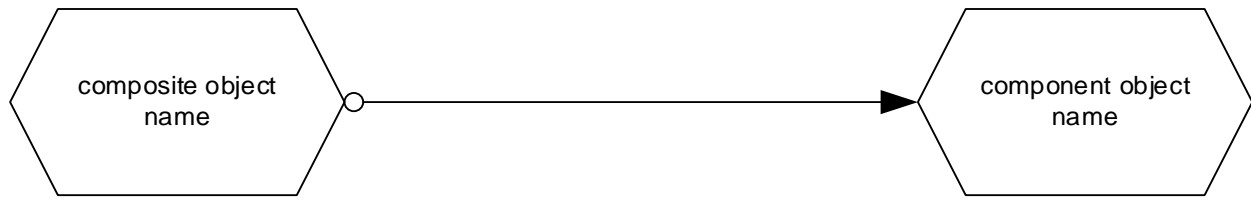


Figure 55. Assembly between Two Objects

## 13 Role Boxes

Role boxes are an alternative way to represent relationships in a model. Role boxes are especially useful when relationships are to be illustrated between the data attributes of a single composite type, where there are often no foreign key relationship lines that may be labeled. See Figure 56 below for an example of their use.

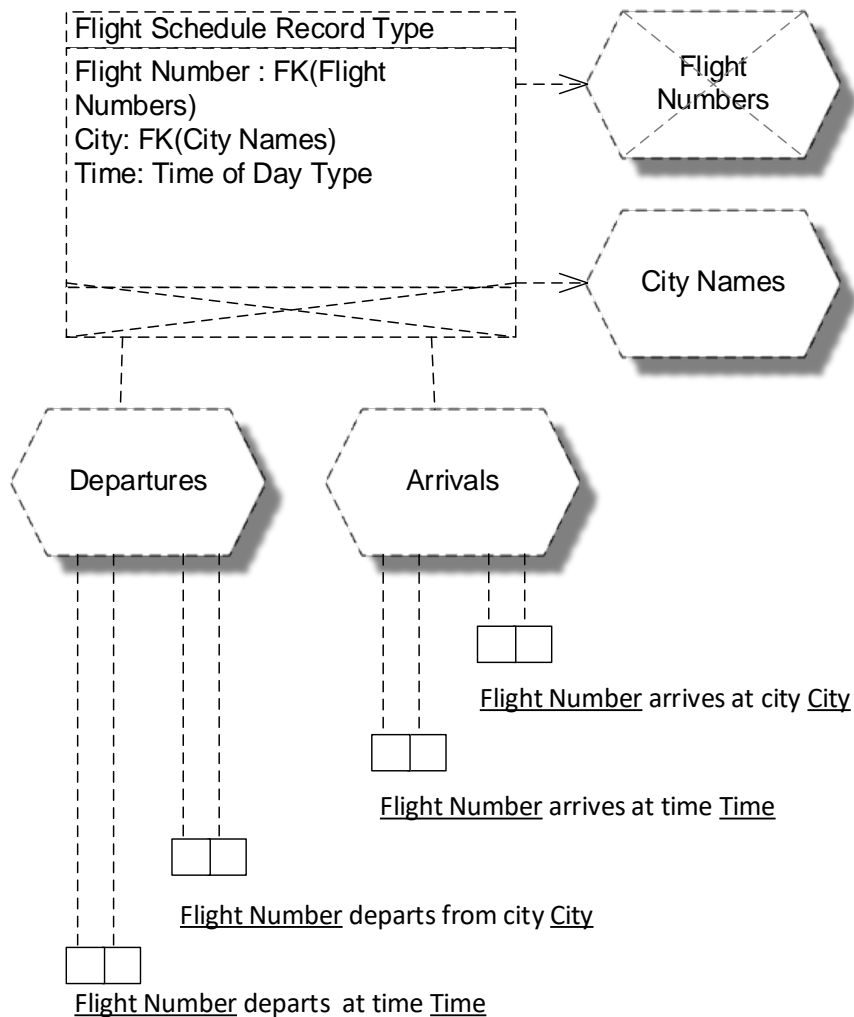


Figure 56. Example of Use of Role Boxes

Each group of adjacent small boxes, called **role boxes**, represents a relationship. The relationship is expressed by a sentence that makes an assertion. The sentence is a predicate. Each of the

predicate's variables is indicated by a name that is underlined. Each small box represents one of the predicate's variables, and is bound to a data attribute of the same name from the composite type (or class) to which the small box is connected by an unadorned line.

Each role box may be connected to only one composite entity (type, class, variable, object, variable collection, or object collection). Although the example in Figure 56 above only shows lines connecting each role box group to a single composite variable collection, role boxes in a single group may be connected to any number of composite entities.

Relationships between any number of data attributes, and predicates having any number of variables, may be represented simply by adding a role box for each data attribute/predicate variable.