

**Semantically Complete Model  
Specification  
(SCM-SPEC-16.10.2005-RU)**

**СЕМАНТИЧЕСКИ ПОЛНАЯ МОДЕЛЬ  
Спецификация**

Vladimir Ovchinnikov ([ovch@lipetsk.ru](mailto:ovch@lipetsk.ru))  
Владимир Овчинников

# ОГЛАВЛЕНИЕ

1	Роль данного документа .....	4
2	Назначение семантически полной модели.....	4
3	Концептуальная спецификация SCM .....	4
3.1	Структура SCM-схемы.....	4
3.1.1	Предметная область .....	5
3.1.2	Понятие.....	8
3.1.3	Ассоциация.....	9
3.1.4	Состояние SCM-схемы.....	11
3.1.4.1	Экземпляр понятия .....	11
3.1.4.2	Экземпляр ассоциации .....	12
3.1.4.3	Состояние схемы .....	14
3.1.4.4	Состояние ассоциации .....	14
3.1.5	Ограничение.....	15
3.1.6	Локальная и единая схемы.....	15
3.1.7	Описанные и порожденные элементы схем.....	16
3.1.8	Типизация понятий.....	17
3.1.8.1	Значение.....	19
3.1.8.1.1	Простые и составные значения .....	19
3.1.8.1.2	Кодирование значений .....	20
3.1.8.1.3	Приведение значений .....	21
3.1.8.2	Метатип .....	21
3.1.8.3	Простой тип.....	22
3.1.8.3.1	Базовый тип .....	22
3.1.8.3.2	Справочник.....	24
3.1.8.3.3	Ограниченный тип.....	26
3.1.8.3.4	Диапазон значений .....	27
3.1.8.3.5	Непрерывный диапазон.....	28
3.1.8.3.6	Дискретный диапазон.....	29
3.1.8.3.7	Диапазон чисел .....	29
3.1.8.3.8	Диапазон целых чисел.....	30
3.1.8.3.9	Диапазон дат .....	30
3.1.8.3.10	Диапазон кодов .....	30
3.1.8.3.11	Строка ограниченной длины .....	31
3.1.8.3.12	Строка по шаблону .....	31
3.1.8.3.13	Набор значений.....	32
3.1.8.3.14	Набор чисел.....	33
3.1.8.3.15	Набор целых чисел .....	33
3.1.8.3.16	Набор дат .....	33
3.1.8.3.17	Набор строк .....	33
3.1.8.3.18	Набор кодов.....	34
3.1.8.3.19	Набор диапазонов .....	34
3.1.8.3.20	Набор непрерывных диапазонов.....	35
3.1.8.3.21	Набор дискретных диапазонов .....	35
3.1.8.3.22	Набор диапазонов чисел .....	36
3.1.8.3.23	Набор диапазонов целых чисел.....	36
3.1.8.3.24	Набор диапазонов дат.....	36
3.1.8.3.25	Набор диапазонов кодов .....	37
3.1.8.4	Составной тип .....	37
3.1.8.4.1	Структура .....	39
3.1.8.4.2	Вектор .....	41

3.1.8.4.3	Множество.....	42
3.1.8.4.4	Ограниченный составной тип.....	43
3.1.8.4.5	Ограниченный вектор .....	44
3.1.8.4.6	Ограниченное множество .....	44
3.1.8.4.7	Набор значений структуры.....	44
3.1.8.4.8	Набор значений вектора.....	45
3.1.8.4.9	Набор значений множества.....	45
3.2	Свойства SCM-схемы.....	46
3.2.1	Прямое использование понятий предметной области.....	46
3.2.2	Свойство семантической полноты.....	46
3.2.3	Отсутствие собственных имен ассоциаций .....	46
3.2.4	Отсутствие деталей реализации.....	46
4	Платформено-независимая спецификация SCM.....	47
4.1	Объектная спецификация SCM-схемы.....	47
4.1.1	Предметная область .....	47
4.1.2	Схема .....	55
4.1.3	Элемент схемы.....	59
4.1.4	Содержимое ассоциации.....	61
4.1.5	Типы.....	64
4.1.5.1	Базовый тип.....	65
4.1.5.2	Справочник.....	65
4.1.5.3	Диапазон чисел .....	67
4.1.5.4	Типы, описываемые целочисленным диапазоном.....	68
4.1.5.5	Диапазон дат .....	68
4.1.5.6	Строка по шаблону .....	69
4.1.5.7	Набор простых значений.....	69
4.1.5.8	Набор диапазонов .....	70
4.1.5.9	Составной тип .....	71
4.1.5.10	Ограниченный составной тип.....	72
4.1.5.11	Набор составных значений .....	73
4.1.6	Составное значение .....	74
4.1.7	Константы .....	75
4.1.8	Итераторы.....	76
4.1.9	Ограничение.....	78
4.1.10	Исключения.....	78
4.2	Транзакционная работа .....	83
4.3	Параллельная работа .....	83
4.4	Управление доступом .....	84
4.5	Многоязыковая и терминологическая поддержка .....	84
4.6	Спецификация SCM в виде IDL.....	84

# 1 Роль данного документа

Данный документ содержит формализацию семантически полной модели (SCM), являющейся ядром семейства тесно взаимосвязанных спецификаций:

- Спецификация текстовой нотации семантически полной модели (SCM-TEXT-SPEC).
- Спецификация графической нотации семантически полной модели (SCM-GRAPH-SPEC).
- Спецификация языка ограничений семантически полной модели (SCM-CL-SPEC).
- Спецификация семантически полного языка запросов (SCM-SCQL-SPEC).
- Спецификация версионной семантически полной модели (SCM-VER-SPEC).

Данный документ представляет собой законченное описание предметной области (см. п. 0) "Модель.SCM". Спецификация SCM осуществляется на двух уровнях – концептуальном (вычислительно-независимом, Computational Independent) и платформо-независимом (Platform Independent) в соответствии с OMG MDA<sup>1</sup>.

## 2 Назначение семантически полной модели

Семантически полная модель представляет собой вычислительно-независимую модель<sup>2</sup> в соответствии с классификацией OMG MDA. SCM используется для решения следующих задач:

- проектирование систем:
  - формализация структуры предметной области на вычислительно-независимом уровне;
  - генерация структуры системы (реляционной схемы, UML диаграмм и т.д.);
- обеспечение функционирования системы семантической интеграции данных:
  - описание глобальной схемы;
  - отображение решений по реализации интегрируемых систем в глобальную схему;
  - исполнение запросов к глобальной схеме;
  - навигация по структуре глобальной схемы;
- формальное структурирование знаний для достижения следующих целей:
  - повышения качества знаний о предметной области;
  - ускорения и повышения качества процесса приобретения знаний;
  - тестирования знаний о структуре предметной области.

## 3 Концептуальная спецификация SCM

### 3.1 Структура SCM-схемы

Здесь и далее для достижения формальной точности изложения идей применяется текстовая нотация SCM, которая вводится в документе SCM-TEXT-SPEC; также применяется язык ограничений, вводимый в отдельной спецификации SCM-CL-SPEC. Если вы не знакомы с SCM и используете данный документ для ее изучения, то в ходе первого прочтения рекомендуется пропускать формальные вставки, обведенные рамками. При последующих прочтениях формальные вставки, с одной стороны, могут использоваться как пример SCM-схемы, с другой стороны, как формально точное определение семантики SCM.

SCM-схема (далее просто схема) строится на элементах следующих видов:

- понятие;
- ассоциация;

<sup>1</sup> OMG Model Driven Architecture Guide V1.0.1 (<http://www.omg.org/cgi-bin/doc?omg/03-06-01>), 2003

<sup>2</sup> Здесь и далее понятие "модель" – инструмент моделирования, а "схема" – результат моделирования.

- ограничение.

#### **Domain: Модель.SCM**

Понятие является Элементом схемы ≡

Ассоциация является Элементом схемы ≡

Ограничение является Элементом схемы ≡

Схема может содержать Элементы схемы

Обоснование. Данного набора элементов достаточно для моделирования любой предметной области (включая динамический аспект). Аналогичный набор элементов можно встретить в любом другом подходе концептуального моделирования, хотя и под другими названиями (понятие соответствует сущности, типу объекта в других подходах; ассоциация – связи, типу факта).

Каждое ограничение входит только в одну схему.

[FOR ALL (Ограничение–Элемент схемы) {COUNT(Элемент схемы–Схема) = 1}  
] <Ограничение должно входить только в одну схему.>

Обоснование. Ограничение настолько сильно зависит от структуры схемы, что разделение ограничений между различными схемами не имеет практического смысла. Поэтому было принято решение каждое ограничение объявлять строго в одной схеме. При этом ограничения с одинаковой структурой, но относящиеся к различным схемам, считаются различными, но структурно эквивалентными друг другу.

### **3.1.1 Предметная область**

Предметная область соответствует некоторой области знания и представляет собой совокупность понятий. Каждое понятие обязательно входит в состав некоторой предметной области.

*Пример. "Задача", "Человек", "Проект" – понятия предметной области "Управление проектами".*

Понятие объявлено в Предметной области →

Обоснование. Одно понятие, в принципе, может использоваться в различных предметных областях, но объявляться оно должно строго в одной предметной области. Это позволяет рассматривать предметные области как контейнеры понятий, каждое из которых объявлено в единственном экземпляре. Другое важное следствие – понятие может уникально идентифицироваться строкой, понятной человеку, и состоящей из двух частей: полное обозначение предметной области и собственное обозначение понятия. Иерархическая организация предметных областей (см. ниже) позволяет объявлять более общие понятия в рамках более общих предметных областей (расположенных ближе к вершине иерархии), а более частные понятия – в рамках более специальных предметных областей (расположенных глубже по иерархии).

Предметная область и SCM-схема не имеют непосредственной связи: они связаны исключительно через понятия. Понятия одной SCM-схемы могут относиться к различным предметным областям, равно как понятия одной предметной области могут использоваться в различных SCM-схемах.

Обоснование. Основное назначение иерархии предметных областей – определить общее пространство понятий и сгруппировать понятия по семантическому критерию. Если бы было принято решение о включении предметной области в схему в качестве собственной ее части, схемы стали бы разрозненными, не связанными между собой. Именно поэтому иерархия предметных областей вводится как отдельный интегрирующий слой, определяющий все возможные понятия. Посредством данного слоя, две различные схемы могут использовать общие понятия. Тем самым SCM-схемы могут рассматриваться как частично интегрированные с момента их создания.

Каждая предметная область имеет некоторое обозначение в виде содержательного словосочетания, соответствующего сути предметной области.

*Пример. Предметную область управления проектами целесообразно обозначить словосочетанием "Управление проектами".*

Предметная область имеет Обозначение →

Обоснование. Иерархия предметных областей является отражением представления людей о реальности, поэтому с каждой предметной областью должно быть ассоциировано интуитивно понятное обозначение. Более того, это обозначение должно быть по крайней мере частью идентификатора (если не полным идентификатором) предметной области, что будет формализовано далее.

Обозначение представляется в виде различных строк, в зависимости от того, какой язык используется для его формулирования, но для каждого языка может быть задана только одна строка обозначения. Существует возможность задания одной строки обозначения для всех языков сразу. В этом случае строка указывается для единственного языка "Все". Язык "Все" служит средством задания строки, используемой по умолчанию для всех тех языков, для которых не задана собственная строка обозначения в явном виде. *Конкретный список возможных языков подлежит уточнению в дальнейшем.*

*Пример. Предметная область управления проектами имеет обозначение "Управление проектами" на русском языке, и обозначение "Project Management" на английском.*

Обозначение определяет Строку для Языка

[Язык = {"Все", "Английский", "Русский", "Немецкий", "Французский", "Испанский", ...}]

[(Обозначение, Строка, Язык): (Обозначение, Язык) → Строка

] <Для каждого языка может быть задана только одна строка в рамках обозначения.>

Обоснование. В существующих подходах концептуального моделирования поддерживаются только два возможных обозначения для одного понятия (сущности, типа объекта) – краткое и развернутое. При этом не конкретизируется используемый язык, а для локализации схемы необходимо создавать другую (никак не связанную с исходной) схему, в которой все обозначения переведены на необходимый язык. Здесь же предлагается заложить возможность локализации концептуальной схемы без ее тиражирования – схема создается в единственном экземпляре и показывается пользователю на том языке, на котором он предпочитает ее видеть. Поэтому в данном подходе любое обозначение представляет собой не просто строку, а структуру, определяющую свою строку для каждого из языков. При этом сохраняется возможность создавать унифицированные обозначения, одинаковым образом представленные для различных языков, путем использования специально введенного для этого языка "Все". Строки, заданные для языка "Все", также используются и в том случае, если обозначение не содержит строки для предпочитаемого пользователем языка в явном виде, что позволяет постепенно наращивать локализацию концептуальной схемы, а не добиваться абсолютной локализации на стадии ее разработки.

Предметные области образуют иерархию по принципу часть-целое, причем каждая вложенная предметная область может входить только в одну главную предметную область. Одна и та же предметная область может одновременно играть как роль вложенной, так и роль главной предметной области, по отношению к разным предметным областям.

*Пример. Предметная область "Управление проектами" может являться вложенной по отношению к предметной области "Менеджмент" и главной по отношению к предметной области "Расчет затрат на проекты".*

Главная предметная область является Предметной областью ≡

Вложенная предметная область является Предметной областью ≡

Вложенная предметная область является частью Главной предметной области →

Иллюстрация. Схематично иерархия предметных областей и понятий в них может быть представлена как совокупность вложенных друг в друга прямоугольников – предметных областей – с возможностью объявления в их рамках понятий – окружностей (см. рисунок).

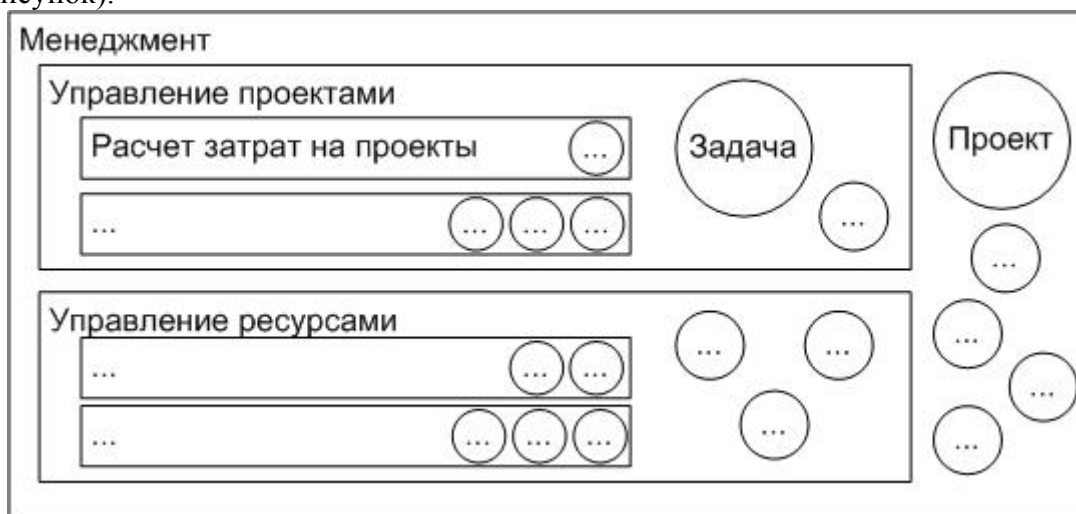


Рисунок 1 Схематичная иллюстрация иерархии предметных областей и понятий в них

Обоснование. Иерархия является простой и понятной структурой, позволяющей рассматривать главные предметные области как контейнеры, содержащие вложенные предметные области. Также иерархия позволяет использовать надежный и понятный способ идентификации предметных областей, когда идентификатор отдельной предметной области представляет собой последовательность обозначений предметных областей, лежащих на пути от корневой ПО к данной (см. ниже). Более сложные способы организации предметных областей, например с возможностью подчинения одной вложенной предметной области нескольким главным, неизбежно приводят к усложнению способа идентификации предметных областей, затрудняют их поиск, требуют больших усилий по администрированию и менее прозрачны для пользователей.

Каждая предметная область характеризуется полным обозначением, уникальным среди всех предметных областей. Полное обозначение формируется как конкатенация через точку в рамках каждого языка: а) обозначений вышестоящих предметных областей, б) обозначения данной предметной области. Конкатенация осуществляется в направлении от корня дерева предметных областей к данной предметной области. Существует только одна корневая предметная область. Обозначением корневой предметной области на всех языках является пустая строка "".

*Пример. Предметная область "Расчет затрат на проекты" имеет полное обозначение "Менеджмент.Управление Проектами.Расчет затрат на проекты", если рассматривать предметную область "Менеджмент" как непосредственно вложенную в корневую.*

Корневая предметная область является Предметной областью  $\equiv$   
 [(Корневая предметная область–Предметная область) EQUALS  
 ((Предметная область) MINUS  
 (Предметная область–Вложенная предметная область–Главная предметная область)  
 )] <Предметная область является корневой, если не имеет главных предметных  
 областей.>  
 [COUNT(Корневая предметная область) = 1] <Существует только одна корневая  
 предметная область.>  
 [FOR ALL (Корневая предметная область–Предметная область–(Обозначение,  
 Строка, Язык)) EXISTS {Строка = ""}  
 ] <Обозначение корневой предметной области на всех языках – пустая строка.>  
Полное обозначение является Обозначением  $\equiv$

Предметная область идентифицируется Полным обозначением  $\equiv$   
 [(Предметная область–Полное обозначение–(Обозначение, Строка, Язык))  
 EQUALS  
 ((SELECT Предметная область, Язык  
   (IF EXISTS (Предметная область(главная)–Корневая предметная область)  
     THEN Обозначение  
     ELSE Строка(главная) + '!' + Строка) => Строка  
 FROM ((Строка, Язык, Обозначение)–Предметная область–  
   Вложенная предметная область–Главная предметная область–  
   Предметная область(главная)–Полное обозначение(главное)–  
   (Обозначение(главное), Строка(главная), Язык))  
 UNION  
 SELECT Предметная область, Язык, Строка  
 FROM ((Строка, Язык, Обозначение)–Предметная область–Корневая предметная  
 область))  
 ] <Полное обозначение некорневой предметной области представляет собой  
 конкатенацию обозначений предметных областей через точку, начиная от корневой  
 предметной области до данной (в рамках каждого языка). Если данная предметная  
 область является корневой или вложенной в корневую, то ее полное обозначение  
 равно собственному обозначению.>  
 (((Предметная область–Полное обозначение–(Обозначение, Язык, Строка)),  
 (Предметная область–Вложенная предметная область–Главная предметная  
 область)+):  
 Предметная область  $\equiv$ (Язык, Главная предметная область) $\equiv$  Строка  
 ] <Строки полных обозначений предметных областей уникальны для каждой  
 предметной области в рамках каждого языка и вышестоящей предметной области,  
 если она есть.>

Обоснование. Использование конкатенации обозначений в качестве идентификатора предметной области позволяет охватить в идентификаторе весь путь от корня иерархии до конкретной предметной области. Это упрощает поиск предметной области и для человека, и алгоритмически. Более того, фактически, каждая предметная область имеет набор строковых идентификаторов – свой идентификатор для каждого языка. Каждый такой идентификатор уникален лишь в рамках соответствующего ему языка, – допускается использование одного и того же строкового идентификатора для различных понятий, но в рамках различных языков. Это позволяет достичь высокой гибкости локализации – локализация может разрабатываться отдельно для каждого из языков. Так как корневая предметная область является частью пути к любой из предметных областей, то в качестве ее обозначения выбрана пустая строка, что позволяет сократить полные обозначения за счет удаления общей повторяющейся части.

### 3.1.2 Понятие

Понятие – термин предметной области, используемый в работе ее экспертами. Каждое понятие имеет обозначение, представляемое в виде содержательного словосочетания на различных языках.

*Пример. Словосочетание из одного слова "Задача" является обозначением понятия "задача" на русском языке. Это понятие используется в работе экспертами предметной области управления проектами.*

**Понятие** имеет Обозначение →

Каждое понятие имеет уникальное среди всех существующих понятий полное обозначение, представляющее собой конкатенацию полного обозначения предметной области и собственного обозначения понятия через точку в рамках каждого языка.

*Пример. Понятие "Задача", определенное в рамках предметной области "Менеджмент. Управление проектами" будет иметь полное обозначение "Менеджмент. Управление проектами. Задача". При этом верно, что не существует другого понятия, которое бы имело то же полное обозначение на русском языке.*

Понятие идентифицируется Полным обозначением  $\equiv$   
 [(Понятие–Полное обозначение–(Обозначение, Язык, Строка)) EQUALS  
 (SELECT Понятие, Язык, Строка(ПО) + '.' + Строка(понятия) => Строка  
 FROM ((Язык, Строка(понятия), Обозначение(понятия))–Понятие–Предметная  
 область–Полное обозначение(ПО)–(Обозначение(ПО), Строка(ПО), Язык)))  
 ] <Полное обозначение понятия – конкатенация полного обозначения  
 соответствующей предметной области и обозначения понятия через точку (в  
 рамках каждого языка).>  
 [(Понятие–Полное обозначение–(Обозначение, Строка, Язык)):  
 Понятие  $\equiv$ (Язык) $\equiv$  Строка  
 ] <Понятия имеют уникальные полные обозначения в рамках каждого языка.>

Обоснование. Количество понятий, объявленных в предметных областях, может быть очень значительным. В этих условиях уникальность обозначений понятий проще обеспечивать, если в полное обозначение понятия включено как часть полное обозначение той предметной области, к которой оно относится.

Понятия не могут включаться в схему самостоятельно: понятие считается включенным в схему, если в нее входит хотя бы одна ассоциация, построенная на данном понятии (см. 0).

[SELECT Схема, Понятие FROM (Схема–Элемент схемы–Понятие) EQUALS  
 SELECT Схема, Понятие FROM (Схема–Элемент схемы–Ассоциация–Понятие)  
 ] <Понятие включено в схему, если в данную схему включена хотя бы одна  
 ассоциация, построенная на данном понятии.>

Обоснование. Включение понятия в схему без включения в ту же схему хотя бы одной ассоциации, построенной на этом понятии, не несет никакой дополнительной информации: из факта включения понятия в схему сложно сделать какие-либо полезные выводы и как-либо использовать эту информацию. В результате самостоятельное включение понятий в схемы запрещено.

### 3.1.3 Ассоциация

Ассоциация служит для связывания понятий и представляет собой *множество* связываемых понятий (не мультимножество и не последовательность), состоящее из двух или более понятий. Каждая ассоциация построена на уникальном множестве понятий, то есть множество понятий является идентификатором ассоциации, – зная множество понятий можно однозначно определить соответствующую ему ассоциацию.

*Пример. Запрос "получить содержимое ассоциации, построенной на понятиях Человек и Задача" имеет строгую семантику и ссылается на совершенно определенную ассоциацию.*

Ассоциация связывает множество Понятий  $\Leftarrow$   
 [FOR ALL (Ассоциация) EXISTS (Ассоциация, COUNT(Понятие)>1)  
 ] <Каждая ассоциация связывает два или более понятий.>

Ассоциация не имеет собственного обозначения. Ссылку на ассоциацию можно сделать посредством перечисления обозначений входящих в нее понятий в любом порядке. Стандартный синтаксис такой ссылки является частью SCQL (см. SCM-SCQL-SPEC) и его формализация отсутствует в данной спецификации. В общих словах, такая ссылка представляет собой перечисление понятий (точнее, их обозначений), входящих в ассоциацию, через запятую в круглых скобках в любом порядке.

*Пример. Перечисления понятий (Человек, Задача) и (Задача, Человек) ссылаются на одну и ту же ассоциацию.*

Обоснование. Использование множеств понятий в качестве идентификаторов ассоциаций является одним из ключевых отличий предлагаемого подхода от других техник концептуального моделирования. Это позволяет избавить пользователей SCM от необходимости запоминать собственные обозначения ассоциаций и создает предпосылки для семантически полного языка запросов (SCQL), обладающего особыми свойствами (см. SCM-SCQL-SPEC).

Ассоциация может входить в произвольное количество схем (и быть по-разному ограниченной в каждой из них, см. 3.1.5). Понятия, входящие в одну ассоциацию, считаются связанными в контексте тех схем, в которые входит ассоциация.

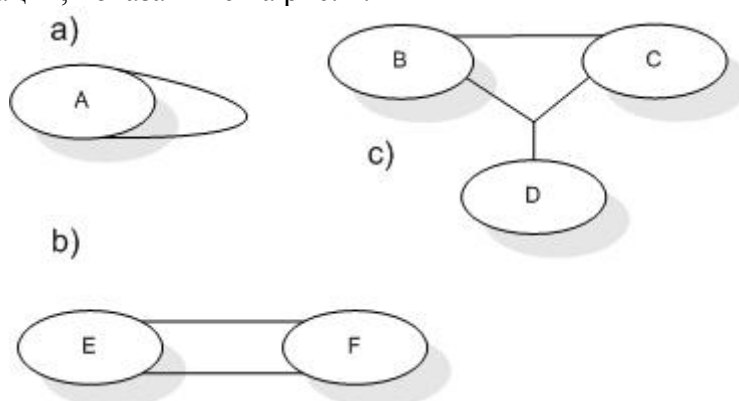
*Пример. Понятия Человек и Задача могут быть связанным в контексте одной схемы, и быть не связанными в контексте других схем.*

В рамках схемы не может существовать двух ассоциаций, построенных на множествах понятий, одно из которых является собственным подмножеством другого. Данное ограничение носит название "свойство семантической полноты" и его обоснование приводится в п. 3.2.2.

*Пример. Если в некоторой схеме есть ассоциация (Человек, Задача), то в этой схеме не может существовать других ассоциаций, построенных на обоих этих понятиях одновременно. Так, например, в этой схеме не может существовать ассоциация (Человек, Задача, Проект), а ассоциация (Человек, Проект) – может.*

```
[FOR ALL (Ассоциация(первая)×Ассоциация(вторая)) {
IF EXISTS (Ассоциация(первая)–Элемент схемы(первый)–Схема–Элемент
схемы(второй)–Ассоциация(вторая)) THEN EXISTS
      ((Ассоциация(первая), SET(Понятие)(первое)),
      (Ассоциация(вторая), SET(Понятие)(второе)),
      (SET(Понятие)(первое) NOT SUBSET OF SET(Понятие)(второе)) ) }
] <В рамках схемы не может существовать двух ассоциаций, связывающих
множества понятий, одно из которых является подмножеством другого.>
```

Иллюстрация. Обозначим понятия овалами, а ассоциации – соединительными линиями между овалами. Описанное ограничение семантической полноты запрещает объявлять ассоциации, показанные на рис. 2.



**Рисунок 2 Запрещенные в SCM способы объявления ассоциаций**

Но это не означает, что данные виды связей не могут быть описаны средствами SCM. Для их корректного описания в SCM необходимо в каждом из случаев одно из понятий классифицировать по соответствующему критерию. Под классификацией понимается введение дополнительных понятий, являющихся частным случаем исходного понятия. При этом вновь добавленные понятия связываются с исходным при помощи бинарной эквивалентной ассоциации, зачастую называемой ассоциацией наследования. На рис. 3 показан преобразованный вариант рис. 2, в котором все ассоциации удовлетворяют

ограничению семантической полноты. На этом рисунке ассоциации наследования обозначены полыми стрелками. Обозначения исходных понятий сохранены для обеспечения возможности сравнения результата преобразования с рис. 2. В случаях (b) и (c) классификации могли быть подвергнуты понятия "С" и "F" вместо понятий "В" и "Е" соответственно.

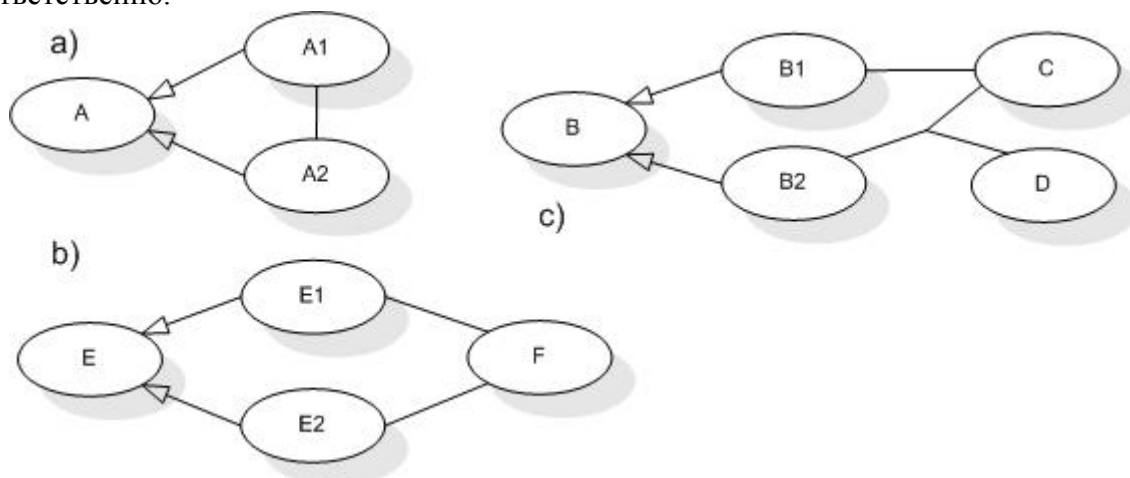


Рисунок 3 Допустимые в SCM способы объявления ассоциаций

Пример. Рассмотрим конкретный пример для случая (a). Здесь понятие "А" может соответствовать понятию "Задача", а два вновь добавленных понятия – "Главной задаче" и "Вложенной задаче". Таким образом, смысл добавленных ассоциаций наследования будет следующим: "Главная задача является Задачей" и "Вложенная задача является Задачей". При этом смысл исходной ассоциации может быть следующим "Вложенная задача является частью Главной задачи".

### 3.1.4 Состояние SCM-схемы

SCM-схема как таковая формализует знание о структуре предметных областей. Знание о состоянии предметных областей выражается в состоянии схемы.

*Пример. Знание о существовании ассоциации (Человек, Задача), показывающей назначение людей на задачи в качестве исполнителей, – знание о структуре предметной области "Управление проектами". Знание о том, что "Иван Петров" является исполнителем задачи "Разработка спецификации", – знание о состоянии этой предметной области.*

Состояние схемы может изменяться во времени в результате действий пользователей и программ. Состояние схемы, актуальное на данный момент, считается текущим состоянием схемы. Именно с текущим состоянием схемы работают все выражения SCQL.

#### 3.1.4.1 Экземпляр понятия

Каждое понятие подразумевает совокупность объектов/явлений реальности, подпадающих под это понятие.

*Пример. Понятие "Автомобиль" подразумевает совокупность всех объектов реальности, которые соответствуют определению автомобиля.*

Один и тот же объект/явление реальности может соответствовать различным понятиям.

*Пример. Конкретный объект реальности автобус "А001СР48" соответствует как понятию "Автобус", так и понятию "Автомобиль".*

Соответствие объекта/явления понятию назовем экземпляром понятия. Таким образом, каждый экземпляр понятия говорит о соответствии некоторого объекта/явления некоторому понятию.

*Пример. В рамках предыдущего примера определены два различных экземпляра понятий: "Автобус: A001CP48" и "Автомобиль: A001CP48", соответствующих одному объекту реальности, но разным понятиям.*

Экземпляр понятия соответствует Объекту →

Понятие может принимать бесконечное число экземпляров, каждый из которых должен относиться к своему объекту/явлению.

*Пример. Один и тот же автобус не может быть представлен в виде двух различных экземпляров одного понятия "Автобус".*

Экземпляр понятия относится к Понятию →

[(Явление–Экземпляр понятия–Понятие):

Явление ≡ (Понятие) ≡ Экземпляр понятия] <Каждое явление может обозначать не более одного экземпляра в рамках каждого понятия и наоборот.>

Иллюстрация. Обозначим затемненными овалами объекты реальности, а белыми овалами – экземпляры понятий. Тогда соответствие экземпляров понятий "Автомобиль" и "Автобус" объектам реальности можно проиллюстрировать, как это показано на рис. 4. В центре рисунка перечисляются все существующие объекты/явления, начиная с интересующих нас автобусов и автомобилей. Некоторые из данных объектов являются автомобилями, и для таких объектов объявлены соответствующие экземпляры понятия "Автомобиль". Некоторые из объектов, являющихся автомобилями, также являются автобусами, поэтому для этих объектов объявлены экземпляры понятия "Автобус". Для тех же из объектов, которые не являются ни автобусом, ни автомобилем, не объявлены экземпляры ни того, ни другого понятия.

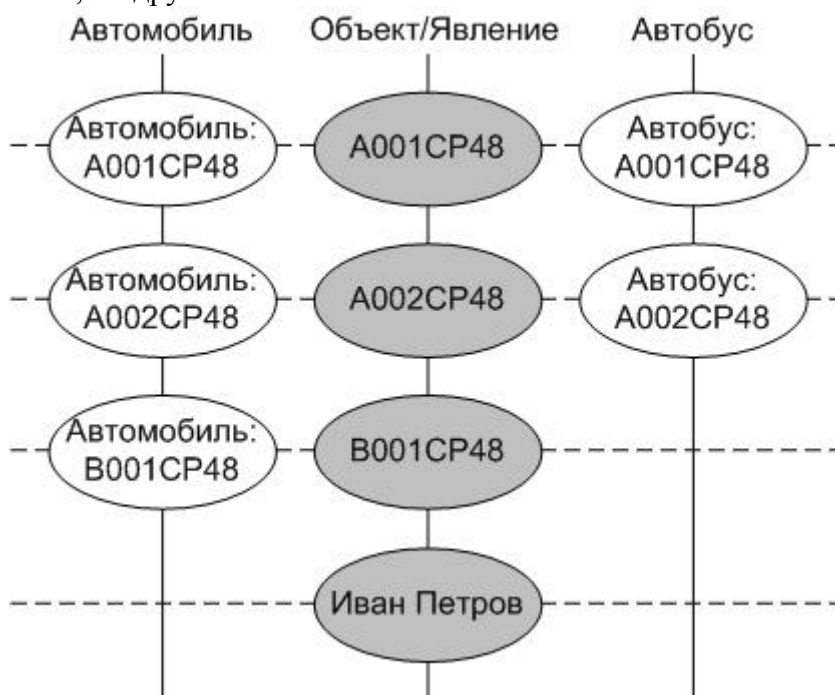


Рисунок 4 Соответствие экземпляров понятий "Автомобиль" и "Автобус" конкретным объектам реальности

### 3.1.4.2 Экземпляр ассоциации

Экземпляр ассоциации является атомарным фактом взаимосвязи экземпляров понятий и представляет собой *множество* (не мультимножество и не последовательность) из двух или более связываемых экземпляров понятий.

*Пример. Экземпляр ассоциации (Человек: "Иван Петров", Задача: "Разработка спецификации") – множество из двух связываемых экземпляров понятий.*

Экземпляр ассоциации связывает Экземпляры понятий ←

```
[FOR ALL (Экземпляр ассоциации) EXISTS
(Экземпляр ассоциации, COUNT(Экземпляр понятия)>1)
] <Каждый экземпляр ассоциации связывает два или более экземпляра понятий.>
```

Экземпляр ассоциации принадлежит той ассоциации, которая построена на том же множестве понятий, что и сам экземпляр.

*Пример. Экземпляр ассоциации (Человек: "Иван Петров", Задача: "Разработка спецификации") принадлежит ассоциации (Человек, Задача).*

Иллюстрация. Обозначим утолщенной сплошной линией ассоциации между понятиями, утолщенной пунктирной линией – экземпляры ассоциаций, овалами – экземпляры понятий. Тогда примеры экземпляров ассоциаций "Водитель управляет Автомобилем" и "Автобус является Автомобилем" могут быть представлены так, как это показано на рис. 5. Из рисунка следует, что автомобиль А001СР48 также является автобусом и управляется водителем Иваном Петровым. Автомобиль В001СР48 в свою очередь не является автобусом и управляется водителем Сергеем Леви. При этом оба перечисленных водителя могут управлять автомобилем А002СР48, который является автобусом.

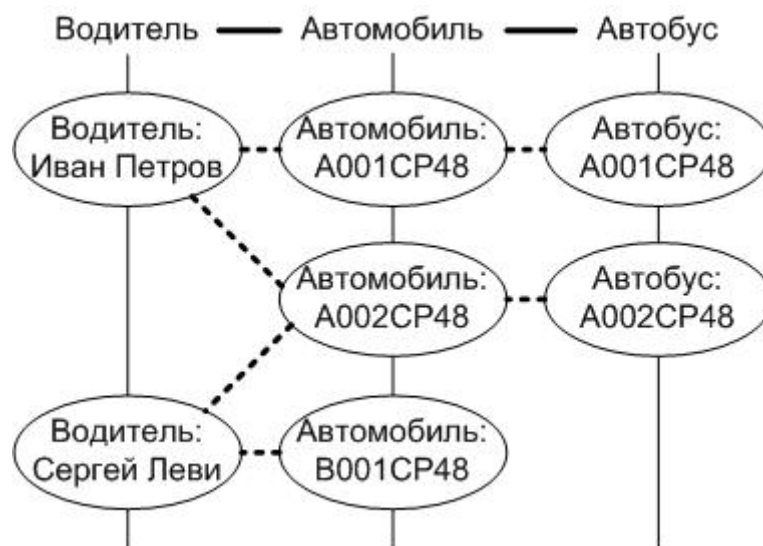


Рисунок 5 Экземпляры ассоциаций (Водитель, Автомобиль) и (Автомобиль, Автобус)

Понятия не могут принимать неопределенных значений (**nil**-значений, **null**-значений) в рамках экземпляров ассоциаций, так как тот же эффект в данном подходе достигается простым отсутствием таких экземпляров ассоциаций.

*Пример. Вместо объявления экземпляра ассоциации (Человек: "Иван Петров", Задача: **nil**) необходимо отказаться от создания данного экземпляра ассоциации вовсе.*

Экземпляр ассоциации не может существовать без соответствующей ассоциации, поэтому для каждого экземпляра ассоциации должна обязательно существовать ассоциация, которой он принадлежит.

```
Экземпляр ассоциации принадлежит Ассоциации →
[(Экземпляр ассоциации, Ассоциация) EQUALS
((Экземпляр ассоциации × Ассоциация),
(SELECT Экземпляр ассоциации, SET(Понятие)(экземпляра)
FROM (Экземпляр ассоциации–Экземпляр понятия–Понятие)),
(Ассоциация, SET(Понятие)(ассоциации)),
(SET(Понятие)(экземпляра) = SET(Понятие)(ассоциации))
)] <Каждый экземпляр принадлежит той ассоциации, которая связывает то же
множество понятий.>
```

Обоснование. Ассоциации служат для декларирования принципиальной возможности существования соответствующих экземпляров ассоциаций. Если

предположить, что экземпляр ассоциации может существовать без соответствующей ассоциации, то в этом случае ассоциации потеряют свою основную функцию. Поэтому каждый экземпляр ассоциации должен принадлежать определенной ранее объявленной ассоциации.

### 3.1.4.3 Состояние схемы

Состояние схемы относится строго к одной схеме и представляет собой множество экземпляров ассоциаций. Каждый из этих экземпляров должен относиться к ассоциации, включенной в данную схему. Таким образом, схема должна включать в себя все те ассоциации, для которых существуют экземпляры в ее состоянии (плюс ассоциации с пустым состоянием).

*Пример. Если состояние схемы включает экземпляр ассоциации (Человек: "Иван Петров", Задача: "Разработка спецификации"), то сама схема должна содержать ассоциацию (Человек, Задача).*

Состояние схемы относится к Схеме →  
Состояние схемы состоит из Экземпляров ассоциаций  
[(Схема–Состояние схемы–Экземпляр ассоциации–Ассоциация) PART OF  
(Схема–Элемент схемы–Ассоциация)  
] <Схема включает в себя все те ассоциации, для которых существуют экземпляры ассоциаций в ее состояниях.>

Иллюстрация. В качестве иллюстрации состояния схемы может использовать рис. Рисунок 5. Здесь состоянием схемы является совокупность всех экземпляров ассоциаций, обозначенных пунктирными линиями.

### 3.1.4.4 Состояние ассоциации

Состояние ассоциации представляет собой множество экземпляров ассоциаций, принадлежащих одной ассоциации. Состояние ассоциации относится к той же ассоциации, к которой принадлежат все его экземпляры ассоциаций.

Состояние ассоциации относится к Ассоциации →  
Состояние ассоциации содержит Экземпляры ассоциации ←  
[(Состояние ассоциации–Экземпляр ассоциации–Ассоциация) PART OF  
(Состояние ассоциации–Ассоциация)] <Все экземпляры состояния ассоциации принадлежат той же ассоциации, что и само состояние.>

Иллюстрация. Рассмотрим рис. 5. Здесь в качестве состояния ассоциации можно рассматривать группу экземпляров ассоциаций (представленных пунктирными линиями), расположенных строго под одной из ассоциаций (представленных сплошными горизонтальными линиями). Например, экземпляр ассоциации (Автомобиль: "A002CP48", Автобус: "A002CP48") не относится к ассоциации (Водитель, Автомобиль), а экземпляр ассоциации (Водитель: "Сергей Леви", Автомобиль: "B001CP48") – относится.

Обоснование. В рамках формализации сказано, что не может существовать двух состояний, состоящих из одного и того же множества экземпляров ассоциаций. Такое решение было принято по той причине, что два состояния, состоящие из одного и того же множества экземпляров ассоциаций, с семантической точки зрения не отличаются. Не имеет смысла рассматривать два состояния ассоциации как различающиеся, если они содержат одно и то же множество экземпляров ассоциаций.

Обоснование. В рамках формализации сказано, что состояние ассоциации должно обязательно состоять из хотя бы одного экземпляра ассоциации. Действительно, если допустить существование состояния ассоциации без содержащихся в нем экземпляров, то такое состояние семантически ничем не отличается, если бы мы решили, что данного состояния просто не существует. Для исключения неоднозначности интерпретации

пустого состояния принято решение рассматривать такие состояния ассоциаций как несуществующие.

*Пример. Сразу после объявления существования ассоциации (Человек, Задача), состояние данной ассоциации в рамках соответствующего состояния схемы будет пустым, то есть состояние схемы не будет содержать никакого состояния данной ассоциации.*

Состояние схемы можно представить как множество состояний ассоциаций. При этом каждая ассоциация принимает то состояние, которое получается группировкой всех экземпляров данной ассоциации, встречающихся в данном состоянии схемы.

*Пример. Экземпляр ассоциации (Человек: "Иван Петров", Задача: "Разработка спецификации") является частью состояния ассоциации (Человек, Задача) в одном состоянии схемы, и не является его частью в другом состоянии схемы, если он входит в первое состояние схемы, но не входит во второе.*

Состояние схемы состоит из Состояний ассоциаций  
[(Состояние схемы–Состояние ассоциации–SET(Экземпляр ассоциации)) EQUALS  
(Состояние схемы–SET(Экземпляр ассоциации)–Ассоциация)  
] <Состояние схемы состоит из тех состояний ассоциаций, которые построены на  
множестве экземпляров, содержащихся в данном состоянии схемы.>

### 3.1.5 Ограничение

Не любое из состояний схемы допустимо с точки зрения семантики предметной области. Чтобы ограничить множества допустимых состояний схемы в нее [схему] включаются дополнительные элементы – ограничения. Каждое ограничение представляет собой утверждение о состоянии схемы, которое должно выполняться для всех допустимых состояний. Ограничения формулируются с использованием специального языка ограничений. Семантика языка ограничений SCM является предметом отдельной спецификации – SCM-CL-SPEC.

Ограничения могут иметь собственные обозначения, уникально идентифицирующие их в рамках соответствующих схем, и представляющие собой понятные пользователю строковые названия на тех языках, которые могут потребоваться для общения с пользователями.

Ограничение может иметь идентифицирующее Обозначение  $\equiv$   
[(Схема–Элемент схемы–Ограничение–(Обозначение, Строка, Язык)):  
Понятие  $\equiv$ (Язык, Схема) $\equiv$  Строка  
] <Ограничения имеют уникальные обозначения в рамках каждого языка и той  
схемы, в которую данное ограничение включено.>

Пояснение. К тем ограничениям, которые не имеют уникальных обозначений, существует единственный способ доступа – по их структуре. Собственные обозначения ограничений полезны в тех случаях, когда необходимо организовать дружественный для пользователей SCM способ ссылки на ограничения.

### 3.1.6 Локальная и единая схемы

Локальная схема является схемой, расположенной в некоторой предметной области и подразумевает ее текущее состояние, доступное в рамках данной предметной области в настоящий момент времени.

Локальная схема является Схемой  $\equiv$   
Локальная схема находится в Предметной области  $\rightarrow$   
Локальная схема подразумевает текущее Состояние схемы  $\equiv$   
[(Локальная схема, Схема) EQUALS (Локальная схема–Состояние схемы–Схема)  
] <Локальная схема является той схемой, состояние которой она подразумевает.>

Вывод. Таким образом, предметная область является основным входом в пространство понятий и схем SCM. Без такого входа данная спецификация была бы не полной, так как было бы не ясно в каком пространстве существуют схемы и их состояния, как к ним получить доступ. Теперь достаточно получить доступ к корневой предметной области, чтобы иметь возможность перейти ко всем ее вложенным предметным областям, их понятиям и схемам, а через схемы к ассоциациям, понятиям и ограничениям.

Локальная схема имеет собственное многоязыковое обозначение, представляющее собой ее содержательное описание.

Локальная схема имеет Обозначение →

В рамках каждого из языков локальная схема имеет уникальное среди всех существующих локальных схем полное обозначение, представляющее собой конкатенацию полного обозначения предметной области и собственного обозначения локальной схемы через точку.

Локальная схема идентифицируется Полным обозначением ≡

[(Локальная схема–Полное обозначение–(Обозначение, Язык, Строка)) EQUALS (SELECT Локальная схема, Язык, Строка(ПО) + '!' + Строка(схемы) => Строка FROM ((Язык, Строка(схемы), Обозначение(схемы))–Локальная схема–Предметная область–Полное обозначение(ПО)–(Обозначение(ПО), Строка(ПО), Язык))) ] <Полное обозначение локальной схемы – конкатенация полного обозначения соответствующей предметной области и обозначения локальной схемы через точку (в рамках каждого языка).>  
 [(Локальная схема–Полное обозначение–(Обозначение, Строка, Язык)):  
 Локальная схема ≡(Язык)≡ Строка  
 ] <Локальные схемы имеют уникальные полные обозначения в рамках каждого языка.>

Пояснение. Таким образом, полное обозначение локальной схемы может использоваться для ссылки на нее и поиска в рамках предметных областей.

Особую роль играет единая схема, существующая в единственном экземпляре и представляющая собой формализацию текущих представлений об актуальном состоянии реальности, которое согласованно с экспертами в соответствующих предметных областях и интегрировано в единую схему (включая состояние схемы). Единая схема предназначена для формализации общепринятых представлений о реальности, тогда как локальные схемы служат для целей разработки новых, еще не согласованных частей единой схемы или формализации представлений, отличающихся от общепринятых.

Единая схема является Схемой ≡

[COUNT(Единая схема) = 1  
 ] <В каждый момент времени существует лишь одна единая схема.>

Единая схема подразумевает текущее Состояние схемы ≡

[(Единая схема, Схема) EQUALS (Единая схема–Состояние схемы–Схема)  
 ] <Единая схема является той схемой, состояние которой она подразумевает.>

Пояснение. Именно единая схема используется по умолчанию для исполнения SCQL запросов, если обратное не указано в явном виде.

### 3.1.7 Описанные и порожденные элементы схем

Только часть элементов схемы описываются в явном виде. Остальные элементы могут быть выведены (порождены), как правило, из описанных элементов. Такие элементы считаются существующими в схеме с момента появления тех элементов схемы, из которых они порождены.

Описанный элемент схемы является Элементом схемы ≡

Порожденный элемент схемы является Элементом схемы ≡

Пояснение. Основное отличие неописанных порожденных элементов схем от описанных состоит в том, что для них не имеет смысла операция перебора, так как количество неописанных порожденных элементов может быть очень велико. Для неописанных порожденных элементов можно лишь проверять факт существования конкретного элемента, в то время описанные элементы схемы можно перебирать по порядку.

Пояснение. Часть из порожденных элементов могут быть также описанными. Это бывает полезным в тех случаях, когда для них необходимо указать некоторые дополнительные параметры, не определяемые в ходе индукции.

Некоторые из элементов схем, представляющих собой базовый набор элементов, существуют по определению (например, понятие "Число"). Такие элементы являются вырожденным случаем порожденных элементов, так как они постулированы, а не выведены из других элементов.

**Базовый элемент схемы** является **Порожденным элементом** схемы  $\equiv$

Пояснение. Базовые элементы также могут отсутствовать среди описанных элементов схем, так как их состав известен априори. Базовыми элементами считаются все понятия, ассоциации и ограничения, введенные в данной спецификации.

Пояснение. Конкретные правила вывода элементов, не являющихся базовыми, приводятся в тех спецификациях, где соответствующие порожденные элементы играют важную роль. Например, в спецификации SCM-SCQL-SPEC вводятся ролевые понятия, порождаемые из понятий.

### 3.1.8 Типизация понятий

Экземпляры понятий представляются в рамках вычислительных систем как значения переменных.

*Пример. Экземпляр понятия "Автомобиль: 'A002CP48'" может представляться строкой 'A002CP48', числом 385 или неким другим значением.*

**Экземпляр понятия** представляется в виде **Значения**  $\rightarrow$

[(Понятие–Экземпляр понятия–Значение):

Значение  $\equiv$  (Понятие)  $\equiv$  Экземпляр понятия

] <Каждое значение представляет только один экземпляр в рамках одного понятия.>

Значения могут быть как простых типов (число, строка и др.), так и составных типов (массив, структура и др.).

*Пример. Понятие "Строка документа" может быть представлено в виде структуры из двух числовых переменных: номер документа, номер строки внутри документа.*

**Значение принадлежит к Типу**  $\rightarrow$

Каждое понятие представляется в вычислительной системе в виде типа, который определяет набор значений, допустимых для понятия. Каждый экземпляр понятия кодируется в виде некоторого значения, принадлежащего тому типу, в виде которого представляется данное понятие.

**Понятие** представляется в виде **Типа**  $\rightarrow$

[(Понятие–Тип–Значение) EQUALS (Понятие–Экземпляр понятия–Значение)

] <Все экземпляры понятия представляются значениями только соответствующего типа и наоборот.>

Вопрос. *Почему нет возможности представить одно понятие в виде нескольких типов?* В этом случае затрудняется представление понятий в рамках вычислительной системы. Дело в том, что переменные в рамках вычислительной системы всегда принадлежат только одному типу, и существует единственный способ хранения значений

разных типов в одной переменной, состоящий в приведении всех переменных к одному типу, а именно к строке. Хранение всех переменных в виде строки неэффективно. Принятое решение позволяет достичь прозрачности, ясности представления понятий в рамках вычислительной системы без снижения эффективности.

Иллюстрация. На рис. 6 представлен пример типизации понятия "Возраст". Здесь приняты следующие обозначения: квадраты – значения типа "Число", эллипсы – экземпляры понятия "Возраст", пунктирные линии показывают соответствие значений типа и экземпляров понятия, а стрелка говорит о том, что понятие "Возраст" построено именно на типе "Число". Как можно видеть из иллюстрации, экземпляр понятия фактически представляет собой сочетание двух элементов: понятие, к которому относится экземпляр, и значение, в виде которого оно кодируется.

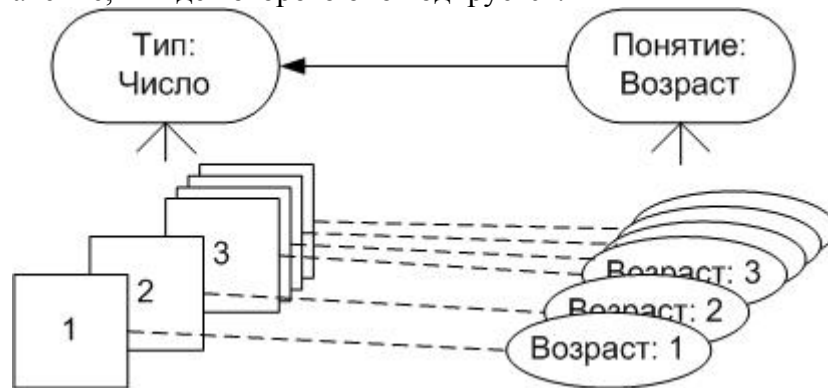


Рисунок 6 Пример понятия "Возраст", построенного на типе "Число"

С концептуальной точки зрения тип также представляет собой некоторое понятие, – то понятие, семантика которого эквивалентна самому типу. Такие понятия назовем собственными понятиями типа. Среди всех понятий, представляемых в виде некоторого типа, может существовать только одно собственное понятие этого типа.

*Пример. Собственным понятием типа "Число" является понятие "Число", в то время как числом могут представляться и другие понятия: "Возраст", "Цвет" и т.д.*

Собственное понятие является Понятием  $\equiv$   
 $[ \text{FOR ALL (Тип)} \{ \text{COUNT (Тип-Понятие-Собственное понятие)} = 1 \}$   
 $] \langle \text{Может существовать только одно собственное понятие среди всех понятий, представленных одним типом.} \rangle$

Вопрос. Каково назначение собственных понятий типов, как их можно использовать? Наличие собственных понятий позволяет переходить от типов к эквивалентным понятиям там, где это необходимо. Например, может потребоваться раскрыть структуру составного типа через понятия, а не через вложенные в него типы. В этом случае структура становится понятнее пользователям – специалистам в предметной области, – в то время как декомпозиция составного типа на вложенные типы понятна только специалистам в информационных технологиях.

Типы могут создаваться разными людьми для различных целей. Для повышения управляемости большого массива типов вводится возможность группировки их по предметным областям. При этом стандартные типы рассматриваются вне каких-либо предметных областей.

Тип может находиться в Предметной области →

Для ссылки на типы используются их обозначения. Каждый тип имеет строковое обозначение, уникальное в рамках его предметной области для каждого из языков, на которых это обозначение сформулировано. Если тип является стандартным (не входит ни в одну предметную область), то его строковое обозначение уникально среди всех таких типов для каждого из языков.

Тип имеет Обозначение →

[(Предметная область–Тип–(Обозначение, Строка, Язык)):  
 Тип ≡(Предметная область, Язык)≡ Строка  
 ] <Строки обозначений типов уникальны в рамках каждой предметной области для каждого из языков.>  
 [((Тип–(Обозначение, Строка, Язык)) MINUS (Тип–Предметная область)):  
 Тип ≡(Язык)≡ Строка  
 ] <Строки обозначений типов, не входящих в предметные области, уникальны среди всех таких типов для каждого из языков.>

Если тип входит в состав предметной области, то его строковое обозначение может быть не уникальным, так как может существовать другой тип в другой предметной области с тем же строковым обозначением на том же языке. Поэтому каждому типу для каждого из языков ставится в соответствие строка полного обозначения, уникально идентифицирующая данный тип среди всех возможных типов (и стандартных, и не стандартных) в рамках данного языка. Строка полного обозначения типа для некоторого языка представляет собой, – для тех типов, которые включены в предметные области, – конкатенацию через точку строки полного обозначения предметной области и строки обозначения типа на соответствующем языке. Если же тип является стандартным, то строкой его полного обозначения является строка собственного обозначения типа на соответствующем языке.

Тип имеет Полное обозначение ≡  
 [(Тип–Полное обозначение–(Обозначение, Строка, Язык)) EQUALS  
 (SELECT Тип, Язык, Строка(ПО) + '.' + Строка(типа) => Строка FROM  
 ((Язык, Строка(типа), Обозначение(типа))–Тип–Предметная область–  
 Полное обозначение(ПО)–(Обозначение(ПО), Строка(ПО), Язык))  
 UNION  
 SELECT Тип, Язык, Строка FROM  
 ((Тип–(Обозначение, Строка, Язык)) MINUS (Тип–Предметная область)))  
 ] <Полное обозначение типа представляет собой совокупность строк для каждого из языков. Каждая из этих строк образована конкатенацией строки полного обозначения соответствующей предметной области и строки обозначения типа через точку на соответствующем языке, или, если тип не находится в какой-либо предметной области, то эта строка – строка собственного обозначения типа на соответствующем языке.>

### 3.1.8.1 Значение

#### 3.1.8.1.1 Простые и составные значения

Значения делятся на две категории – простые и составные. Составные значения состоят из ряда вложенных значений, каждое из которых располагается в своей позиции в рамках составного значения. Вложенные значения, в свою очередь, могут быть как простыми, так и составными. Вложенное значение может входить в одно составное значение несколько раз, если оно находится в различных позициях.

*Пример. Если мы рассмотрим вектор чисел, то число 3 может находиться в нем многократно, например, <2, 3, 1, 5, 3>. Здесь число 3 занимает две позиции: позицию №2 и №5. Другой пример, если мы рассмотрим структуру из двух чисел, то число 3 может быть в обеих позициях этой структуры: <3, 3>.*

Составное значение является Значением ≡  
Вложенное значение является Значением ≡  
Вложенное значение входит в Составное значение в Позиции  
Простое значение является Значением ≡  
 [FOR ALL (Простое значение)

{COUNT(Простое значение–Значение–Составное значение) = 0}  
] <Простое значение не является составным.>

Не допускаются циклические вложения значений друг в друга (непосредственно или опосредовано).

[NO\_CYCLE SEQ(Значение(вложенное), Значение(составное)) OVER  
(Значение(составное)–(Составное значение, Позиция, Вложенное значение)–  
Значение(вложенное))  
] <В направленном графе вложения значений не должно быть циклов  
(направленность графа видна из факта использования SEQ вместо SET,  
направление – от вложенных значений к составным).>

Пояснение. Значения хранятся в рамках вычислительных систем, причем для хранения вложенных значений выделяются дополнительные ячейки памяти. Наличие циклического вложения значений приведет к потребности в бесконечном объеме памяти, что недопустимо.

Различают составные значения двух принципиально отличающихся категорий – векторные и множественные значения. В рамках векторных значений каждое вложенное значение занимает уникальную позицию, а в рамках множественных значений все вложенные значения занимают позицию 1. В результате множественные значения не могут содержать одно и то же вложенное значение несколько раз, в то время как векторные – могут.

Векторное значение является Составным значением  $\equiv$   
[SEQUENCE Позиция, 1, 1 WITHIN Векторное значение  
OVER (Векторное значение–(Составное значение, Вложенное значение, Позиция))  
] <Позиции вложенных значений представляют собой последовательность целых чисел от 1 через 1 (уникальны для вложенных значений) в рамках векторного значения.>

Множественное значение является Составным значением  $\equiv$   
[(Множественное значение–(Составное значение, Вложенное значение, Позиция))  
EQUALS (Множественное значение–(Составное значение, Вложенное значение,  
Позиция = 1))  
] <Позиции значений, вложенных во множественные значения, равны 1.>  
[COUNT (Векторное значение–Составное значение–Множественное значение) = 0  
AND FOR ALL (Составное значение) {  
COUNT (Векторное значение–Составное значение) = 1 OR  
COUNT (Составное значение–Множественное значение) = 1 }  
] <Составное значение может быть либо векторным, либо множественным.>

Вопрос. *Структурное значение тоже является составным значением, почему оно не определено здесь?* В рамках структурного значения позиции вложенных значений играют столь же принципиальную роль, что и в рамках векторных значений. Более того, и векторное и структурное значения равноправно можно представить как последовательность значений. Концептуально эти виды значений не отличаются, поэтому принято решение для избежания избыточности ограничиться векторным значением.

### 3.1.8.1.2 Кодирование значений

Любое значение можно представить в виде числа, однозначно соответствующего данному значению в рамках его типа. Кодирование значений служит для их упорядочения в рамках типа.

Значение кодируется Числом  $\rightarrow$   
[(Тип–Значение–Число): Значение  $\equiv$ (Тип) $\equiv$  Число  
] <В рамках типа каждое значение кодируется уникальным числом.>

Пояснение. Значения разных типов могут быть различны, но кодироваться одним числом. Например, число 23 и целое число 23 – это разные значения, закодированные одним числом.

Вопрос. *Кодирование значений имеет какое-либо отношение к представлению значений в рамках вычислительных систем?* Прежде всего, кодирование значений соответствует базовому способу сортировки. При этом кодирование может как соответствовать, так и не соответствовать представлению значений в вычислительной системе. Базовый способ сортировки – это способ, который используется по умолчанию, но в явном виде может быть задан другой, отличный от базового способ сортировки.

### 3.1.8.1.3 Приведение значений

Значения различных типов могут приводиться друг к другу, как в явном виде, так и неявно.

*Пример. Строка "1234" может быть приведена как в явном виде к числу 1234, так и к целому числу 1234. Примером неявного преобразования является приведение числа 1234 к целому числу 1234.*

Приводимое значение является Значением  $\equiv$   
Приведенное значение является Значением  $\equiv$   
Приводимое значение приводится к Приведенному значению  $\rightarrow$

Пояснение. Конкретные правила приведения значений описаны ниже для соответствующих типов. В данной спецификации описан базовый набор приведений типов, так как в языках программирования есть свои механизмы приведения типов, которые могут использоваться наряду с механизмами SCM. Например, в спецификации отсутствуют следующие приведения типов: строки к числу, производной структуры к базовой структуре, вектора производных структур к вектору базовых структур, множества производных структур к множеству базовых структур и др. Данная спецификация может быть расширена дополнительными способами приведения типов, если будет выявлена соответствующая необходимость.

Вопрос. *Должны ли значения, которые одинаково кодируются, приводиться одно к другому?* Не обязательно. Например, строка "abc" может кодироваться числом  $0x616263 * 16^{(1000000-3)}$ , но это не значит, что данная строка приводится к данному числу. Здесь мы умножили шестнадцатеричную запись строки на  $16^{(1000000-3)}$  для того, чтобы сохранить порядок строк в алфавитном порядке, без учета длины строк, например, в случае "abc", "abcd", "abd". Число 1000000 взято из предположения, что максимальная длина строки 1000000, а число 16 – из предположения, что каждая буква кодируется 16 битами; если это не так, то данные числа должны быть увеличены соответствующим образом. При этом совершенно не обязательно при сравнении строк вычислять соответствующие им кодирующие числа в явном виде. Достаточно применить процедуру, которая будет эквивалентна сравнению данных кодирующих чисел. Для строк эта процедура очевидна – посимвольное сравнение строк от начала.

Вопрос. *Могут ли значения, кодируемые по-разному, приводиться одно к другому?* Могут. Например, строка "1234" может кодироваться числом  $0x31323334 * 16^{(1000000-4)}$ , но при этом она приводится к числу 1234 (которое кодируется тем же числом 1234).

### 3.1.8.2 Метатип

Тип определяет набор значений, которые может принимать понятие. Этот набор значений может быть подчинен неким правилам.

*Пример. Может существовать тип "целые числа от 0 до 9", который допускает только значения 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Этот тип приводится к целому числу, но он не является типом "целое число".*

Типы, которые имеют общие правила определения допустимых значений, относятся к одному метатипу. Простые типы без каких-либо ограничений значений относятся к метатипу "Базовый". Существуют метатипы, представляющие собой диапазоны или наборы простых типов, а также наборы диапазонов простых типов. Для составных типов отдельно существуют метатипы "Структура", "Вектор" и "Множество", а также метатипы, ограничивающие составные типы по размерности, и наборы составных типов. Все перечисленные метатипы и соответствующие типы описаны в последующих пунктах.

Тип относится к Метатипу →  
 [Метатип = {"Базовый",  
 "Справочник", "Диапазон чисел", "Диапазон целых чисел", "Диапазон дат",  
 "Диапазон кодов", "Строка ограниченной длины", "Строка по шаблону", "Набор чисел", "Набор целых чисел", "Набор дат", "Набор строк", "Набор кодов", "Набор диапазонов чисел", "Набор диапазонов целых чисел", "Набор диапазонов дат",  
 "Набор диапазонов кодов",  
 "Структура", "Вектор", "Множество", "Ограниченный вектор", "Ограниченное множество", "Набор структурных значений", "Набор векторных значений", "Набор множественных значений"}]

Пояснение. Состав метатипов может расширяться в будущем при необходимости. Конкретные типы, относящиеся к тому или иному метатипу (кроме базового), могут объявляться динамически в процессе работы с SCM. Базовый же набор типов фиксирован и перечислен в рамках соответствующего метатипа "Базовый".

Вопрос. *Зачем введено понятие "Метатип", разве недостаточно уже существующего понятия "Тип"?* В рамках существующих метатипов пользователями могут объявляться производные типы. В этом процессе метатип определяет общие требования к структуре каждого типа. Например, метатип "Диапазон чисел" говорит о том, что каждый тип данного метатипа должен описываться двумя числами, – левой и правой границей, – и типом границ. Каждый из типов некоторого метатипа подразумевает свое множество значений, но все его типы построены по общему правилу, характерному данному метатипу. Именно наличие различных правил формирования типов и стало причиной введения понятия "Метатип". Коротко метатип можно определить как правило формирования типов.

### 3.1.8.3 Простой тип

Простые типы – это типы, все значения которых являются также простыми (не имеют вложенных значений). Простые типы включают в свой состав фиксированный набор базовых типов, а также могут создаваться различные производные простые типы (например, диапазон чисел, справочник и др.) в процессе работы с SCM.

Простой тип является Типом ≡  
 [Простой тип = {..., "Число", "Целое число", "Булево число", "Дата", "Строка", ...}]  
 [(Простое значение–Значение) EQUALS (Значение–Тип–Простой тип)  
 ] <Все простые значения относятся к простому типу.>

#### 3.1.8.3.1 Базовый тип

Базовые типы – фиксированный набор фундаментальных типов, представимых эффективным образом в вычислительных системах. Базовыми типами являются число, целое число, булево число, дата и строка. Все базовые типы относятся к метатипу "Базовый". Те типы, которые не являются базовыми, считаются производными.

Базовый тип является Типом ≡  
 [(Тип–Базовый тип) EQUALS  
 ((Тип–Простой тип), (Простой тип = "Число" OR Простой тип = "Целое число" OR

Простой тип = "Булево число" OR Простой тип = "Дата" OR Простой тип = "Строка"))

] <Базовый тип является одним из простых типов: "Число", "Целое число", "Булево число", "Дата", "Строка".>

[(Тип–Базовый тип) EQUALS

(Тип–Метатип = "Базовый")

] <Базовые типы относятся к метатипу "Базовый".>

Производный тип является Типом ≡

[FOR ALL ((Тип) MINUS (Тип–Базовый тип)) EXISTS (Тип–Производный тип)

] <Тип считается производным, если он не является базовым.>

Вопрос. *Почему в набор базовых типов не включены такие типы, как короткое число, короткое целое число, строка фиксированной длины и т.п.?* С концептуальной точки зрения короткое число представляет собой число, имеющее ограниченный набор допустимых значений. Принято решение не вводить базовые типы в тех случаях, когда аналогичного эффекта можно добиться объявлением соответствующего производного типа. Например, вместо введения базового типа "Короткое целое число" можно объявить производный тип "Целое число от -32767 до +32768", относящийся к метатипу "Диапазон целых чисел". Вместо введения базового типа "Строка фиксированной длины" можно объявить производный тип "Строка длиной от 0 до 5" (ограничив строку длиной до пяти символов включительно) или производный тип "Строка длиной от 5 до 5" (ограничив длину строки строго пятью символами), относящиеся к метатипу "Строка ограниченной длины", и т.д.

Принцип представления базовых типов в вычислительных системах соответствует принятому в рамках конкретной архитектуры вычислительной системы. Точность (размерность) для базовых типов выбирается по максимуму: число – long double, целое число – long, дата – long, строка – не ограничена. Производные типы могут иметь меньшую точность, если они ограничены соответствующим образом (например, "целое число от -32767 до 32768").

Число – простое значение, относящееся к типу "Число".

Число является Простым значением ≡

[(Число, Простое значение) EQUALS

(Простое значение–Тип–Простой тип = "Число")

] <Число относится к типу "Число".>

Целое число – простое значение, относящееся к типу "Целое число".

Целое число является Простым значением ≡

[(Целое число, Простое значение) EQUALS

(Простое значение–Тип–Простой тип = "Целое число")

] <Целое число относится к типу "Целое число".>

Целое число и число – различные значения. Но при этом целое число может быть приведено к числу, причем к тому, которое закодировано тем же способом (и наоборот).

*Пример. Целое число 3 и число 3 – это не одно и то же, так как первое относится к типу "Целое число", а второе – "Число". Тем не менее, эти значения приводятся друг к другу.*

Целое число приводится к Числу ≡

[(Целое число, Число) EQUALS

(Целое число–Простое значение(целое число)–Значение(целое число)–Приводимое значение–Приведенное значение–Значение(число)–Простое значение(число)–Число)]

[SELECT Целое число, Число

FROM (Целое число–Простое значение–Значение–Число)

EQUALS SELECT Целое число, Число

FROM (Целое число–Число(целое)–Простое значение–Значение–Число)

] <Целое число и число закодированы одинаково.>

Данное приведение значений и все приведения, описанные ниже, являются частными случаями приведения значений, описанного в пункте 3.1.8.1.3.

Булево число – простое значение, относящееся к типу "Булево число". Булево число принимает только два значения – 0 (ложь) и 1 (истина).

Булево число является Простым значением ≡  
[(Булево число, Простое значение) EQUALS  
(Простое значение–Тип–Простой тип = "Булево число")  
] <Булево число относится к типу "Булево число".>  
[(Булево число–Простое значение–Значение–Число)  
EQUALS (Число = 0 OR Число = 1)  
] <Существует только два булевых числа – одно 0, другое – 1.>

Булево число приводится к целому числу, а через целое число и просто к числу.

Булево и приведенное целое число закодированы одинаково.

Булево число приводится к Целому числу ≡  
[(Булево число, Целое число) EQUALS  
(Булево число–Простое значение(булево число)–Значение(булево число)–  
Приводимое значение–Приведенное значение–Значение(целое число)–Простое  
значение(целое число)–Целое число)]  
[SELECT Булево число, Число  
FROM (Булево число–Простое значение–Значение–Число)  
EQUALS SELECT Булево число, Число  
FROM (Булево число–Целое число–Простое значение–Значение–Число)  
] <Булево число и целое число закодированы одинаково.>

Дата – простое значение, относящееся к типу "Дата".

Дата является Простым значением ≡  
[(Дата, Простое значение) EQUALS  
(Простое значение–Тип–Простой тип = "Дата")  
] <Дата относится к типу "Дата".>

Строка – простое значение, относящееся к типу "Строка".

Строка является Простым значением ≡  
[(Строка, Простое значение) EQUALS  
(Простое значение–Тип–Простой тип = "Строка")  
] <Строка относится к типу "Строка".>

### 3.1.8.3.2 Справочник

Справочник является типом, относящимся к метатипу "Справочник". Справочники позволяют поставить в соответствие коду некое естественное обозначение/наименование на различных естественных языках. Данное обозначение используется для человеко-машинного взаимодействия, а все ссылки на значения справочников хранятся в виде кодов.

Обоснование. Использование кодов для ссылки на значения справочника имеет несколько преимуществ: а) более компактно, по сравнению с использованием самих обозначений; б) позволяет многоязыковое представление обозначений кода; в) позволяет централизованную корректировку обозначений кодов в одном месте.

Справочник является Типом ≡  
[(Справочник, Тип) EQUALS (Тип–Метатип = "Справочник")  
] <Справочник относится к метатипу "Справочник".>

Код представляет собой простое значение, относящееся к одному из справочников.

Код является Простым значением ≡  
[(Код, Простое значение) EQUALS

(Простое значение–Значение–Тип–Справочник)

] <Код относится к одному из справочников.>

Код приводится к целому числу, причем в рамках каждого справочника существует свое взаимнооднозначное отображение кодов в целые числа. Данное приведение значений является частным случаем приведения значений, описанного в пункте 3.1.8.1.3.

Код представляется в виде Целого числа →

[(Код, Целое число) EQUALS

(Код–Простое значение(код)–Значение(код)–Приводимое значение–Приведенное значение–Значение(целое число)–Простое значение(целое число)–Целое число)]

[(Справочник–Тип–Значение–Простое значение–Код–Целое число):

Код ≡(Справочник)≡ Целое число

] <В рамках одного справочника существует единственное целое число для каждого кода.>

Коды справочников являются многоязыковыми, то есть каждому коду может соответствовать свое строковое обозначение в рамках каждого естественного языка (см. перечисление естественных языков выше).

Код определяет Строку для Языка

[(Код, Строка, Язык): (Код, Язык) → Строка

] <Для каждого языка может быть задана только одна строка в рамках кода.>

Некоторые из кодов могут иметь историю изменения их обозначений. Это необходимо в тех случаях, когда требуется буквальное восстановление внешнего вида документов на некоторый момент времени в прошлом, что, тем не менее, не запрещает представить эти документы по обозначениям, актуальным в настоящий момент времени. История кода представляет собой множество сочетаний строк, языков и тех моментов времени, до которых действовало каждое сочетание (не включая границу). Причем для каждого сочетания языка и момента времени в рамках истории кода может быть задана только одна строка.

*Пример. Запись истории (История кода: 28, Строка: "Машина", Язык: "Русский", Момент времени: "19.10.2005 11:17:00") говорит о том, что в рамках истории кода 28 обозначение "Машина" на русском языке действовало до 19.10.2005 11:17:00 не включая данный момент времени.*

История кода является Кодом ≡

История кода определяет Строку для Языка вплоть до Момент времени

[(История кода, Строка, Язык, Момент времени):

(История кода, Язык, Момент времени) → Строка

] <Для каждого языка может быть задана только одна строка в рамках истории кода на некоторый момент времени.>

Пояснение. Код определяет актуальные обозначения на текущий момент времени. Полное отсутствие каких-либо строк для кода должно расцениваться как отмена действия данного кода. Отмена означает, что не рекомендуется занесение новой информации в состояния схем, ссылающейся на данный код. Тем не менее, ссылка на такой код может заноситься в состояния схем, то есть отмена кода не эквивалентна его удалению. Удаление кода может быть осуществлено только в том случае, если не существует ни одной ссылки на него в рамках существующих состояний схем.

Пояснение. Обозначение кода на некоторый момент времени в прошлом может быть получено по его истории. Для этого берется та запись истории, момент завершения действия которой строго больше искомого момента и при этом ближе к искомому моменту, чем все остальные записи истории. Код может быть отменен с некоторого момента времени в прошлом. В этом случае все его строки переносятся в исторические записи, действующие вплоть до момента отмены, а сам код отменяется (все его строки удаляются, остается только код без строк).

### 3.1.8.3.3 Ограниченный тип

Каждый тип подразумевает некое множество значений. Если тип объявляется на основе другого, исходного типа как подмножество его значений, то такой тип называется ограниченным. Ограниченный тип объявляется на основе лишь одного исходного типа. Допускается объявление нескольких ограниченных типов на основе одного исходного.

*Пример. Тип "Целое число от -32767 до 32768" является ограниченным типом, построенным на основе исходного типа "Целое число".*

Исходный тип является Типом  $\equiv$   
Ограниченный тип является Типом  $\equiv$   
Ограниченный тип построен на Исходном типе  $\rightarrow$

Значения, принадлежащие исходным типам, назовем исходными, а значения, принадлежащие ограниченным типам – допустимыми.

Исходное значение является Значением  $\equiv$   
[(Исходное значение–Значение) EQUALS (Значение–Тип–Исходный тип)  
] <Исходное значение принадлежит одному из исходных типов.>  
Допустимое значение является Значением  $\equiv$   
[(Допустимое значение–Значение) EQUALS (Значение–Тип–Ограниченный тип)  
] <Допустимое значение должно принадлежать одному из ограниченных типов.>

Каждое допустимое значение приводится к одному исходному значению соответствующего исходного типа. Данное приведение значений является частным случаем общего приведения значений, описанного в пункте 3.1.8.1.3.

Допустимое значение приводится к Исходному значению  $\rightarrow$   
[(Допустимое значение, Исходное значение) EQUALS  
(Допустимое значение–Значение(допустимое)–Приводимое значение–Приведенное значение–Значение(исходное)–Исходное значение)]  
[FOR ALL (Допустимое значение, Исходное значение)  
EXISTS (Исходное значение–Значение(исходное)–Тип(исходный)–Исходный тип–  
Ограниченный тип–Тип(ограниченный)–Значение(допустимое)–Допустимое значение)  
] <Допустимое значение может приводиться только к тем исходным значениям, которые принадлежат тому исходному типу, на котором построен данный ограниченный тип.>

Из определения ограниченного типа можно заключить, что соответствие исходных и допустимых значений в рамках каждого ограниченного типа должно быть взаимнооднозначным.

*Пример. Значение исходного типа "Целое число" может соответствовать только одному допустимому значению типа "Целое число от -32767 до 32768", и наоборот.*

[(Ограниченный тип–Тип–Значение–Допустимое значение–Исходное значение):  
Исходное значение  $\equiv$ (Ограниченный тип) $\equiv$  Допустимое значение  
] <В рамках ограниченного типа каждому исходному значению может соответствовать только одно допустимое значение.>

Если считать, что ограниченный тип просто сужает исходный тип без каких-либо преобразований значений, то верно положить, что допустимое значение и соответствующее ему исходное значение должны быть закодированы одинаковым образом.

[SELECT Допустимое значение, Число FROM (Допустимое–Значение–Число)  
EQUALS SELECT Допустимое значение, Число  
FROM (Допустимое значение–Исходное значение–Значение–Число)  
] <Допустимое значение и исходное значение закодированы одинаково.>

Пояснение. При необходимости в будущем специфицировать механизмы определения новых типов не просто путем ограничения существующих, а с некими преобразованиями значений, потребуется объявить другую категорию типов, отличную от категории ограниченных типов.

### 3.1.8.3.4 Диапазон значений

Диапазон значений представляет собой ограниченный тип, допустимые значения которого представляют собой непрерывный диапазон значений исходного типа. Допустимые значения, принадлежащие диапазону значений, назовем значениями диапазона.

<p><u>Диапазон значений</u> является Типом <math>\equiv</math>          [FOR ALL (Диапазон значений–Тип) EXISTS (Тип–Ограниченный тип)          ] &lt;Диапазон значений является ограниченным типом.&gt;</p> <p><u>Значение диапазона</u> является Значением <math>\equiv</math>          [(Значение диапазона, Значение) EQUALS (Значение–Тип–Диапазон значений)          ] &lt;Значение диапазона относится к одному из диапазонов значений.&gt;</p>
--

Диапазон значений определяется левой и правой границей, представляющими собой значения исходного типа.

<p><u>Левая граница</u> является Значением <math>\equiv</math>  <u>Правая граница</u> является Значением <math>\equiv</math>  <u>Диапазон значений</u> определяетсялевой границей <math>\rightarrow</math>  <u>Диапазон значений</u> определяется Правой границей <math>\rightarrow</math>          [FOR ALL (              SELECT Диапазон значений, Значение              FROM (Диапазон значений–Левая граница–Значение)              UNION SELECT Диапазон значений, Значение              FROM (Диапазон значений–Правая граница–Значение))          EXISTS (Диапазон значений–Тип(ограниченный)–Ограниченный тип–              Исходный тип–Тип(исходный)–Значение)          ] &lt;И левая, и правая граница являются значениями исходного типа для диапазона значений.&gt;</p>
---

Сочетание левой и правой границ назовем диапазоном. Тогда диапазон значений определяется тем диапазоном, который имеет те же левую и правую границы, что и сам диапазон значений. Диапазон может дополнительно определяться типом границ: включая обе границы, не включая обе границы, включая только правую или включая только левую.

<p><u>Диапазон</u> определяетсялевой границей <math>\rightarrow</math>  <u>Диапазон</u> определяется Правой границей <math>\rightarrow</math>  <u>Диапазон значений</u> определяется Диапазоном <math>\rightarrow</math>              [(Диапазон значений, Левая граница) EQUALS              (Диапазон значений–Диапазон–Левая граница)]              [(Диапазон значений, Правая граница) EQUALS              (Диапазон значений–Диапазон–Правая граница)]          Диапазон может определяться Типом границ <math>\rightarrow</math>          [Тип границ = {"[", "]", "(", "}" ]</p>
--

Вопрос. Зачем вводить понятие "Диапазон" в явном виде, разве недостаточно задать левую и правую границу для диапазона значений? Такое решение было принято по двум причинам: а) диапазон может иметь дополнительные характеристики, например, тип границ (с включением, без включения и т.п.); б) тип может быть ограничен не только при помощи одного диапазона, но и при помощи более сложных конструкций на основе диапазона, например, при помощи набора диапазонов (см. ниже набор диапазонов).

### 3.1.8.3.5 Непрерывный диапазон

Непрерывный диапазон представляет собой диапазон значений, обязательно определяемый типом границ.

*Пример. Тип "Число в диапазоне (-3, 15]" является непрерывным диапазоном, построенным на типе "Число", с левой границей -3, правой границей 15 и типом границ "]"*.

Непрерывный диапазон является Типом  $\equiv$   
[FOR ALL (Непрерывный диапазон–Тип) EXISTS (Тип–Диапазон значений)  
] <Непрерывный диапазон является диапазоном значений.>  
Непрерывный диапазон определяется Типом границ  $\rightarrow$   
[(Непрерывный диапазон, Тип границ) EQUALS  
(Непрерывный диапазон–Тип–Диапазон значений–Диапазон–Тип границ)]

Пояснение. Такой диапазон назван непрерывным по той причине, что только для непрерывных диапазонов имеет смысл определять типы границ. В остальных случаях для диапазонов достаточно задать крайние границы и постулировать, что эти границы всегда включаются в диапазон. Непрерывные диапазоны имеет смысл определять на действительно непрерывных исходных типах (например, "число") или рассматриваемых как непрерывные (например, "дата").

Непрерывный диапазон допускает те исходные значения, которые находятся между левой и правой границами, причем включение/исключение граничных значений осуществляется в соответствии с заданным типом границ.

[SELECT Непрерывный диапазон, Число  
FROM (Непрерывный диапазон–Тип–Значение–Число)  
EQUALS  
SELECT Непрерывный диапазон, Число FROM  
((Непрерывный диапазон–Тип(ограниченный)–Ограниченный тип–Исходный тип–  
Тип(исходный)–Значение–Число),  
(Тип границ–Непрерывный диапазон–Тип(ограниченный)–Диапазон значений–  
[Левая граница, Правая граница]),  
Значения между (Левая граница, Правая граница, Тип границ, Значение))  
] <В непрерывный диапазон входят только те значения исходного типа, которые  
находятся между левой и правой границей (в соответствии с порядком  
кодирующих чисел), с включением/исключением границ в соответствии с типом  
границ.>

Исходные значения, находящиеся между левой и правой границами, определяются в соответствии со способом их кодирования и типом границ. Если тип границы предусматривает включение левой границы, то левая граница входит в результирующий набор значений, то же самое и с правой границей.

*Пример. Дата '01.01.2005' будет находиться между датами '31.12.2004' и '01.01.2006' по той причине, что их кодирующие числа находятся в том же соотношении.*

Значения между (Левая граница, Правая граница, Тип границ, Значение) := [  
((Левая граница×Правая граница×Тип границ×Значение),  
(Левая граница–Значение(левая)–Число(левое)),  
(Правая граница–Значение(правая)–Число(правое)), (Значение–Число))  
WHEN {  
IF Тип границ = "[" THEN  
    {Число(левое) <= Число AND Число <= Число(правое)}  
ELSE IF Тип границ = "]" THEN  
    {Число(левое) < Число AND Число < Число(правое)}  
ELSE IF Тип границ = "]" THEN

```

        {Число(левое) <= Число AND Число < Число(правое)}
ELSE {Число(левое) < Число AND Число <= Число(правое)} }
] <В диапазон значений между двумя границами входят только те значения,
которые закодированы числами находящимися между левой и правой границей, со
включением/исключением границ в соответствии с типом границ.>

```

Пояснение. Данный предикат вводится и отдельно, в явном виде, потому, что в дальнейшем он будет повторно использован для специфицирования набора непрерывных диапазонов.

### 3.1.8.3.6 Дискретный диапазон

Дискретный диапазон представляет собой такой диапазон значений, для которого не определяется тип границ, так как он всегда одинаков – включение обеих границ.

*Пример. Тип "целое число в диапазоне от -3 до 15" представляет собой дискретный диапазон, построенный на типе "целое число", с левой границей -3, правой границей 15. В этот диапазон входят все целые числа от -3 до 15, включая -3 и 15.*

```

Дискретный диапазон является Типом ≡
[FOR ALL (Дискретный диапазон–Тип) EXISTS (Тип–Диапазон значений)
] <Дискретный диапазон является диапазоном значений.>
[FOR ALL (Дискретный диапазон) NOT EXISTS
(Дискретный диапазон–Тип–Диапазон значений–Диапазон–Тип границ)
] <Для дискретного диапазона тип границ не задается.>

```

Дискретный диапазон включает все исходные значения, которые находятся между левой и правой границами, включая обе границы. Исходные значения, находящиеся между левой и правой границами, определяются в соответствии с их кодами.

*Пример. Строка "abc" будет находиться между строками "abb" и "abd" по той причине, что ее кодирующее число находится между кодирующими числами данных строк: "abc" – 0x616263\*16^(1000000-3), "abb" – 0x616262\*16^(1000000-3), и "abd" – 0x616264\*16^(1000000-3). В случае строк существует эквивалентная процедура сравнения без вычисления кодирующих чисел – посимвольное сравнение от начала строк (см. 3.1.8.1.3).*

```

[SELECT Дискретный диапазон, Число
FROM (Дискретный диапазон–Тип–Значение–Число)
EQUALS
SELECT Дискретный диапазон, Число FROM
((Дискретный диапазон–Тип(ограниченный)–Ограниченный тип–Исходный тип–
Тип(исходный)–Значение–Число),
(Тип(ограниченный)–Диапазон значений–[Левая граница, Правая граница]),
(Левая граница–Значение(левое)–Число(левое)),
(Правая граница–Значение(правое)–Число(правое)),
(Число(левое)<= Число AND Число <= Число(правое)))
] <В дискретный диапазон входят только те значения исходного типа, которые
находятся между левой и правой границей включительно (в соответствии с
порядком кодирующих чисел).>

```

Вопрос. Почему для дискретных диапазонов не задается тип границ в явном виде, почему всегда используется включение? Причина такого решения состоит в том, что в дискретном одномерном пространстве любой диапазон без включения границ тривиальным образом сводится к диапазону с включением границ. Например, непрерывный диапазон "целое число в диапазоне (-3, 15]" сводится к дискретному диапазону "целое число от -2 до 15" (то есть [-2, 15]).

### 3.1.8.3.7 Диапазон чисел

Диапазон чисел является непрерывным диапазоном, относящимся к метатипу "Диапазон чисел", исходным типом которого является число.

Диапазон чисел является Типом ≡  
[(Диапазон чисел, Тип) EQUALS (Тип–Метатип = "Диапазон чисел")  
] <Диапазон чисел относится к метатипу "Диапазон чисел".>  
[FOR ALL (Диапазон чисел, Тип) EXISTS (Тип–Непрерывный диапазон)  
] <Диапазон чисел является непрерывным диапазоном.>  
[FOR ALL (Диапазон чисел–Тип) EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Простой тип = "Число")  
] <Исходным типом диапазона чисел является число.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся диапазонами чисел, путем задания левой и правой границ, а также типа границ, например "Число в диапазоне (-3, 15]".

### 3.1.8.3.8 Диапазон целых чисел

Диапазон целых чисел является дискретным диапазоном, относящимся к метатипу "Диапазон целых чисел", исходным типом которого является целое число.

Диапазон целых чисел является Типом ≡  
[(Диапазон целых чисел, Тип) EQUALS (Тип–Метатип = "Диапазон целых чисел")  
] <Диапазон целых чисел относится к метатипу "Диапазон целых чисел".>  
[FOR ALL (Диапазон целых чисел, Тип) EXISTS (Тип–Дискретный диапазон)  
] <Диапазон целых чисел является дискретным диапазоном.>  
[FOR ALL (Диапазон целых чисел–Тип) EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Простой тип = "Целое число")  
] <Исходным типом диапазона целых чисел является целое число.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся диапазонами целых чисел, путем задания левой и правой границ, например "Целое число в диапазоне от -3 до 15".

### 3.1.8.3.9 Диапазон дат

Диапазон дат является непрерывным диапазоном, относящимся к метатипу "Диапазон дат", исходным типом которого является "Дата".

Диапазон дат является Типом ≡  
[(Диапазон дат, Тип) EQUALS (Тип–Метатип = "Диапазон дат")  
] <Диапазон дат относится к метатипу "Диапазон дат".>  
[FOR ALL (Диапазон дат, Тип) EXISTS (Тип–Непрерывный диапазон)  
] <Диапазон дат является непрерывным диапазоном.>  
[FOR ALL (Диапазон дат–Тип) EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Простой тип = "Дата")  
] <Исходным типом диапазона дат является дата.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся диапазонами дат, путем задания левой и правой границ, а также типа границ, например "Дата в диапазоне ['01.01.2005', '01.01.2006')".

### 3.1.8.3.10 Диапазон кодов

Диапазон кодов является дискретным диапазоном, относящимся к метатипу "Диапазон кодов", исходным типом которого является справочник.

Диапазон кодов является Типом ≡  
[(Диапазон кодов, Тип) EQUALS (Тип–Метатип = "Диапазон кодов")  
] <Диапазон кодов относится к метатипу "Диапазон кодов".>  
[FOR ALL (Диапазон кодов, Тип) EXISTS (Тип–Дискретный диапазон)

] <Диапазон кодов является дискретным диапазоном.>  
 [FOR ALL (Диапазон кодов–Тип)  
 EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Справочник)  
 ] <Исходным типом диапазона кодов является справочник.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся диапазонами кодов, путем задания левой и правой границ, например "Код в диапазоне от 5 до 15".

### 3.1.8.3.11 Строка ограниченной длины

Строка ограниченной длины является ограниченным типом, относящимся к метатипу "Строка ограниченной длины", исходным типом которого является строка.

Строка ограниченной длины является Типом ≡  
 [FOR ALL (Строка ограниченной длины–Тип) EXISTS (Тип–Ограниченный тип)  
 ] <Строка ограниченной длины является ограниченным типом.>  
 [(Строка ограниченной длины, Тип) EQUALS  
 (Тип–Метатип = "Строка ограниченной длины")  
 ] <Строка ограниченной длины относится к метатипу "Строка ограниченной длины".>  
 [FOR ALL (Строка ограниченной длины–Тип) EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Простой тип = "Строка")  
 ] <Исходным типом строки ограниченной длины является строка.>

Строка ограниченной длины параметризуется минимальной и максимальной длинами. Допустимые значения данного типа – все возможные строки, длина которых находится между этими двумя параметрами включительно.

*Пример. Тип "Строка длиной от 0 до 5" подразумевает значения, которые представляют собой любые строки длиной от 0 до 5".*

Строка ограниченной длины параметризована Минимальной длиной ≡  
 [Минимальная длина = Целое число]  
Строка ограниченной длины параметризована Максимальной длиной ≡  
 [Максимальная длина = Целое число]  
 [SELECT Строка ограниченной длины, Число  
 FROM (Строка ограниченной длины–Тип–Значение–Число)  
 EQUALS  
 SELECT Строка ограниченной длины, Число FROM  
 ((Строка ограниченной длины–[Минимальная длина, Максимальная длина]),  
 SELECT Строка, Число, LENGTH(Строка)  
 FROM (Строка–Значение–Число)),  
 (LENGTH(Строка)  
 BETWEEN Минимальная длина AND Максимальная длина))  
 ] <В строку ограниченной длины входят только те строки, длина которых больше минимальной длины включительно и меньше максимальной длины включительно.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся строками ограниченной длины, путем задания минимальной и максимальной длин, например, "Строка длиной от 0 до 5" или просто "Строка длиной до 5".

### 3.1.8.3.12 Строка по шаблону

Строка по шаблону является ограниченным типом, относящимся к метатипу "Строка по шаблону", исходным типом которого является строка.

Строка по шаблону является Типом ≡  
 [FOR ALL (Строка по шаблону–Тип) EXISTS (Тип–Ограниченный тип)

] <Строка по шаблону является ограниченным типом.>  
 [(Строка по шаблону, Тип) EQUALS (Тип–Метатип = "Строка по шаблону")  
 ] <Строка по шаблону относится к метатипу "Строка по шаблону".>  
 [FOR ALL (Строка по шаблону–Тип) EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Простой тип = "Строка")  
 ] <Исходным типом строки по шаблону является строка.>

Строка по шаблону параметризуется множеством шаблонов. Допустимые значения такого типа – все строки, которые удовлетворяют хотя бы одному из заданных шаблонов.  
*Пример. Тип "Строка по шаблону 'Задача%' или 'ЗАДАЧА%'" подразумевает в качестве значений все строки, начинающиеся с подстрок "Задача" или "ЗАДАЧА" и имеющие любое продолжение.*

Строка по шаблону параметризована Шаблонами –  
 [SELECT Строка по шаблону, Число  
     FROM (Строка по шаблону–Тип–Значение–Число)  
 EQUALS  
 SELECT Строка по шаблону, Число  
     FROM ((Строка по шаблону–Шаблон), (Строка–Значение–Число),  
     (Строка LIKE Шаблон))  
 ] <В строку по шаблону входят только те строки, которые удовлетворяют хотя бы одному из указанных шаблонов.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся строками по шаблону, путем задания набора шаблонов, как например, "Строка по шаблону 'Задача%' или 'ЗАДАЧА%'".

Вопрос. *Что собой представляет шаблон и операция LIKE?* Шаблон представляет собой строку, в которой используется символ '%' для обозначения "любого сочетания символов". Для указания в шаблоне символа '%' как такового необходимо использовать комбинацию символов '%%'. Операция LIKE представляет собой предикат, принимающий истинные значения в тех случаях, когда строка (первый параметр) удовлетворяет шаблону (второй параметр). Данная операция детально описана в спецификации SCQL. Из данного пояснения можно сделать вывод, что механизм шаблонов SCQL аналогичен механизму шаблонов SQL.

### 3.1.8.3.13 Набор значений

Набор значений представляет собой разновидность ограниченного типа, параметризованного списком исходных значений. Для набора значений допустимыми являются те и только те исходные значения соответствующего исходного типа, которые перечислены в списке в явном виде.

*Пример. "Целое число из набора {3, 8, 15}" представляет собой набор значений, построенный на исходном типе "Целое число". Допустимыми значениями для этого типа являются 3, 8 и 15.*

Набор значений является Типом ≡  
 [FOR ALL (Набор значений–Тип) EXISTS (Тип–Ограниченный тип)  
 ] <Набор значений является ограниченным типом.>  
Значение набора является Значением ≡  
 [(Значение набора–Значение) EQUALS (Значение–Тип–Набор значений)  
 ] <Значение набора принадлежит одному из наборов значений.>  
Набор значений построен на Исходном типе →  
 [(Набор значений, Исходный тип) EQUALS  
 (Набор значений–Тип–Ограниченный тип–Исходный тип)]  
Значение набора приводится к Исходному значению →  
 [(Значение набора, Исходное значение) EQUALS  
 (Значение набора–Значение–Допустимое значение–Исходное значение)]

Набор значений параметризован списком Исходных значений  
[FOR ALL (Набор значений, Исходное значение)  
EXISTS (Исходное значение–Значение–Тип–Исходный тип–Набор значений)  
] <Набор значений может быть параметризован только исходными значениями,  
принадлежащими исходному типу, на котором построен данный набор.>  
[(Набор значений–Исходное значение) EQUALS  
(Набор значений–Тип–Значение–Значение набора–Исходное значение)  
] <Для каждого исходного значения, параметризующего набор значений,  
существует значение набора, приводимое к этому исходному значению.>

#### **3.1.8.3.14 Набор чисел**

Набор чисел является набором значений, относящимся к метатипу "Набор чисел", исходным типом которого является число.

Набор чисел является Набором значений ≡  
[(Набор чисел–Набор значений–Тип) EQUALS (Тип–Метатип = "Набор чисел")  
] <Набор чисел относится к метатипу "Набор чисел".>  
[FOR ALL (Набор чисел–Набор значений–Исходный тип)  
EXISTS (Исходный тип–Тип–Простой тип = "Число")  
] <Исходный тип набора чисел является числом.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами чисел, например, "Число из набора {3.5, 8.2, 15}".

#### **3.1.8.3.15 Набор целых чисел**

Набор целых чисел является набором значений, относящимся к метатипу "Набор целых чисел", исходным типом которого является целое число.

Набор целых чисел является Набором значений ≡  
[(Набор целых чисел–Набор значений–Тип) EQUALS  
(Тип–Метатип = "Набор целых чисел")  
] <Набор целых чисел относится к метатипу "Набор целых чисел".>  
[FOR ALL (Набор целых чисел–Набор значений–Исходный тип)  
EXISTS (Исходный тип–Тип–Простой тип = "Целое число")  
] <Исходный тип набора целых чисел является целым числом.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами целых чисел, например, "Целое число из набора {3, 8, 15}".

#### **3.1.8.3.16 Набор дат**

Набор дат является набором значений, относящимся к метатипу "Набор дат", исходным типом которого является простой тип "Дата".

Набор дат является Набором значений ≡  
[(Набор дат–Набор значений–Тип) EQUALS (Тип–Метатип = "Набор дат")  
] <Набор дат относится к метатипу "Набор дат".>  
[FOR ALL (Набор дат–Набор значений–Исходный тип)  
EXISTS (Исходный тип–Тип–Простой тип = "Дата")  
] <Исходный тип набора дат является датой.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами дат, например, "Дата из набора {'01.01.2004', '01.01.2005', '01.01.2006'}".

#### **3.1.8.3.17 Набор строк**

Набор строк является набором значений, относящимся к метатипу "Набор строк", исходным типом которого является простой тип "Строка".

Набор строк является Набором значений ≡  
[(Набор строк–Набор значений–Тип) EQUALS (Тип–Метатип = "Набор строк")  
] <Набор строк относится к метатипу "Набор строк".>  
[FOR ALL (Набор строк–Набор значений–Исходный тип)  
EXISTS (Исходный тип–Тип–Простой тип = "Строка")  
] <Исходный тип набора строк является строкой.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами строк, например, "Строка из набора {'a', 'ab', 'abc'}".

### 3.1.8.3.18 Набор кодов

Набор кодов является набором значений, относящимся к метатипу "Набор кодов", исходным типом которого является один из справочников.

Набор кодов является Набором значений ≡  
[(Набор кодов–Набор значений–Тип) EQUALS (Тип–Метатип = "Набор кодов")  
] <Набор кодов относится к метатипу "Набор кодов".>  
[FOR ALL (Набор кодов–Набор значений–Исходный тип)  
EXISTS (Исходный тип–Тип–Справочник)  
] <Исходный тип набора кодов является справочником.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами кодов одного из справочников, например, "Код из набора кодов {1, 2, 5} по справочнику 'Должность'".

### 3.1.8.3.19 Набор диапазонов

Набор диапазонов представляет собой ограниченный тип, параметризованный множеством диапазонов. Левая и правая границы этих диапазонов обязательно относятся к исходному типу. Допустимыми значениями набора диапазонов являются все те значения исходного типа, которые удовлетворяют любому из диапазонов. Смысл, который вкладывается в словосочетание "удовлетворяет диапазону" несколько различен для непрерывных и дискретных диапазонов, что раскрывается в последующих пунктах.

Набор диапазонов является Типом ≡  
[FOR ALL (Набор диапазонов–Тип) EXISTS (Тип–Ограниченный тип)  
] <Набор диапазонов является ограниченным типом.>  
Значение набора диапазонов является Значением ≡  
[(Значение набора диапазонов–Значение) EQUALS  
(Значение–Тип–Набор диапазонов)  
] <Значение набора диапазонов принадлежит одному из наборов диапазонов.>  
Набор диапазонов определяется Диапазонами  
[FOR ALL (SELECT Набор диапазонов, Значение  
FROM (Набор диапазонов–Диапазон–Левая граница–Значение)  
UNION SELECT Набор диапазонов, Значение  
FROM (Набор диапазонов–Диапазон–Правая граница–Значение))  
EXISTS (Набор диапазонов–Тип(ограниченный)–Ограниченный тип–  
Исходный тип–Тип(исходный)–Значение)  
] <И левая, и правая границы являются значениями исходного типа для диапазона значений.>

Вопрос. *Зачем было введено понятие диапазонов значений, ведь диапазон значений является частным случаем набора диапазонов?* Да, действительно, диапазон значений является частным случаем, но в результате этого он имеет и более простую структуру. Единственный смысл введения отдельных метатипов для диапазонов значений заключается в упрощении их использования по сравнению с более общими метатипами,

основанными на наборах диапазонов. Зачастую дробление интервалов не требуется, и в этом случае может использоваться диапазон значений вместо набора диапазонов.

### 3.1.8.3.20 Набор непрерывных диапазонов

Набор непрерывных диапазонов представляет собой такой набор диапазонов, каждый из диапазонов которого имеет тип границ.

Набор непрерывных диапазонов является Типом ≡  
[FOR ALL (Набор непрерывных диапазонов–Тип) EXISTS (Тип–Набор диапазонов)  
] <Набор непрерывных диапазонов является набором диапазонов.>  
[FOR ALL (Набор непрерывных диапазонов–Тип–Набор диапазонов–Диапазон)  
EXISTS (Диапазон–Тип границ)  
] <Для каждого диапазона из набора непрерывных диапазонов обязательно задан тип границ.>

Допустимыми значениями набора непрерывных диапазонов являются те значения исходного типа, которые удовлетворяют хотя бы одному параметризующему диапазону. Значение удовлетворяет диапазону, если оно находится между его левой и правой границами, причем включение границ внутрь диапазона определяется их типом.

[SELECT Набор непрерывных диапазонов, Число  
FROM (Набор непрерывных диапазонов–Тип–Значение–Число)  
EQUALS  
SELECT Набор непрерывных диапазонов, Число  
FROM ((Набор непрерывных диапазонов–Тип(ограниченный)–  
Ограниченный тип–Исходный тип–Тип(исходный)–Значение–Число),  
(Тип(ограниченный)–Набор диапазонов–Диапазон–[Левая граница, Правая  
граница, Тип границ]),  
Значения между (Левая граница, Правая граница, Тип границ, Значение))  
] <В набор непрерывных диапазонов входят только те значения исходного типа, которые находятся между левой и правой границей хотя бы одного диапазона (в соответствии с порядком кодирующих чисел), с включением/исключением границ в соответствии с типом границ.>

Пояснение. В данной формализации используется предикат, определенный ранее в пункте 3.1.8.3.5.

### 3.1.8.3.21 Набор дискретных диапазонов

Набор дискретных диапазонов представляет собой такой набор диапазонов, все диапазоны которого не имеют типа границ.

Набор дискретных диапазонов является Типом ≡  
[FOR ALL (Набор дискретных диапазонов–Тип) EXISTS (Тип–Набор диапазонов)  
] <Набор дискретных диапазонов является набором диапазонов.>  
[FOR ALL (Набор непрерывных диапазонов–Тип–Набор диапазонов–Диапазон)  
NOT EXISTS (Диапазон–Тип границ)  
] <Для набора дискретных диапазонов тип границ не задается.>

Допустимыми значениями набора дискретных диапазонов являются те значения исходного типа, которые удовлетворяют хотя бы одному параметризующему диапазону. Значение удовлетворяет диапазону, если оно находится между его левой и правой границами включительно.

[SELECT Набор дискретных диапазонов, Число  
FROM (Набор дискретных диапазонов–Тип–Значение–Число)  
EQUALS  
SELECT Набор дискретных диапазонов, Число  
FROM ((Набор дискретных диапазонов–Тип(ограниченный)–Ограниченный

тип–Исходный тип–Тип(исходный)–Значение–Число),  
(Тип(ограниченный)–Набор диапазонов–Диапазон–[Левая граница, Правая граница]),  
(Левая граница–Значение(левое)–Число(левое)),  
(Правая граница–Значение(правое)–Число(правое)),  
(Число(левое)<= Число AND Число <= Число(правое)))

] <В набор дискретных диапазонов входят только те значения исходного типа, которые находятся между левой и правой границей включительно (в соответствии с порядком кодирующих чисел) для хотя бы одного диапазона.>

### **3.1.8.3.22 Набор диапазонов чисел**

Набор диапазонов чисел представляет собой набор непрерывных диапазонов, относящийся к метатипу "Набор диапазонов чисел", исходным типом которого является число.

Набор диапазонов чисел является Типом ≡

[(Набор диапазонов чисел, Тип) EQUALS

(Тип–Метатип = "Набор диапазонов чисел")

] <Набор диапазонов чисел относится к метатипу "Набор диапазонов чисел".>

[FOR ALL (Набор диапазонов чисел, Тип)

EXISTS (Тип–Набор непрерывных диапазонов)

] <Набор диапазонов чисел является набором непрерывных диапазонов.>

[FOR ALL (Набор диапазонов чисел–Тип) EXISTS (Тип–Ограниченный Тип–Исходный тип–Тип(исходный)–Простой тип = "Число")

] <Исходным типом набора диапазонов чисел является число.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами диапазонов чисел, например, "Число принадлежащее одному из диапазонов {(0.5, 5], [8,10)}".

### **3.1.8.3.23 Набор диапазонов целых чисел**

Набор диапазонов целых чисел представляет собой набор дискретных диапазонов, относящийся к метатипу "Набор диапазонов целых чисел", исходным типом которого является целое число.

Набор диапазонов целых чисел является Типом ≡

[(Набор диапазонов целых чисел, Тип) EQUALS

(Тип–Метатип = "Набор диапазонов целых чисел")

] <Набор диапазонов целых чисел относится к метатипу "Набор диапазонов целых чисел".>

[FOR ALL (Набор диапазонов целых чисел, Тип)

EXISTS (Тип–Набор дискретных диапазонов)

] <Набор диапазонов целых чисел является набором дискретных диапазонов.>

[FOR ALL (Набор диапазонов целых чисел–Тип) EXISTS (Тип–Ограниченный Тип–Исходный тип–Тип(исходный)–Простой тип = "Целое число")

] <Исходным типом набора диапазонов целых чисел является целое число.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами диапазонов целых чисел, например, "Целое число принадлежащее одному из диапазонов {[0, 5], [8,10]}" (границы диапазонов включены, так как диапазоны дискретные).

### **3.1.8.3.24 Набор диапазонов дат**

Набор диапазонов дат представляет собой набор непрерывных диапазонов, относящийся к метатипу "Набор диапазонов дат", исходным типом которого является простой тип "Дата".

Набор диапазонов дат является Типом  $\equiv$   
[(Набор диапазонов дат, Тип) EQUALS (Тип–Метатип = "Набор диапазонов дат")  
] <Набор диапазонов дат относится к метатипу "Набор диапазонов дат".>  
[FOR ALL (Набор диапазонов дат, Тип)  
EXISTS (Тип–Набор непрерывных диапазонов)  
] <Набор диапазонов дат является набором непрерывных диапазонов.>  
[FOR ALL (Набор диапазонов дат–Тип) EXISTS (Тип–Ограниченный Тип–Исходный тип–Тип(исходный)–Простой тип = "Дата")  
] <Исходным типом набора диапазонов дат является дата.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами диапазонов дат, например, "Дата принадлежащая одному из диапазонов {'01.01.2004', '01.01.2005'}, ['01.01.2006', '01.01.2007']".

### 3.1.8.3.25 Набор диапазонов кодов

Набор диапазонов кодов представляет собой набор дискретных диапазонов, относящийся к метатипу "Набор диапазонов кодов", исходным типом которого является один из справочников.

Набор диапазонов кодов является Типом  $\equiv$   
[(Набор диапазонов кодов, Тип) EQUALS  
(Тип–Метатип = "Набор диапазонов кодов")  
] <Набор диапазонов кодов относится к метатипу "Набор диапазонов кодов".>  
[FOR ALL (Набор диапазонов кодов, Тип)  
EXISTS (Тип–Набор дискретных диапазонов)  
] <Набор диапазонов кодов является набором дискретных диапазонов.>  
[FOR ALL (Набор диапазонов кодов–Тип)  
EXISTS (Тип–Ограниченный Тип–Исходный тип–Тип(исходный)–Справочник)  
] <Исходным типом набора диапазонов кодов является справочник.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами диапазонов кодов, например, "Коды справочника 'Должность' принадлежащие одному из диапазонов {[3, 8], [15, 20]}" (границы диапазонов включены, так как диапазоны дискретные). В отличие от других наборов диапазонов, набор диапазонов кодов дополнительно параметризуется исходным типом справочника, на основе которого он объявляется.

### 3.1.8.4 Составной тип

Кроме простых типов, значения которых рассматриваются как атомарные и не имеющие собственной структуры, существуют также составные типы, значения которых представляют собой некую структуру из вложенных значений. Все значения составных типов являются составными значениями, и наоборот, все составные значения относятся к составным типам (см. описание составных значений выше).

Составной тип является Типом  $\equiv$   
[COUNT(Составной тип–Тип–Простой тип) = 0  
] <Составной тип не является простым типом.>  
[(Составное значение–Значение) EQUALS (Значение–Тип–Составной тип)  
] <Все составные значения относятся к составному типу.>

Составной тип представляет собой последовательность вложенных в него типов. В рамках составного типа у каждого вложенного типа своя уникальная позиция. Вложенный

тип может включаться в различные составные типы без ограничений, но не допускается циклическое вложение (как непосредственное, так и опосредованное) типов друг в друга.

Вложенный тип является Типом =

Составной тип состоит из Вложенного типа в Позиции

[SEQUENCE Позиция, 1, 1 WITHIN Составной тип

OVER (Составной тип, Вложенный тип, Позиция)] <Позиции составного типа представляют собой последовательность целых чисел от одного через один.>

[NO\_CYCLE SEQ(Тип(вложенный), Тип(составной)) OVER

(Тип(составной)–(Составной тип, Позиция, Вложенный тип)–Тип(вложенный))

] <В направленном графе вложения типов не должно быть циклов.>

Каждой позиции составного типа должно быть поставлено в соответствие то понятие, значения которого находятся в данной позиции. В рамках составного типа понятия должны быть уникальны, так как они могут использоваться для идентификации элементов составного типа.

*Пример. Тип "Структура, построенная на понятиях 'паспорт' типа строка, 'целое число(возраст)' типа целое число и 'целое число(детей)' типа целое число" может принимать значения: (паспорт => '0001№000001', целое число(возраст) => 28, целое число(детей) => 1), (паспорт => '0001№000002', целое число(возраст) => 35, целое число(детей) => 3) и др.*

*Определение структуры приводится ниже.*

Позиция Составного типа соответствует Понятию

[(Позиция, Составной тип, Понятие): Позиция = (Составной тип) = Понятие

] <В рамках составного типа каждой позиции взаимнооднозначно соответствует одно понятие.>

[(Составной тип, Позиция, Вложенный тип) EQUALS

(Составной тип, Позиция, Понятие)

] <Для каждой позиции составного типа определено некоторое понятие.>

Пояснение. Соответствие понятия и позиции составного типа позволяет раскрыть содержимое составного типа в терминах понятий, в результате чего нет необходимости вводить искусственные обозначения для позиций составных типов, например, структур (см. описание структур ниже).

Вложенный тип, который располагается в некоторой позиции составного типа, и тот тип, в виде которого представляется понятие, расположенное в той же позиции, должны совпадать. Учитывая эту особенность, при описании составных типов можно не указывать типы, к которым относятся их позиции, так как они равны типам, в виде которых представляются соответствующие понятия.

*Пример. Тип из предыдущего примера может быть описан более кратким образом: "Структура, построенная на понятиях 'паспорт', 'целое число(возраст)', 'целое число(детей)'".*

[FOR ALL ((Позиция, Понятие, Составной тип), ((Составной тип, Позиция, Вложенный тип)–Тип)) EXISTS (Понятие–Тип)

] <Позиция составного типа должна относиться к тому же вложенному типу, в виде которого представляется понятие, находящееся в этой позиции.>

Пояснение. Различие типов понятия и соответствующей позиции структуры приведет к неоднозначности трактовки значений структуры. Поэтому было принято решение избежать данной неоднозначности через описанное ограничение.

Вывод. В рамках SCM составные типы могут рассматриваться не только как последовательность вложенных типов, но и альтернативным образом: как последовательность понятий (а через них подразумеваются вложенные типы).

Вопрос. Почему бы не ограничиться представлением типа, как последовательности понятий, зачем определять вложенные типы? Действительно, составной тип может быть эквивалентно представлен и как последовательность

вложенных типов с понятиями, и просто как последовательность понятий. Так как понятие определяет семантический элемент предметной области, а тип – способ представления данных в вычислительной системе, то, исходя из общепринятой терминологии, более правильным считать, что составной тип состоит из вложенных типов, что и определяет способ его представления. А возможность представления составного типа в виде последовательности понятий является ничем иным, как дополнительным следствием свойств SCM. При этом на практике целесообразно описывать все составные типы единым способом, и второй способ является, очевидно, более удобным, так как не требует указания и вложенных типов, и понятий одновременно; достаточно перечислить только понятия (см. предыдущий пример).

Вопрос. *Тогда почему бы не исключить составные типы вообще, оставив простые типы и возможность непосредственного вложения понятий друг в друга?* Такое решение не было принято по причине того, что оно смешивает вопросы семантики понятий и представления понятий в вычислительной системе. Для разделения этих вопросов потребовалось отдельно рассмотреть типы, простые и составные, как средства представления данных в вычислительных системах, и отдельно, в явном виде, определить взаимосвязь типов и понятий. При этом, свойства этой взаимосвязи таковы, что составные типы могут быть описаны как последовательности понятий, но это не означает, что понятия служат для представления данных в рамках вычислительных систем.

Содержимое составного значения однозначно соответствует самому составному значению в рамках каждого типа. Другими словами, в рамках одного типа не должно быть нескольких составных значений, имеющих одинаковое содержимое. Под содержимым составного значения подразумевается множество пар вложенных значений и их позиций.

```
[(SELECT Тип, Составное значение, SET(<Вложенное значение, Позиция>
      FROM (Вложенное значение, Позиция, Составное значение)–Значение–Тип)
): Составное значение ≡(Тип)≡ SET(<Вложенное значение, Позиция>)
] <Каждое составное значение взаимнооднозначно определено множеством пар
(вложенное значение, позиция) в рамках соответствующего составного типа.>
```

Вопрос. *Зачем нужна уникальность содержимого составных значений в рамках типа?* Общепринят тот факт, что если два значения относятся к одному типу и имеют одинаковое содержимое, то они считаются эквивалентными. Решение запретить дублирование эквивалентных составных значений было принято с целью определить в явном виде данное правило эквивалентности – эквивалентные составные значения представляют собой одно и то же значение.

#### 3.1.8.4.1 Структура

Структура является составным типом и относится к соответствующему метатипу "Структура". Количество позиций в значениях структуры фиксировано и определяется составом вложенных в нее типов. Никаких ограничений на разнообразие вложенных типов не налагается.

```
Структура является Составным типом ≡
[(Структура, Составной тип) EQUALS
(Составной тип–Тип–Метатип = "Структура")
] <Структура относится к метатипу "Структура".>
```

Все составные значения структур являются векторными значениями, то есть существенную роль играет позиция каждого вложенного значения в их рамках. Каждое значение структуры должно подчиняться ей, то есть должно иметь то же количество вложенных значений, что и количество вложенных типов структуры, и в каждой позиции значения структуры должно содержать вложенное значение того типа, который располагается в той же позиции данной структуры.

```
[FOR ALL (Структура–Составной тип–Тип–Значение–Составное значение)
 EXISTS ((Составное значение–Векторное значение) WHEN
```

FOR ALL (Составное значение, Вложенное значение, Позиция)  
EXISTS (Вложенное значение–Значение–Тип(вложенный)–  
(Вложенный тип, Позиция, Составной тип)))

] <Каждое составное значение структуры является векторным значением, в каждой позиции которого находятся вложенное значение, относящееся именно к тому типу, который вложен в ту же позицию данной структуры.>

Пояснение. Понятие структуры во многом аналогично понятию класса в объектно-ориентированном программировании (его структурной составляющей, без поведения). Единственным принципиальным отличием является отсутствие собственного обозначения для позиций структуры (тогда как в классе все переменные уникально поименованы в рамках класса). Отсутствие собственных обозначений объясняется тем требованием, что работа с SCM должна осуществляться исключительно в понятиях предметной области. Поэтому вместо собственного обозначения позиции структуры используется понятие.

Аналогично классам объектно-ориентированного программирования, структуры SCM могут наследоваться друг от друга. При этом каждая производная структура должна наследоваться лишь от одной базовой структуры.

Производная структура является Структурой ≡

Базовая структура является Структурой ≡

Производная структура наследуется от Базовой структуры →

Только часть вложенных типов производной структуры являются собственными, остальные наследуются от базовой структуры. Собственные вложенные типы указываются для структуры в явном виде, остальные подразумеваются исходя из базовой структуры.

Структура имеет собственный Вложенный тип в Позиции

[(Структура, Вложенный тип, Позиция): (Структура, Позиция) → Вложенный тип

] <В одной позиции структуры может находиться только один вложенный тип.>

В конечном счете производная структура состоит сначала из вложенных типов базовой структуры, в тех же позициях, в которых они располагаются в базовой структуре, а затем из собственных вложенных типов, в последующих позициях.

[(Структура–(Составной тип(производный), Вложенный тип, Позиция)) EQUALS

((Структура, Вложенный тип, Позиция) UNION

(Структура–Производная структура–Базовая структура–Структура(базовая)–  
(Составной тип(базовый), Вложенный тип, Позиция))

] <Общее множество пар (вложенный тип, позиция) производной структуры равно объединению собственного множества этих пар производной структуры и общего множества пар базовой структуры.>

Пояснение. Несмотря на составную природу производных структур, позиции в ней должны нумероваться от 1 через 1, подобно любому другому составному типу (см. описание составного типа выше).

В графе наследования структур не должно быть циклов, иначе в соответствии с приведенным определением производные структуры будут содержать бесконечное количество вложенных типов, что недопустимо.

[NO\_CYCLE SET(Структура(базовая), Структура(производная))

OVER (Структура(базовая)–Базовая структура–Производная структура–  
Структура(производная))

] <В графе наследования структур не должно быть циклов.>

Иллюстрация. На рисунке 7 представлен схематичный пример базовой и производной структур. Из рисунка видно, что один и тот же вложенный тип может находиться в различных позициях структуры; все позиции пронумерованы уникально от 1 через 1; каждой позиции обязательно соответствует уникальное понятие; собственные вложенные типы и позиции производного типа дополняют базовый типа, а не перекрывают его.

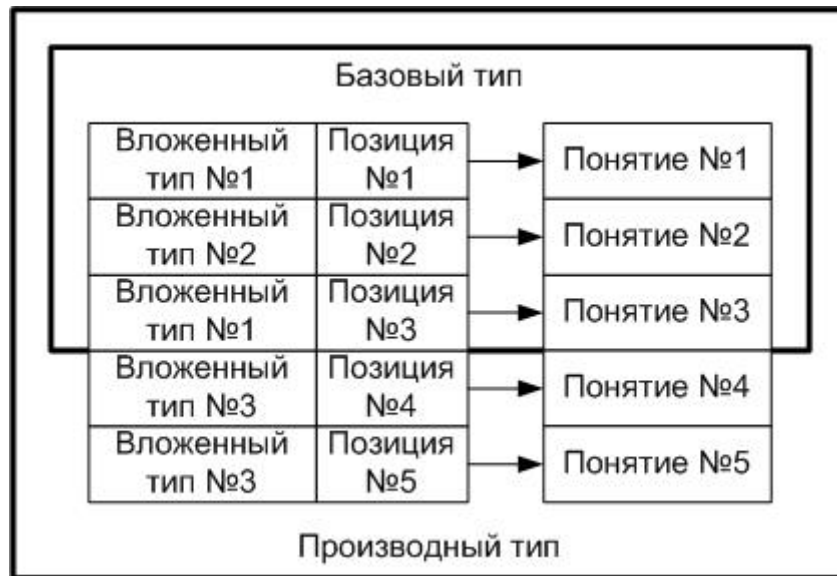


Рисунок 7 Пример базовой и производной структур

### 3.1.8.4.2 Вектор

Вектор представляет собой составной тип и относится к соответствующему метатипу "Вектор".

Вектор является Составным типом =  
 [(Вектор, Составной тип) EQUALS (Составной тип–Тип–Метатип = "Вектор")  
 ] <Вектор относится к метатипу "Вектор".>

Вектор имеет единственный вложенный тип, и все вложенные значения его составных значений относятся к этому типу. Количество вложенных значений может быть произвольным.

*Пример. Тип "Вектор целых чисел понятия 'Возраст'", созданный в соответствии с метатипом "Вектор", может принимать, например, такие значения: <1, 3, 5, 5, 3>, <8, 6, 8>. Обращаем ваше внимание на тот факт, что целые числа в рамках данных векторных значений могут повторяться в различных позициях, и их количество может быть произвольным. Учитывая свойства составных типов, тот же тип может быть объявлен более простым образом как "Вектор 'Возрастов'".*

Вектор построен на Type →  
 [(Вектор, Тип) EQUALS  
 (Вектор–(Составной тип, Вложенный тип, Позиция))  
 ] <Каждый вектор состоит из единственного вложенного типа – того, на котором он построен.>  
 [FOR ALL (Вектор–Составной тип–Тип–Значение–  
 (Составное значение, Вложенное значение, Позиция))  
 EXISTS (Вложенное значение–Значение–Тип(базовый)–Вектор))  
 ] <Каждое составное значение вектора является таким значением, вложенные значения которого относятся только к базовому вложенному типу.>

Иллюстрация. На рисунке 8 приводится схематичный пример вектора, построенного на некотором вложенном типе, и его значений. Здесь эллипсы – значения вектора, окружности – значения вложенного типа, кружки с числами – позиции вложенных значений в рамках значений вектора. Как можно видеть, значения вектора представляют собой все возможные последовательности значений вложенного типа, в отличие от значений структур, в которых количество вложенных значений для каждого составного значения строго определено.

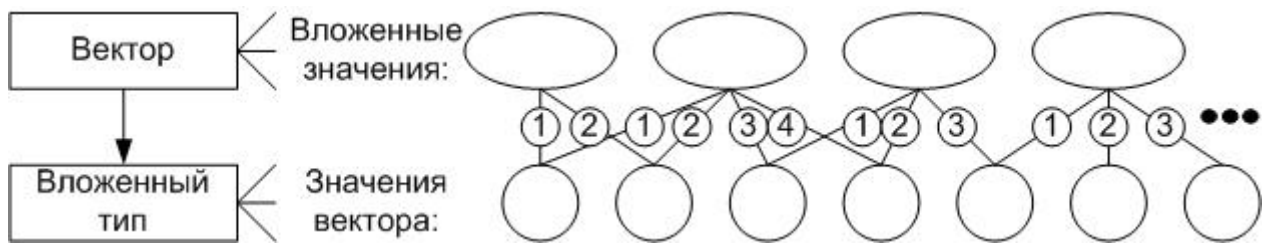


Рисунок 8 Схематичный пример вектора и его значений

### 3.1.8.4.3 Множество

Множество представляет собой составной тип и относится к соответствующему метатипу "Множество".

Множество является Составным типом  $\equiv$   
 [(Множество, Составной тип) EQUALS  
 (Составной тип–Тип–Метатип = "Множество")  
 ] <Множество относится к метатипу "Множество".>

Множество, как и вектор, имеет единственный вложенный тип, и все вложенные значения его составных значений относятся к этому типу; количество вложенных значений может быть произвольным.

*Пример. Тип "Множество целых чисел понятия 'Возраст'", созданный в соответствии с метатипом "Множество", может принимать, например, такие значения: {1, 3, 5}, {8, 6, 19, 25}. Обращаем ваше внимание на тот факт, что целые числа в рамках данных значений множеств не повторяются, и их количество может быть произвольным. Учитывая свойства составных типов, тот же тип может быть объявлен более простым образом как "Множество 'Возрастов'".*

Множество построено на Type  $\rightarrow$   
 [(Множество, Тип) EQUALS  
 (Множество–(Составной тип, Вложенный тип, Позиция))  
 ] <Каждое множество состоит из единственного вложенного типа – того, на котором оно построено.>  
 [FOR ALL (Множество–Составной тип–Тип–Значение–  
 (Составное значение, Вложенное значение, Позиция))  
 EXISTS (Вложенное значение–Значение–Тип(базовый)–Множество))  
 ] <Каждое составное значение множества является значением, вложенные значения которого относятся только к базовому вложенному типу.>

Иллюстрация. На рисунке 9 приводится схематичный пример множества, построенного на некотором вложенном типе, и его значений. Обозначения сохранены те же, что и на рисунке 8: эллипсы – значения вектора, окружности – значения вложенного типа. Из рисунка видно, что у вложенных значений отсутствуют позиции в рамках составного значения (все позиции равны 1, что равносильно отсутствию), в отличие от значений вектора. Таким образом, значения множества представляют собой все возможные множества значений вложенного типа.

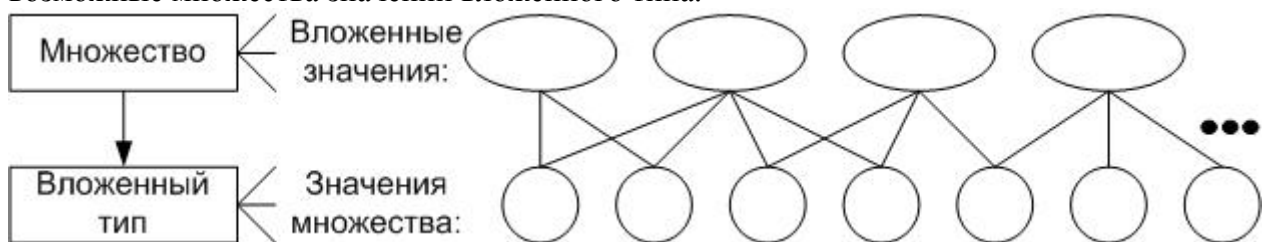


Рисунок 9 Схематичный пример множества и его значений

### 3.1.8.4.4 Ограниченный составной тип

Ограниченный составной тип строится на некотором исходном типе и допускает те его значения, размерность которых находится в определенных рамках. Ограниченный составной тип представляет собой ограниченный тип, исходным типом которого является составной тип.

Ограниченный составной тип является Типом  $\equiv$   
[FOR ALL (Ограниченный составной тип–Тип) EXISTS (Тип–Составной тип)  
] <Ограниченный составной тип является составным типом.>  
[FOR ALL (Ограниченный составной тип–Тип)  
EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Составной тип)  
] <Ограниченный составной тип является ограниченным типом, исходным типом  
которого является составной тип.>

Ограниченный составной тип предназначен для ограничения возможной размерности его составных значений. Поэтому он характеризуется двумя параметрами: минимальной и максимальной размерностями.

Ограниченный составной тип ограничен Минимальной размерностью  $\rightarrow$   
[Минимальной размерностью = Целое число]  
Ограниченный составной тип ограничен Максимальной размерностью  $\rightarrow$   
[Максимальная размерностью = Целое число]

Основным свойством ограниченного составного типа является то, что все его допустимые значения имеют размерность между заданными минимальной и максимальной размерностями включительно.

*Пример. Тип "Вектор целых чисел понятия 'Расстояние' размерностью от 3 до 5" допускает значение <1,3,5,3,5>, но не допускает значения <1,3,5,3,5,6> или <1,3>.*

```
[SELECT Ограниченный составной тип, Число
FROM ((Ограниченный составной тип–Тип–Ограниченный тип–Исходный тип),
      (Исходный тип–Тип(исходный)–Значение–[Составное значение, Число]),
      (Ограниченный составной тип–[Минимальная размерность, Максимальная
      размерность]))
WHEN {COUNT(Составное значение, Вложенное значение, Позиция)
      BETWEEN Минимальная размерность AND Максимальная размерность}
EQUALS
SELECT Ограниченный составной тип, Число
FROM (Ограниченный составной тип–Тип–Значение–Число)
] <Все значения исходного типа, которые имеют размерность в заданных пределах
(включая их), приводятся к соответствующим допустимым значениям
ограниченного составного типа.>
```

Следствием свойств ограниченного типа является то, что в рамках ограниченного составного типа все исходные значения имеют те же самые вложенные значения в тех же позициях, что и соответствующие допустимые значения (см. описание ограниченного типа выше, а именно ограничение одинакового кодирования допустимых и исходных значений).

```
[SELECT Значение, SET(<Вложенное значение, Позиция>)
FROM ((Значение–Тип–Ограниченный составной тип),
      (Значение–Допустимое значение–Исходное значение–Значение(исходное)),
      (Значение(исходное)–(Составное значение, Вложенное значение, Позиция)))
EQUALS
SELECT Значение, SET(<Вложенное значение, Позиция>)
FROM ((Составное значение–Значение–Тип–Ограниченный составной тип),
      (Составное значение, Вложенное значение, Позиция))
```

] <Исходное значение имеет те же самые вложенные значения в тех же позициях, что и соответствующее допустимое значение ограниченного составного типа.>

Пояснение. Данное ограничение означает эквивалентность соответствующих исходных и допустимых значений.

#### **3.1.8.4.5 Ограниченный вектор**

Ограниченный вектор представляет собой ограниченный составной тип, исходным типом которого является вектор. Ограниченный вектор относится к соответствующему метатипу "Ограниченный вектор". Ограниченный вектор позволяет объявлять типы векторов с размерностью в заданном диапазоне.

Ограниченный вектор является Типом ≡  
[(Ограниченный вектор, Тип) EQUALS (Тип–Метатип = "Ограниченный вектор")  
] <Ограниченный вектор относится к метатипу "Ограниченный вектор".>  
[FOR ALL (Ограниченный вектор, Тип)  
EXISTS (Тип–Ограниченный составной тип))  
] <Ограниченный вектор является ограниченным составным типом.>  
[FOR ALL (Ограниченный вектор–Тип)  
EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Вектор)  
] <Исходным типом ограниченного вектора является вектор.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся ограниченными векторами, например, "Вектор целых чисел понятия 'Расстояние' размерностью от 3 до 5 " (размерности 3 и 5 допустимы).

#### **3.1.8.4.6 Ограниченное множество**

Ограниченное множество представляет собой ограниченный составной тип, исходным типом которого является множество. Ограниченное множество относится к соответствующему метатипу "Ограниченное множество" и позволяет объявлять типы множеств с размерностью в заданном диапазоне.

Ограниченное множество является Типом ≡  
[(Ограниченное множество, Тип) EQUALS  
(Тип–Метатип = "Ограниченное множество")  
] <Ограниченное множество относится к метатипу "Ограниченное множество".>  
[FOR ALL (Ограниченное множество, Тип)  
EXISTS (Тип–Ограниченный составной тип))  
] <Ограниченное множество является ограниченным составным типом.>  
[FOR ALL (Ограниченное множество–Тип)  
EXISTS (Тип–Ограниченный тип–Исходный тип–Тип(исходный)–Множество)  
] <Исходным типом ограниченного множества является множество.>

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся ограниченными множествами, например, "Множество целых чисел понятия 'Расстояние' размерностью от 3 до 5 " (размерности 3 и 5 допустимы).

#### **3.1.8.4.7 Набор значений структуры**

Набор значений структуры представляет собой набор значений, исходным типом которого является некоторая структура. Набор значений структуры относится к соответствующему метатипу "Набор значений структуры" и позволяет объявлять типы, допускающие только те значения исходной структуры, которые перечислены в рамках данного набора в явном виде.

Набор значений структуры является Типом ≡  
[FOR ALL (Набор значений структуры–Тип) EXISTS (Тип–Набор значений)  
] <Набор значений структуры является набором значений.>

```

[(Набор значений структуры–Тип) EQUALS
(Тип–Метатип = "Набор значений структуры")
]<Набор значений структуры относится к метатипу "Набор значений структуры".>
[FOR ALL (Набор значений структуры–Тип–Набор значений–Исходный тип)
EXISTS (Исходный тип–Тип–Составной тип–Структура)
]<Исходный тип набора значений структуры является структурой.>

```

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами значений структуры, например, "Набор значений структуры, построенной на понятиях 'паспорт', 'целое число(возраст)', 'целое число(детей): {(паспорт => '0001№000001', целое число(возраст) => 28, целое число(детей) => 1), (паспорт => '0001№000002', целое число(возраст) => 35, целое число(детей) => 3)}". Конечно, обозначение типа может быть гораздо короче, чем его расшифровка, приведенная в кавычках.

### 3.1.8.4.8 Набор значений вектора

Набор значений вектора представляет собой набор значений, исходным типом которого является некоторый вектор. Набор значений вектора относится к соответствующему метатипу "Набор значений вектора" и позволяет объявлять типы, допускающие только те значения исходного вектора, которые перечислены в рамках данного набора в явном виде.

```

Набор значений вектора является Типом ≡
[FOR ALL (Набор значений вектора–Тип) EXISTS (Тип–Набор значений)
]<Набор значений вектора является набором значений.>
[(Набор значений вектора–Тип) EQUALS
(Тип–Метатип = "Набор значений вектора")
]<Набор значений вектора относится к метатипу "Набор значений вектора".>
[FOR ALL (Набор значений вектора–Тип–Набор значений–Исходный тип)
EXISTS (Исходный тип–Тип–Составной тип–Вектор)
]<Исходный тип набора значений вектора является вектором.>

```

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами значений вектора, например, "Набор значений вектора, построенного на понятии 'Возраст': {<1, 3, 5, 5, 3>, <8, 6, 8>}".

### 3.1.8.4.9 Набор значений множества

Набор значений множества представляет собой набор значений, исходным типом которого является некоторое множество. Набор значений множества относится к соответствующему метатипу "Набор значений множества" и позволяет объявлять типы, допускающие только те значения исходного множества, которые перечислены в рамках данного набора в явном виде.

```

Набор значений множества является Типом ≡
[FOR ALL (Набор значений множества–Тип) EXISTS (Тип–Набор значений)
]<Набор значений множества является набором значений.>
[(Набор значений множества–Тип) EQUALS
(Тип–Метатип = "Набор значений множества")
]<Набор значений множества относится к метатипу "Набор значений множества".>
[FOR ALL (Набор значений множества–Тип–Набор значений–Исходный тип)
EXISTS (Исходный тип–Тип–Составной тип–Множество)
]<Исходный тип набора значений множества является множеством.>

```

Пояснение. Пользователь SCM может объявлять конкретные типы, являющиеся наборами значений множества, например, "Набор значений множества, построенного на понятии 'Возраст': {{1, 3, 5}, {8, 6, 19, 25}}".

## 3.2 Свойства SCM-схемы

### 3.2.1 Прямое использование понятий предметной области

SCM-схема не содержит искусственных введений, которые бы сложным образом соотносились с понятиями предметной области.

*Пример. Атрибут реляционной схемы является ячейкой хранения данных, но не понятием предметной области. Одно понятие может быть представлено несколькими атрибутами в разных таблицах, при этом в реляционной схеме отсутствует информация о взаимосвязи между атрибутами и понятиями. Поэтому реляционная схема служит для хранения данных, но не концептуального описания предметной области.*

SCM-схема построена исключительно на понятиях предметной области.

### 3.2.2 Свойство семантической полноты

Свойство семантической полноты SCM выражено в ограничении ассоциации (см. 0) "в рамках схемы не может существовать двух ассоциаций, связывающих множества понятий, одно из которых является подмножеством другого". Оно означает, что каждая ассоциация несет в себе *полную* информацию о *семантике* взаимосвязи понятий, то есть схема не может содержать другой ассоциации, которая бы описывала взаимосвязь тех же понятий альтернативным образом.

*Пример. Если в схеме существует ассоциация (Человек, Вид навыка, Уровень навыка), то в той же схеме не может существовать, например, ассоциация (Человек, Вид навыка). Поэтому ассоциация (Человек, Вид навыка, Уровень навыка) полностью описывает семантику взаимосвязи понятий Человек, Вид навыка, Уровень навыка.*

### 3.2.3 Отсутствие собственных имен ассоциаций

Ассоциация SCM построена на уникальном множестве понятий, следовательно множество понятий является ее идентификатором. Поэтому нет необходимости в назначении ассоциациям собственных имен, что позволяет избавить проектировщика и пользователя схемы от необходимости помнить их точное написание.

*Пример. Ассоциация (Задача, Человек) могла быть названа как "назначение", "решение" и др. Но в SCM-схеме данная ассоциация не будет иметь собственного имени, так как на нее можно однозначно сослаться перечислив понятия.*

### 3.2.4 Отсутствие деталей реализации

SCM-схема представляет собой исключительно концептуальное описание структуры предметной области и не содержит деталей реализации. Поэтому SCM позиционируется как вычислительно-независимая модель (Computational Independent Model) OMG MDA. Реализация SCM-схемы в виде программной системы требует ее отображения в модели других типов (реляционная модель, UML и др.), содержащих большее количество деталей реализации.

*Пример. В ходе создания реляционной схемы на основе SCM-схемы, ассоциация (Человек, Телефон) типа многие-к-одному может быть отображена в виде полей "Телефон" и "Человек" в таблице "Человек" с первичным ключом по полю "Человек"; а ассоциация (Человек, Задача) типа многие-ко-многим может быть отображена в виде отдельной таблицы с полями "Человек", "Задача" и первичным ключом по обоим полям.*

## 4 Платформо-независимая спецификация SCM

Платформо-независимая спецификация SCM определяет набор интерфейсов, используемых для взаимодействия с SCM-схемами, представленными в виде объектов в рамках вычислительных систем. Для платформо-независимой спецификации SCM далее будет применяться UML.

Все перечисляемые ниже операции могут генерировать исключения ForbiddenAction (действие запрещено), UnderlyingSystemException (необработываемое исключение, возникшее при работе системы, используемой конкретной реализацией SCM), InternalInconsistency (возникла внутренняя противоречивость состояния системы, требуется обращение к разработчикам), а также исключения, подразумеваемые спецификацией OMG Transaction Service Specification V1.4 (<http://www.omg.org/cgi-bin/doc?formal/2003-09-02>), 2003. Поэтому данные исключения не перечисляются в явном виде при описании операций, но подразумеваются.

### 4.1 Объектная спецификация SCM-схемы

#### 4.1.1 Предметная область

Каждая предметная область реализует интерфейс Domain, который обеспечивает получение/модификацию содержимого предметных областей, навигацию между ними. Интерфейс Domain предоставляет перечисленные далее операции.

<<CORBAInterface>> Domain	
○-	
+ getRootDomain ()	: Domain
+ getUnitedSchema ()	: Schema
+ getDesignation ()	: string
+ getFullDesignation ()	: string
+ getMainDomain ()	: Domain
+ getNestedDomains ()	: Iterator
+ getNestedDomain (string designation)	: Domain
+ createNestedDomain (string designation)	: Domain
+ removeNestedDomain (Domain domain)	: void
+ existsConcept (string designation)	: boolean
+ getConcept (string designation)	: Concept
+ getDescribedConcept (string designation)	: Concept
+ getDescribedConcepts ()	: Iterator
+ createDescribedConcept (string designation, Type type)	: Concept
+ removeDescribedConcept (Concept concept)	: void
+ getLocalSchema (string designation)	: Schema
+ getLocalSchemas ()	: Iterator
+ createLocalSchema (string designation)	: Schema
+ removeLocalSchema (Schema schema)	: void
+ getBaseType (short baseType)	: BaseType
+ getType (string designation)	: Type
+ createDictionary (string designation)	: Dictionary
+ createNumberRange (string designation, double min, double max, short rangeType)	: NumberRange
+ createIntegerRange (string designation, long min, long max)	: IntegerRange
+ createDateRange (string designation, long min, long max, short rangeType)	: DateRange
+ createCodeRange (string designation, long min, long max)	: IntegerRange
+ createLimitedLengthString (string designation, long min, long max)	: IntegerRange
+ createTemplatedString (string designation, GeneralIterator templates)	: TemplatedString
+ createNumberCollection (string designation, GeneralIterator values)	: SimpleValueCollection
+ createIntegerCollection (string designation, GeneralIterator values)	: SimpleValueCollection
+ createDateCollection (string designation, GeneralIterator values)	: SimpleValueCollection
+ createStringCollection (string designation, GeneralIterator values)	: SimpleValueCollection
+ createCodeCollection (string designation, GeneralIterator values)	: SimpleValueCollection
+ createNumberRangeCollection (string designation, Iterator ranges)	: RangeCollection
+ createIntegerRangeCollection (string designation, Iterator ranges)	: RangeCollection
+ createDateRangeCollection (string designation, Iterator ranges)	: RangeCollection
+ createCodeRangeCollection (string designation, Iterator ranges)	: RangeCollection
+ createStructure (string designation, Iterator concepts, CompoundType baseStructure)	: CompoundType
+ createVector (string designation, Concept concept)	: CompoundType
+ createSet (string designation, Concept concept)	: CompoundType
+ createLimitedVector (string designation, Concept concept, unsigned long min, unsigned long max)	: LimitedCompoundType
+ createLimitedSet (string designation, Concept concept, unsigned long min, unsigned long max)	: LimitedCompoundType
+ createStructureValueCollection (string designation, CompoundType sourceType, Iterator values)	: CompoundValueCollection
+ createVectorValueCollection (string designation, CompoundType sourceType, Iterator values)	: CompoundValueCollection
+ createSetValueCollection (string designation, CompoundType sourceType, Iterator values)	: CompoundValueCollection
+ removeType (Type type)	: void

Domain `getRootDomain()` – получить корневой домен, в который данный домен (this) вложен непосредственно или опосредованно. В принципе, в разных вычислительных пространствах могут существовать несколько различных деревьев доменов, но у каждого из этих деревьев должна быть своя единая схема. Предполагается, что существует одно общепринятое/согласованное дерево доменов и его единая схема (образующие вместе пространство понятий), а все остальные альтернативные деревья доменов и их единые схемы являются экспериментальными для тех или иных целей. Пространство понятий представляет собой пространство интеграции различных систем, публикующих свои данные в виде SCM. В рамках пространства понятий выполняются все свойства SCM, перечисленные в данной спецификации.

Schema `getUnitedSchema()` – получить единую схему, соответствующую тому дереву доменов, к которому относится данный домен.

`string getDesignation()` – получить обозначение предметной области на текущем языке (*текущий язык определяется в спецификации MLANG-SPEC, далее этот термин многократно используется*).

`string getFullDesignation()` – получить полное обозначение предметной области.

`Domain getMainDomain()` – получить главную предметную область по отношению к данной предметной области; возвращает **nil (null)**, если предметная область является корневой;

`Iterator getNestedDomains()` – получить итератор вложенных предметных областей (объекты, возвращаемые итератором, реализуют интерфейс `Domain`).

`Domain getNestedDomain(string designation)` – получить вложенную предметную область, имеющую обозначение, равное <обозначение данной предметной области> + '!' + <параметр designation>. *Здесь и далее будем называть такое обозначение составным.* Поэтому возвращаемая предметная область может быть вложена как непосредственно, так и опосредованно через другие предметные области. Исключения:

- `NoDomain` – искомая вложенная предметная область не существует.

`Domain createNestedDomain(string designation)` – создать новую предметную область, вложенную в данную предметную область, с присвоением указанного обозначения. Обозначение может быть составным, аналогично `Domain.getNestedDomain`. Исключения:

- `DuplicateDomain` – в данной предметной области уже существует вложенная предметная область с тем же обозначением.

`void removeNestedDomain(Domain domain)` – удаляет указанную предметную область и все ее вложенные предметные области. Может вызываться для любой предметной области дерева предметных областей. Исключения:

- `NoDomain` – указанная в качестве параметра предметная область не принадлежит данному дереву предметных областей (это означает, что в качестве удаляемого задан домен из другого дерева предметных областей);
- `ConceptInUse` – хотя бы одно понятие удаляемой предметной области и вложенных в нее предметных областей используется в любой из схем, существующих в данном дереве предметных областей;
- `NotSupportedDomainCascadeDeletion` – каскадное удаление предметной области и всех вложенных объектов не поддерживается.

`boolean existsConcept(string designation)` – проверить существование понятия (как описанного, так и порожденного) с указанным обозначением (обозначение может быть составным, указывая на вложенную предметную область, начиная от данной предметной области). В частности, данная операция должна эффективно реализовывать проверку существования порожденных понятий, без создания соответствующих объектов.

`Concept getConcept(string designation)` – получить описанное или порожденное понятие по его обозначению (обозначение может быть составным, указывая на понятие во вложенной предметной области). Исключения:

- `NoConcept` – искомое понятие не существует.

`Concept getDescribedConcept(string designation)` – получить описанное понятие по его обозначению (обозначение может быть составным, указывая на понятие во вложенной предметной области). Исключения:

- `NoDescribedConcept` – искомое описанное понятие не существует (при этом может существовать порожденное понятие с тем же обозначением).

`Iterator getDescribedConcepts()` – получить итератор по всем описанным понятиям данной предметной области (объекты, возвращаемые итератором, реализуют интерфейс `Concept`). Получить итератор по порожденным понятиям невозможно в принципе, так как количество порожденных понятий может быть бесконечным.

`Concept createDescribedConcept(string designation, Type type)` – создать новое описанное понятие с заданным обозначением (обозначение может быть составным, указывая на необходимость создания понятия во вложенной предметной области), принадлежащее заданному типу. Данная операция также используется для описания уже существующих порожденных понятий, при этом такое понятие становится и описанным, и порожденным одновременно. Исключения:

- DuplicateDescribedConcept – в данной предметной области уже существует описанное понятие с тем же обозначением.

void removeDescribedConcept(Concept concept) – удаление указанного описанного понятия. Данная операция также используется для удаления понятий, которые одновременно являются и описанным, и порожденными. В этом случае понятие перестает быть описанным, с удалением всей дополнительной информации по нему, но сохраняется как порожденное. При попытке удаления понятия, которое является только порожденным, генерируется исключение, так как порожденные понятия могут быть удалены только через удаление тех понятий, из которых они порождены. Исключения:

- NoDescribedConcept – удаляемое понятие не существует в качестве описанного (но может существовать в качестве порожденного) в данном дереве предметных областей;
- ConceptInUse – удаляемое понятие используется в одной из схем, существующих в данном дереве предметных областей.

Schema getLocalSchema(string designation) – получить локальную схему по ее обозначению из данной предметной области (обозначение может быть составным, указывая на необходимость поиска схемы в соответствующей вложенной предметной области). Исключения:

- NoLocalSchema – искомая локальная схема не существует.

Iterator getLocalSchemas() – получить итератор локальных схем данной предметной области (объекты, возвращаемые итератором, реализуют интерфейс Schema).

Schema createLocalSchema(string designation) – создать локальную схему в данной предметной области с указанным обозначением (обозначение может быть составным, указывая на необходимость создания локальной схемы во вложенной предметной области). Исключения:

- DuplicateLocalSchema – в данной предметной области уже существует локальная схема с тем же обозначением.

void removeLocalSchema(Schema schema) – удалить заданную локальную схему.

Исключения:

- NoLocalSchema – указанной локальной схемы не существует в данном дереве предметных областей (возможно, данная схема принадлежит другому дереву предметных областей или является глобальной);
- NotSupportedLocalSchemaCascadeDeletion – каскадное удаление локальной схемы со всеми вложенными объектами не поддерживается.

BaseType getBaseType(short baseType) – получить объект базового типа по указанной константе (см. константы BaseTypeConstraints). Все объекты базовых типов должны быть разделяемыми (общими) в рамках всего дерева предметных областей, то есть каждая предметная область возвращает один и тот же объект для каждого базового типа. Исключения:

- NoBaseTypeConstant – не существует базового типа для указанной константы типа.

Type getType(string designation) – получить тип по его обозначению (обозначение может быть составным, указывая на тип во вложенной предметной области). Исключения:

- NoType – указанного типа не существует в данном дереве предметных областей.

Dictionary createDictionary(string designation) – создать тип справочника с заданным обозначением (обозначение может быть составным). Исключения:

- DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей.

NumberRange createNumberRange(string designation, double min, double max, short rangeType) – создать тип числового диапазона с заданным обозначением, минимальной и максимальной границами, типом границ. Исключения:

- DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
- BadRange – минимальная граница диапазона больше его максимальной границы;

- NoRangeTypeConstant – не существует указанной константы типа границ.  
IntegerRange createIntegerRange(string designation, long min, long max) – создать тип целочисленного диапазона с заданным обозначением, минимальной и максимальной границами. Исключения:
  - DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
  - BadRange – минимальная граница диапазона больше его максимальной границы.  
DateRange createDateRange(string designation, long min, long max, short rangeType) – создать тип диапазона дат с заданным обозначением, минимальной и максимальной границами, типом границ. Исключения:
    - DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
    - BadRange – минимальная граница диапазона больше его максимальной границы;
    - NoRangeTypeConstant – не существует указанной константы типа границ.  
IntegerRange createCodeRange(string designation, long min, long max) – создать тип диапазона кодов с заданным обозначением, минимальной и максимальной границами. Исключения:
      - DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
      - BadRange – минимальная граница диапазона больше его максимальной границы.  
IntegerRange createLimitedLengthString(string designation, long min, long max) – создать тип строки ограниченной длины с заданным обозначением, минимальной и максимальной длинами. Исключения:
        - DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
        - BadLimits – минимальный предел больше максимального предела длины строки.  
TemplatedString createTemplatedString(string designation, GeneralIterator templates) – создать тип строки по шаблону с заданным обозначением и с заданной совокупностью строковых шаблонов типа string. Итератор должен содержать хотя бы один элемент и все его элементы должны быть типа string. Структура шаблона должна соответствовать описанию, данному выше. Исключения:
          - DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
          - BadElementType – неверный тип элемента итератора (ожидается string);
          - EmptyIterator – пустой итератор (в итераторе должен быть хотя бы один элемент).  
SimpleValueCollection createNumberCollection(string designation, GeneralIterator values) – создать тип "набор чисел" с заданным обозначением на основе заданной совокупности чисел. Итератор должен содержать хотя бы один элемент и все его элементы должны быть числового типа. Исключения:
            - DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
            - BadElementType – неверный тип элемента итератора (ожидается число);
            - EmptyIterator – пустой итератор (в итераторе должен быть хотя бы один элемент).  
SimpleValueCollection createIntegerCollection(string designation, GeneralIterator values) – создать тип "набор целых чисел" с заданным обозначением на основе заданной совокупности целых чисел. Итератор должен содержать хотя бы один элемент и все его элементы должны быть целочисленными. Исключения:
              - DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
              - BadElementType – неверный тип элемента итератора (ожидается целое число);
              - EmptyIterator – пустой итератор (в итераторе должен быть хотя бы один элемент).

`SimpleValueCollection createDateCollection(string designation, GeneralIterator values)`  
– создать тип "набор дат" с заданным обозначением на основе заданной совокупности дат. Итератор должен содержать хотя бы один элемент и все его элементы должны быть типа времени. Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `BadElementType` – неверный тип элемента итератора (ожидается дата);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`SimpleValueCollection createStringCollection(string designation, GeneralIterator values)`  
– создать тип "набор строк" с заданным обозначением на основе заданной совокупности строк. Итератор должен содержать хотя бы один элемент и все его элементы должны быть строками. Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `BadElementType` – неверный тип элемента итератора (ожидается строка);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`SimpleValueCollection createCodeCollection(string designation, GeneralIterator values)`  
– создать тип "набор кодов" с заданным обозначением на основе заданной совокупности кодов. Итератор должен содержать хотя бы один элемент и все его элементы должны быть целыми числами (так как коды представляются в виде целых чисел). Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `BadElementType` – неверный тип элемента итератора (ожидается целое число);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`RangeCollection createNumberRangeCollection(string designation, Iterator ranges)` – создать тип "набор диапазонов чисел" с заданным обозначением на основе заданной совокупности непрерывных диапазонов. Итератор должен содержать хотя бы один элемент и все его элементы должны реализовывать интерфейс `ContinuousRange`. Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `BadRange` – минимальная граница одного из диапазонов больше его максимальной границы;
- `NoRangeTypeConstant` – не существует указанной константы типа границ;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `ContinuousRange`);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`RangeCollection createIntegerRangeCollection(string designation, Iterator ranges)` – создать тип "набор диапазонов целых чисел" с заданным обозначением на основе заданной совокупности дискретных диапазонов. Итератор должен содержать хотя бы один элемент и все его элементы должны реализовывать интерфейс `DiscreteRange`. Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `BadRange` – минимальная граница одного из диапазонов больше его максимальной границы;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `DiscreteRange`);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`RangeCollection createDataRangeCollection(string designation, Iterator ranges)` – создать тип "набор диапазонов дат" с заданным обозначением на основе заданной

совокупности диапазонов дат. Итератор должен содержать хотя бы один элемент и все его элементы должны реализовывать интерфейс `TimeRange`. Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `BadRange` – минимальная граница одного из диапазонов больше его максимальной границы;
- `NoRangeTypeConstant` – не существует указанной константы типа границ;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `TimeRange`);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`RangeCollection createCodeRangeCollection(string designation, Iterator ranges)` – создать тип "набор диапазонов кодов" с заданным обозначением на основе заданной совокупности дискретных диапазонов. Итератор должен содержать хотя бы один элемент и все его элементы должны реализовывать интерфейс `DiscreteRange` (так как коды представляются в виде целых чисел). Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `BadRange` – минимальная граница одного из диапазонов больше его максимальной границы;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `DiscreteRange`);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`CompoundType createStructure(string designation, Iterator concepts, CompoundType baseStructure)` – создать тип структуры с заданным обозначением на основе заданной последовательности вложенных понятий как производный от указанной базовой структуры. Если базовая структура равна `nil` (`null`), то данный тип структуры не является производным. Итератор должен содержать хотя бы один элемент и все его элементы должны реализовывать интерфейс `Concept` (см. выше описание причин того, что в качестве вложенных задаются понятия, а не собственно типы). Все элементы итератора должны быть различными. Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `NoConcept` – одно из понятий не существует в данном дереве предметных областей;
- `NoStructure` – указанный составной тип не является структурой или типом данного дерева предметных областей;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `Concept`);
- `RepeatedElements` – итератор содержит повторяющиеся понятия;
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`CompoundType createVector(string designation, Concept concept)` – создать тип вектора с заданным обозначением на основе заданного понятия. Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `NoConcept` – заданное понятие не существует в данном дереве предметных областей.

`CompoundType createSet(string designation, Concept concept)` – создать тип множества с заданным обозначением на основе заданного понятия. Исключения:

- `DuplicateType` – тип с указанным обозначением уже существует в данном дереве предметных областей;
- `NoConcept` – заданное понятие не существует в данном дереве предметных областей.

`LimitedCompoundType createLimitedVector(string designation, Concept concept, unsigned long min, unsigned long max)` – создать тип ограниченного вектора с заданным

обозначением на основе заданных понятия, минимальной и максимальной размерностей вектора. Исключения:

- DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
- BadLimits – минимальный предел больше максимального предела;
- NoConcept – заданное понятие не существует в данном дереве предметных областей.

LimitedCompoundType createLimitedSet(string designation, Concept concept, unsigned long min, unsigned long max) – создать тип ограниченного множества с заданным обозначением на основе заданных понятия, минимальной и максимальной размерностей вектора. Исключения:

- DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
- BadLimits – минимальный предел больше максимального предела;
- NoConcept – заданное понятие не существует в данном дереве предметных областей.

CompoundValueCollection createStructureValueCollection(string designation, CompoundType sourceType, Iterator values) – создать тип "набор значений структуры" с заданным обозначением на основе заданных исходного типа структуры и совокупности составных значений этой структуры. Составные значения структуры могут представлять собой как объекты с интерфейсом CompoundValue, так и объекты любого типа (интерфейса), в который отображается данная структура (допускается в одном итераторе сочетать и те, и другие виды объектов). Итератор должен содержать хотя бы один элемент и все элементы итератора должны относиться/отображаться к тому типу структуры, который указан в качестве параметра. **Вопросы отображения структур в типы объектно-ориентированного языка рассматриваются в отдельных спецификациях, так как являются частью внутренней реализации SCM (это касается всего последующего текста данной спецификации).** Исключения:

- DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
- NoType – указанного типа структуры не существует в данном дереве предметных областей;
- BadElementType – неверный тип элемента итератора (ожидается объект, реализующий интерфейс CompoundValue и относящийся к указанному типу структуры, см. операцию CompoundValue.getType; или объект, тип/интерфейс которого отражается в указанный тип структуры);
- EmptyIterator – пустой итератор (в итераторе должен быть хотя бы один элемент).

CompoundValueCollection createVectorValueCollection(string designation, CompoundType sourceType, Iterator values) – создать тип "набор значений вектора" с заданным обозначением на основе заданных исходного типа вектора и совокупности составных значений этого вектора. Составные значения вектора могут представлять собой как объекты с интерфейсом CompoundValue, так и объекты любого типа (интерфейса), в который отображается данный вектор (допускается в одном итераторе сочетать и те, и другие виды объектов). Итератор должен содержать хотя бы один элемент и все элементы итератора должны относиться/отображаться к тому типу вектора, который указан в качестве параметра. Исключения:

- DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
- NoType – указанного типа вектора не существует в данном дереве предметных областей;
- BadElementType – неверный тип элемента итератора (ожидается объект, реализующий интерфейс CompoundValue и относящийся к указанному типу вектора, см. операцию

CompoundValue.getType; или объект, тип/интерфейс которого отражается в указанный тип вектора);

- EmptyIterator – пустой итератор (в итераторе должен быть хотя бы один элемент).  
CompoundValueCollection createSetValueCollection(string designation, CompoundType sourceType, Iterator values) – создать тип "набор значений множества" с заданным обозначением на основе заданных исходного типа множества и совокупности составных значений этого множества. Составные значения множества могут представлять собой как объекты с интерфейсом CompoundValue, так и объекты любого типа (интерфейса), в который отображается данное множество (допускается в одном итераторе сочетать и те, и другие виды объектов). Итератор должен содержать хотя бы один элемент и все элементы итератора должны относиться/отображаться к тому типу множества, который указан в качестве параметра. Исключения:
  - DuplicateType – тип с указанным обозначением уже существует в данном дереве предметных областей;
  - NoType – указанного типа множества не существует в данном дереве предметных областей;
  - BadElementType – неверный тип элемента итератора (ожидается объект, реализующий интерфейс CompoundValue и относящийся к указанному типу множества, см. операцию CompoundValue.getType; или объект, тип/интерфейс которого отражается в указанный тип множества);
- EmptyIterator – пустой итератор (в итераторе должен быть хотя бы один элемент).  
void removeType(Type type) – удалить указанный тип из данного дерева предметных областей. Запрещено удаление типа, если на него ссылается хотя бы одно понятие или составной тип в данном дереве предметных областей. Исключения:
  - NoType – указанного типа не существует в данном дереве предметных областей;
  - TypeInUse – удаляемый тип используется понятием или составным типом в рамках данного дерева предметных областей и поэтому не может быть удален.

#### 4.1.2 Схема

Каждая схема реализует интерфейс Schema, который обеспечивает получение/модификацию содержимого схемы. Интерфейс Schema предоставляет перечисленные далее операции.

<<CORBAInterface>>	
0-	Schema
+ getDesignation ()	: string
+ setDesignation (string designation)	: void
+ getDomain ()	: Domain
+ getFullDesignation ()	: string
+ existsSchemaAssociation (Iterator concepts)	: boolean
+ getSchemaAssociation (Iterator concepts)	: SchemaAssociation
+ getDescribedSchemaAssociation (Iterator concepts)	: SchemaAssociation
+ getDescribedSchemaAssociations ()	: Iterator
+ getDescribedSchemaAssociations (Concept concept)	: Iterator
+ getDescribedSchemaAssociations (Model.SCM.CL.Constraint constraint)	: Iterator
+ getSchemaAssociations (Model.SCM.CL.Constraint constraint)	: Iterator
+ createDescribedSchemaAssociation (Iterator concepts)	: SchemaAssociation
+ removeDescribedSchemaAssociation (SchemaAssociation association)	: void
+ fulfillsConstraint (Model.SCM.CL.ConstraintBody constraintBody)	: boolean
+ getDescribedConstraint (string designation)	: Model.SCM.CL.Constraint
+ getDescribedConstraint (Model.SCM.CL.ConstraintBody constraintBody)	: Model.SCM.CL.Constraint
+ getDescribedConstraints ()	: Iterator
+ getDescribedConstraints (SchemaAssociation association)	: Iterator
+ getDescribedConstraints (Concept concept)	: Iterator
+ createDescribedConstraint (Model.SCM.CL.ConstraintBody constraintBody)	: Model.SCM.CL.Constraint
+ removeDescribedConstraint (Model.SCM.CL.Constraint constraint)	: void

`string getDesignation()` – получить обозначение схемы на текущем языке (см. пояснения по текущему языку выше).

`void setDesignation(string designation)` – изменить обозначение схемы для текущего языка.

- `DuplicateLocalSchema` – в предметной области уже существует локальная схема с тем же обозначением на текущем языке (для единой схемы обозначения могут изменяться произвольным образом).

`Domain getDomain()` – получить домен, в котором располагается данная схема. Возвращает корневой домен, если схема является единой.

`string getFullDesignation()` – получить полное обозначение схемы на текущем языке.

`boolean existsSchemaAssociation(Iterator concepts)` – проверить существование в данной схеме ассоциации, построенной на указанной совокупности понятий (реализация данной проверки может быть эффективнее для неописанных порожденных ассоциаций, чем реализация получения ассоциации, так как при этом нет необходимости создавать объект для ассоциации). Множество понятий ассоциации должно в точности совпадать с указанной совокупностью понятий. Итератор должен содержать хотя бы два элемента; все его элементы должны реализовывать интерфейс `Concept`; элементы итератора должны отдаваться в алфавитном порядке полных обозначений понятий и не должны повторяться. Исключения:

- `NoConcept` – одно из понятий не существует в рамках данного дерева предметных областей;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `Concept`);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент);
- `FewElements` – количество понятий в итераторе меньше двух;
- `RepeatedElements` – итератор содержит повторяющиеся понятия;
- `UnorderedIterator` – неупорядоченный итератор (ожидается упорядоченность понятий по полным обозначениям).

`SchemaAssociation getSchemaAssociation(Iterator concepts)` – получить ассоциацию данной схемы, построенную в точности на перечисленных понятиях. Данная операция может вызываться как для получения описанных ассоциаций, так и для получения неописанных порожденных ассоциаций. Итератор должен содержать хотя бы два элемента; все его элементы должны реализовывать интерфейс `Concept`; элементы итератора должны отдаваться в алфавитном порядке полных обозначений понятий и не должны повторяться. Исключения:

- `NoAssociation` – искомой ассоциации не существует в рамках данной схемы;
- `NoConcept` – одно из понятий не существует в рамках данного дерева предметных областей;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `Concept`);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент);
- `FewElements` – количество понятий в итераторе меньше двух;
- `RepeatedElements` – итератор содержит повторяющиеся понятия;
- `UnorderedIterator` – неупорядоченный итератор (ожидается упорядоченность понятий по полным обозначениям).

`SchemaAssociation getDescribedSchemaAssociation(Iterator concepts)` – получить описанную ассоциацию данной схемы, построенную в точности на перечисленных понятиях. Данная операция может вызываться только для получения описанных ассоциаций, вызов для неописанных порожденных ассоциаций приведет к генерации исключения `NoAssociation`. Итератор должен содержать хотя бы два элемента; все его элементы должны реализовывать интерфейс `Concept`; элементы итератора должны

отдаваться в алфавитном порядке полных обозначений понятий и не должны повторяться. Исключения:

- `NoDescribedAssociation` – искомой описанной ассоциации не существует в рамках данной схемы (при этом может существовать неописанная порожденная ассоциация);
- `NoConcept` – одно из понятий не существует в рамках данного дерева предметных областей;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `Concept`);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент);
- `FewElements` – количество понятий в итераторе меньше двух;
- `RepeatedElements` – итератор содержит повторяющиеся понятия;
- `UnorderedIterator` – неупорядоченный итератор (ожидается упорядоченность понятий по полным обозначениям).

`Iterator getDescribedSchemaAssociations()` – получить все описанные ассоциации данной схемы (получить все возможные неописанные порожденные ассоциации невозможно, так как их количество может быть бесконечным, поэтому соответствующей операции не существует). Результирующий итератор перечисляет все ассоциации в алфавитном порядке их синтезированных обозначений (см. `SchemaAssociation.synthesizeDesignation`).

`Iterator getDescribedSchemaAssociations(Concept concept)` – получить все описанные ассоциации данной схемы, включающие указанное понятие (получить все возможные неописанные порожденные ассоциации по понятию невозможно, так как их количество может быть бесконечным, поэтому соответствующей операции не существует). Результирующий итератор перечисляет все ассоциации в алфавитном порядке их синтезированных обозначений (см. `SchemaAssociation.synthesizeDesignation`).

`Iterator getDescribedSchemaAssociations(::Model::SCM::Constraint constraint)` – получить все описанные ассоциации данной схемы, используемые в заданном ограничении. Результирующий итератор перечисляет все ассоциации в алфавитном порядке их синтезированных обозначений (см. `SchemaAssociation.synthesizeDesignation`).

`Iterator getSchemaAssociations(::Model::SCM::Constraint constraint)` – получить все (и описанные, и неописанные порожденные) ассоциации данной схемы, используемые в заданном ограничении (это возможно, так как в рамках ограничения может использоваться лишь ограниченное количество неописанных порожденных ассоциаций). Результирующий итератор перечисляет все ассоциации в алфавитном порядке их синтезированных обозначений (см. `SchemaAssociation.synthesizeDesignation`).

`SchemaAssociation createDescribedSchemaAssociation(Iterator concepts)` – создать описанную ассоциацию в данной схеме по указанному множеству понятий. Описанная ассоциация может быть создана в том числе для порожденных ассоциаций, в этом случае такая ассоциация становится и описанной, и порожденной одновременно. На момент создания ассоциации не должно существовать другой ассоциации, построенной на множестве понятий, полностью включающем указанное множество понятий или полностью включенном в него. Итератор должен содержать хотя бы два элемента; все его элементы должны реализовывать интерфейс `Concept`; элементы итератора должны отдаваться в алфавитном порядке полных обозначений понятий и не должны повторяться. Исключения:

- `DuplicateAssociation` – ассоциация с указанным обозначением уже существует в данной схеме;
- `NoConcept` – одно из понятий не существует в рамках данного дерева предметных областей;
- `IncludedAssociation` – множество понятий создаваемой ассоциации входит как часть или включает в себя как часть множество понятий уже существующей ассоциации в рамках данной схемы;

- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `Concept`);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент);
- `FewElements` – количество понятий в итераторе меньше двух;
- `RepeatedElements` – итератор содержит повторяющиеся понятия;
- `UnorderedIterator` – неупорядоченный итератор (ожидается упорядоченность понятий по полным обозначениям).

`void removeDescribedSchemaAssociation(SchemaAssociation association)` – удалить описанную ассоциацию из данной схемы по указанному множеству понятий. Если описанная ассоциация одновременно является и порожденной, то после выполнения данной операции она останется порожденной, но перестанет быть описанной. Итератор должен содержать хотя бы два элемента; все его элементы должны реализовывать интерфейс `Concept`; элементы итератора должны отдаваться в алфавитном порядке полных обозначений понятий. Исключения:

- `NoDescribedAssociation` – ассоциация с указанным обозначением не принадлежит данной схеме или не является описанной;
- `AssociationInUse` – удаляемая ассоциация используется (например, в одном из ограничений данной схемы или для доступа к содержимому) и поэтому не может быть удалена.

`boolean fulfillsConstraint(::Model::SCM::ConstraintBody constraintBody)` – проверить, выполняется ли в данной схеме ограничение заданной структуры. При этом, для выполнения ограничения совершенно не обязательно, чтобы в схеме существовало ограничение с совпадающей структурой. Ограничение считается выполняющимся, если в схеме есть одно или несколько ограничений, эквивалентно преобразуемых к заданной структуре ограничения. Алгоритм проверки выполнимости ограничения достаточно сложен, является предметом отдельной спецификации и может не поддерживаться в начальных реализациях SCM. Исключения:

- `NotSupportedConstraintEquiChecking` – проверка ограничений на эквивалентность не поддерживается.

`::Model::SCM::Constraint getDescribedConstraint(string designation)` – получить описанное ограничение по указанному обозначению (обозначение не может быть составным, так как обозначения ограничений присваиваются в рамках схемы, см. описание ограничения выше). Не всякое ограничение можно получить по обозначению, так как не всякое ограничение его имеет (см. описание ограничения). Для доступа к ограничениям без обозначений существуют операции, описанные далее. Исключения:

- `NoDescribedConstraint` – искомое описанное ограничение не существует в рамках данной схемы.

`::Model::SCM::Constraint getDescribedConstraint(::Model::SCM::ConstraintBody constraintBody)` – получить описанное ограничение данной схемы по его структуре. Эта операция используется для доступа к ограничениям, не имеющим обозначения. Исключения:

- `NoDescribedConstraint` – искомое описанное ограничение не существует в рамках данной схемы.

`Iterator getDescribedConstraints()` – получить все описанные ограничения данной схемы. Неописанные порожденные ограничения получить невозможно, так как их количество бесконечно, поэтому для таких ограничений не существует соответствующей операции (но допускается проверка существования неописанных порожденных ограничений при помощи операции `Schema.fulfillsConstraint`, см. выше).

`Iterator getDescribedConstraints(SchemaAssociation association)` – получить те описанные ограничения данной схемы, которые используют указанную ассоциацию (ассоциация может быть как описанной, так и неописанной). Исключения:

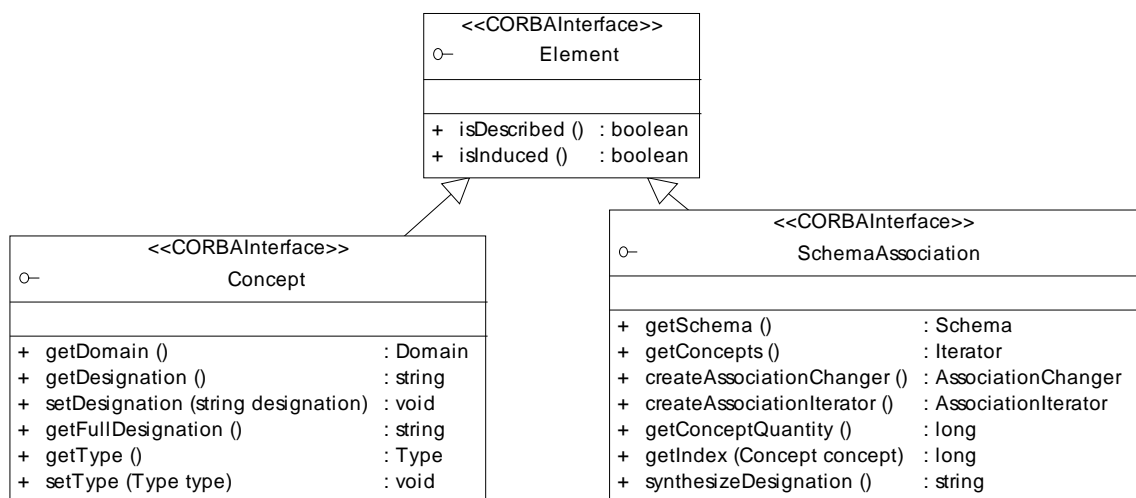
- `NoAssociation` – искомой ассоциации не существует в рамках данной схемы.

Iterator `getDescribedConstraints(Concept concept)` – получить те описанные ограничения данной схемы, которые используют указанное понятие (понятие может быть как описанным, так и неописанным). Исключения:

- `NoConcept` – указанное понятие не существует в данном дереве предметных областей.  
`::Model::SCM::Constraint createDescribedConstraint(::Model::SCM::ConstraintBody constraintBody)` – создать описанное ограничение по указанной структуре ограничения. Если данное ограничение выполнялось в рамках данной схемы до его создания в явном виде (до вызова этой операции), то оно становится описанным и порожденным одновременно.
- `DuplicateConstraintBody` – ограничение с указанной структурой уже существует в данной схеме.  
`void removeDescribedConstraint(::Model::SCM::Constraint constraint)` – удалить указанное описанное ограничение. Если оно выводится из других ограничений схемы, то оно сохраняет свою выполнимость и после его удаления (но при этом становится недоступным как описанное ограничение).
- `NoDescribedConstraint` – искомое описанное ограничение не существует в рамках данной схемы.

### 4.1.3 Элемент схемы

Каждый элемент схемы реализует интерфейс `Element`, который служит базовым интерфейсом для всех видов элементов схем. Данный интерфейс имеет три производных интерфейса: `Concept`, `SchemaAssociation`, `Constraint`. В данной спецификации описываются только первые два интерфейса. Последний интерфейс является предметом отдельной спецификации по языку ограничений `SCM-CL-SPEC`.



Интерфейс `Element` предоставляет следующие операции.

`boolean isDescribed()` – проверить, что данный элемент является описанным.

`boolean isInduced()` – проверить, что данный элемент является порожденным.

Интерфейс `Concept` является расширением интерфейса `Element` и дополнительно предоставляет следующие операции.

`Domain getDomain()` – получить домен, в котором располагается данное понятие.

`string getDesignation()` – получить обозначение понятия.

`void setDesignation(string designation)` – задать новое обозначение для данного понятия на текущем языке. Изменение обозначений допускается только для не порожденных понятий, так как обозначение порожденных понятий однозначно

вычисляется из обозначений исходных понятий и собственных свойств порожденного понятия. Исключения:

- **InducedConcept** – данное понятие является порожденным, а поэтому изменение собственного обозначения не допускается.  
string getFullDesignation() – получить полное обозначение понятия.  
Type getType() – получить тип, в виде которого представляется понятие.  
void setType(Type type) – задать тип, в виде которого будет представляться данное понятие. В начальных реализациях SCM эта функциональность может не поддерживаться. Исключения:
- **NoType** – указанного типа не существует в рамках данного дерева предметных областей;
- **NotSupportedTypeChanging** – изменение типа не поддерживается в данной реализации SCM.

Интерфейс **SchemaAssociation** является расширением интерфейса **Element**. **SchemaAssociation** представляет не столько саму ассоциацию, сколько ее участие (вариант) в конкретной схеме. Поэтому **SchemaAssociation** относится строго к одной схеме – к той, в которой участвует данный вариант ассоциации. Одна ассоциация может быть представлена множеством вариантов, каждый из которых участвует в своей схеме и соответствует своему объекту, реализующему интерфейс **SchemaAssociation**. Но при этом все такие объекты считаются эквивалентными с точки зрения той ассоциации, которую они представляют. **SchemaAssociation** дополнительно к интерфейсу **Element** предоставляет операции, перечисленные далее.

**Schema getSchema()** – получить схему, в которой существует данный вариант ассоциации.

**Iterator getConcepts()** – получить итератор понятий, на которых построена данная ассоциация (объекты, возвращаемые итератором, реализуют интерфейс **Concept**). Итератор возвращает понятия в алфавитном порядке их полных обозначений.

**AssociationChanger createAssociationChanger()** – создать новый менеджер изменений для данного варианта ассоциации, через который можно управлять содержимым данной ассоциации в рамках данной схемы. Функциональность изменения содержимого ассоциаций может не поддерживаться в начальных реализациях SCM. Исключения:

- **NotSupportedContentChanging** – изменение содержимого ассоциаций не поддерживается в данной реализации SCM.

**AssociationIterator createAssociationIterator()** – получить новый итератор содержимого данного варианта ассоциации.

**long getConceptQuantity()** – получить общее количество понятий в данной ассоциации.

**long getIndex(Concept concept)** – получить индекс указанного понятия в рамках данной ассоциации. Индексы присваиваются понятиям от 0 до **SchemaAssociation.getConceptQuantity()-1** в алфавитном порядке их полных обозначений. Индексы понятий используются для эффективного доступа к содержимому итератора ассоциации или менеджера изменений ассоциации (см. ниже операции соответствующих интерфейсов). Исключения:

- **NoConcept** – указанное понятие не существует в рамках данной ассоциации.  
string synthesizeDesignation() – получить синтезированное обозначение ассоциации на текущем языке. Под синтезированным обозначением ассоциации подразумевается перечисление полных обозначений включенных в нее понятий в алфавитном порядке через запятую, в круглых скобках, например "(A.B.C, A.B.D, A.E)", где "A.B.C", "A.B.D" и "A.E" – полные обозначения некоторых понятий.

#### 4.1.4 Содержимое ассоциации

Содержимое ассоциации представлено в виде двух интерфейсов: AssociationChanger – менеджера изменений ассоциации, и AssociationIterator – итератора содержимого ассоциации. Оба интерфейса функционируют в контексте некоторого варианта ассоциации, существующего в свою очередь в рамках определенной схемы.

Количество одновременно существующих объектов, реализующих любой из этих интерфейсов, не ограничено. При этом все изменения, производимые через разные объекты, реализующие AssociationChanger, упорядочиваются в одну последовательность изменений текущей транзакции; а каждый объект, реализующий AssociationIterator, возвращает в точности то состояние, которые было на момент получения итератора, без учета любых последующих изменений в данной транзакции или других транзакциях.

<<CORBAInterface>> AssociationChanger		<<CORBAInterface>> AssociationIterator	
+ getSchemaAssociation ()	: SchemaAssociation	+ getSchemaAssociation ()	: SchemaAssociation
+ setValue (long index, Object value)	: void	+ hasNext ()	: boolean
+ setValue (long index, CompoundValue value)	: void	+ next ()	: void
+ setValue (long index, float value)	: void	+ asObject (long index)	: Object
+ setValue (long index, double value)	: void	+ asCompoundValue (long index)	: CompoundValue
+ setValue (long index, char value)	: void	+ asFloat (long index)	: float
+ setValue (long index, short value)	: void	+ asDouble (long index)	: double
+ setValue (long index, long value)	: void	+ asByte (long index)	: char
+ setValue (long index, long long value)	: void	+ asShort (long index)	: short
+ setValue (long index, unsigned char value)	: void	+ asLong (long index)	: long
+ setValue (long index, unsigned short value)	: void	+ asLongLong (long index)	: long long
+ setValue (long index, unsigned long value)	: void	+ asUnsignedByte (long index)	: char
+ setValue (long index, unsigned long long value)	: void	+ asUnsignedShort (long index)	: unsigned short
+ setValue (long index, boolean value)	: void	+ asUnsignedLong (long index)	: unsigned long
+ setValue (long index, string value)	: void	+ asUnsignedLongLong (long index)	: unsigned long long
+ existsAssociationInstance ()	: boolean	+ asTime (long index)	: long
+ createAssociationInstance (boolean wait)	: void	+ asBoolean (long index)	: boolean
+ removeAssociationInstance (boolean wait)	: void	+ asString (long index)	: string
+ clearValues ()	: void		

Причина выбора именно такой архитектуры управления содержимым ассоциаций состоит в том, что она не требует создания большого количества динамических объектов в ходе работы с содержимым ассоциаций (например, не требуется для каждого экземпляра ассоциации создавать свой объект). Это приводит к росту эффективности использования и памяти, и процессора.

Каждый менеджер содержимого ассоциации реализует интерфейс AssociationChanger, который обеспечивает модификацию содержимого того варианта ассоциации, в рамках которого был создан данный объект. Интерфейс AssociationChanger предоставляет перечисленные далее операции.

SchemaAssociation getSchemaAssociation() – получить вариант ассоциации, которому соответствует и в рамках которого был создан данный менеджер содержимого.

void setValue(long index, Object value) – установить для понятия с указанным индексом указанное в виде объекта значение. Объект должен быть того типа, для которого внутри SCM описано отображение в составное значение того типа, в виде которого представляется данное понятие. Отображение объектов в составные значения является предметом отдельной спецификации и может не поддерживаться в начальных реализациях SCM. *Здесь и далее используется индекс понятия, предварительно полученный при помощи SchemaAssociation.getIndex. Данный и все последующие операции установки set присваивают новое значение внутренней переменной, соответствующей указанному понятию, затирая старое значение. Совокупность значений в этих переменных используется другими операциями данного интерфейса (не set, см. ниже). Исключения:*

- NoConcept – указанный индекс не соответствует ни одному понятию данной ассоциации;
- NoValue – указанное значение не является значением того типа, к которому относится указанное понятие;
- NoMapping – указанный объект не может быть отображен в требуемый тип;
- NotSupportedCompoundValueMapping – отображение объектов в составные значения не поддерживается в данной реализации SCM.

*Различие между исключениями NoMapping и NoValue состоит в следующем. NoValue генерируется только том в случае, если требуемый тип является ограниченным, а устанавливаемое значение (после преобразования) соответствует его исходному типу, но не соответствует наложенным ограничениям. В остальных случаях генерируется NoMapping.*

void setValue(long index, CompoundValue value) – установить для понятия с указанным индексом указанное составное значение. Составное значение должно быть того типа, для которого внутри SCM описано отображение в составное значение того типа, в виде которого представляется данное понятие. Отображение составных значений одного типа в другой тип является предметом отдельной спецификации и может не поддерживаться в начальных реализациях SCM. Исключения:

- NoConcept – указанный индекс не соответствует ни одному понятию данной ассоциации;
- NoMapping – указанное составное значение не может быть отображено в требуемый тип;
- NoValue – указанное значение не является значением того типа, к которому относится указанное понятие;
- NotSupportedCompoundValueMapping – отображение составных значений одного типа в другие типы не поддерживается в данной реализации SCM.

void setValue(long index, float value), void setValue(long index, double value), void setValue(long index, char value), void setValue(long index, short value), void setValue(long index, long value), void setValue(long index, long long value), void setValue(long index, unsigned char value), void setValue(long index, unsigned short value), void setValue(long index, unsigned long value), void setValue(long index, unsigned long long value), void setValue(long index, boolean value), void setValue(long index, string value) – установить для понятия с указанным индексом указанное значение. При необходимости осуществляется попытка преобразования указанного значения в значение требуемого типа. Исключения:

- NoConcept – указанный индекс не соответствует ни одному понятию данной ассоциации;
- NoMapping – указанное значение не может быть отображено в требуемый тип;
- NoValue – указанное значение не является значением того типа, к которому относится указанное понятие (например, оно может не попадать в допустимый диапазон значений).

boolean existsAssociationInstance() – проверить, существует ли в текущем состоянии данной схемы тот экземпляр ассоциации, который был задан при помощи операций установки значений понятий set. *Проверка осуществляется именно по текущему состоянию ассоциации, безотносительно к моменту создания данного менеджера изменений (то есть если были зафиксированы транзакции с изменениями в состоянии данной ассоциации, то данная операция учтет все изменения). То же самое верно и для проверки существования экземпляра ассоциации, неявно осуществляемой в ходе создания или удаления экземпляров (см. операции ниже).* Исключения:

- IncompleteInstance – значения понятий заданы не для всех понятий данной ассоциации.

`void createAssociationInstance(boolean wait)` – создать в рамках текущей транзакции тот экземпляр данной ассоциации, который был задан при помощи операций установки значений понятий `set`. Для разделения доступа к экземплярам ассоциаций используется механизм блокировок. Поэтому данная операция имеет параметр – признак ожидания снятия блокировки. В случае обнаружения блокировки, если признак установлен, то данная операция ожидает снятия блокировки неопределенное время, если же признак не установлен, то операция прерывает свою работу с исключением `InstanceInUse`.

- `IncompleteInstance` – значения понятий заданы не для всех понятий данной ассоциации;
- `DuplicateInstance` – уже существует экземпляр ассоциации с тем же набором значений понятий (экземпляров понятий) в рамках текущего состояния схемы;
- `InstanceInUse` – экземпляр ассоциации используется в настоящий момент, доступ к нему заблокирован.

`void removeAssociationInstance(boolean wait)` – удалить в рамках текущей транзакции тот экземпляр данной ассоциации, который был задан при помощи операций установки значений понятий `set`. Для разделения доступа к экземплярам ассоциаций используется механизм блокировок. Поэтому данная операция имеет параметр – признак ожидания снятия блокировки. В случае обнаружения блокировки, если признак установлен, то данная операция ожидает снятия блокировки неопределенное время, если же признак не установлен, то операция прерывает свою работу с исключением `InstanceInUse`.

- `IncompleteInstance` – значения понятий заданы не для всех понятий данной ассоциации;
- `NoInstance` – не существует экземпляра ассоциации с заданным набором значений понятий (экземпляров понятий) в рамках текущего состояния схемы;
- `InstanceInUse` – экземпляр ассоциации используется в настоящий момент, доступ к нему заблокирован.

`void clearValues()` – очистить содержимое всех переменных, хранящих значения понятий. С момент вызова этой операции все понятия данной ассоциации перестают иметь заданные значения. Для использования трех предыдущих операций значения всех этих понятий нужно будет установить заново.

Каждый итератор содержимого ассоциации реализует интерфейс `AssociationIterator`, который обеспечивает просмотр содержимого того варианта ассоциации, в рамках которого был создан данный объект. Интерфейс `AssociationIterator` предоставляет перечисленные далее операции.

`SchemaAssociation getSchemaAssociation()` – получить вариант ассоциации, которому соответствует и в рамках которого был создан данный итератор содержимого.

`boolean hasNext()` – проверить, существовал ли очередной экземпляр данной ассоциации в состоянии схемы на момент создания данного итератора. Если не существовал, то вызовы операции `AssociationIterator.next` приведут к генерации исключения `NoInstance`.

`void next()` – выбрать очередной экземпляр данной ассоциации из того состояния схемы, которое было на момент создания данного итератора. Доступ к значениям понятий выбранного экземпляра ассоциации осуществляется через операции, описанные далее. Эти операции генерируют исключение `NoInstance` (см. второй вариант текста данного исключения), если данная операция не была вызвана ни разу для итератора. В случае, если на момент создания итератора текущий экземпляр ассоциации являлся последним, то данная операция генерирует `NoInstance`.

- `NoInstance` – на момент создания данного итератора не существовал очередной экземпляр данной ассоциации в состоянии схемы.

`Object asObject(long index)` – получить в виде объекта составное значение понятия, имеющего указанный индекс в рамках данной ассоциации. В качестве типа возвращаемого объекта выбирается тот, который задан по умолчанию для данного

понятия (этот аспект является предметом отдельной спецификации по отображению типов). Исключения:

- NoConcept – указанный индекс не соответствует ни одному понятию данной ассоциации;
- NoInstance – ни один экземпляр данной ассоциации не был выбран итератором с момента его создания (требуется хотя бы один вызов `AssociationIterator.next()`);
- NotSupportedCompoundValueMapping – отображение составных значений в объекты не поддерживается в данной реализации SCM;
- NoMapping – запрашиваемое значение не может быть преобразовано в требуемый тип (например, если оно простое).

`CompoundValue asCompoundValue(long index)` – получить составное значение понятия, имеющего указанный индекс в рамках данной ассоциации. Возвращаемое значение относится к типу, в виде которого представляется данное понятие. Исключения:

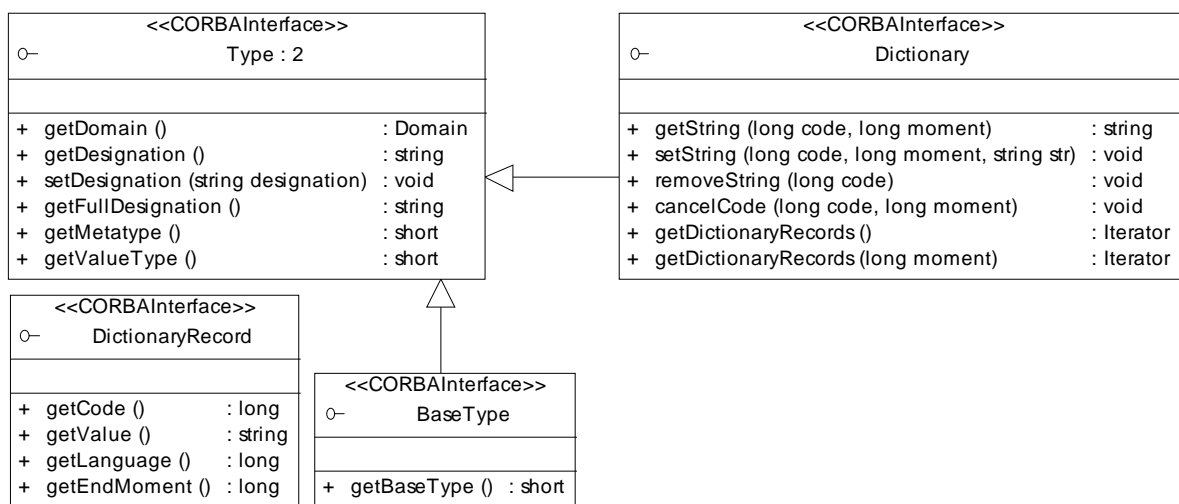
- NoConcept – указанный индекс не соответствует ни одному понятию данной ассоциации;
- NoInstance – ни один экземпляр данной ассоциации не был выбран итератором с момента его создания (требуется хотя бы один вызов `AssociationIterator.next()`);
- NoMapping – запрашиваемое значение не может быть преобразовано в требуемый тип (например, если оно простое).

`float asFloat(long index)`, `double asDouble(long index)`, `char asByte(long index)`, `short asShort(long index)`, `long asLong(long index)`, `long long asLongLong(long index)`, `char asUnsignedByte(long index)`, `unsigned short asUnsignedShort(long index)`, `unsigned long asUnsignedLong(long index)`, `unsigned long long asUnsignedLongLong(long index)`, `long asTime(long index)`, `boolean asBoolean(long index)`, `string asString(long index)` – получить значение понятия, имеющего указанный индекс в рамках данной ассоциации. Если значение понятия на самом деле относится не к тому типу, который запрашивается, то происходит его неявное преобразование к соответствующему типу, если это возможно.

- NoConcept – указанный индекс не соответствует ни одному понятию данной ассоциации;
- NoInstance – ни один экземпляр данной ассоциации не был выбран итератором с момента его создания (требуется хотя бы один вызов `AssociationIterator.next()`);
- NoMapping – запрашиваемое значение не может быть преобразовано в требуемый тип.

#### 4.1.5 Типы

Каждый тип SCM реализует интерфейс `Type`, который предоставляет функциональность, общую для всех типов. Данный общий интерфейс расширяется специальными интерфейсами, предоставляющими специфическую для каждого частного типа функциональность.



Для начала рассмотрим операции, предоставляемые базовым интерфейсом Type. Domain `getDomain()` – получить домен, в котором располагается данный тип. string `getDesignation()` – получить обозначения данного типа на текущем языке. void `setDesignation(string designation)` – установить новое обозначение для данного типа на текущем языке. Изменение обозначений для существующих типов может не поддерживаться. Исключения:

- DuplicateType – тип с указанным обозначением на текущем языке уже существует в данном дереве предметных областей;
- NotSupportedDesignationChanging – изменение обозначений не поддерживается (в том числе для базовых типов).

string `getFullDesignation()` – получить полное обозначение данного типа на текущем языке.

short `getMetatype()` – получить константу метатипа, к которому относится данный тип (см. описание констант в интерфейсе MetatypeConstraints).

short `getValueType()` – получить константу типа значений, в виде которого представляются значения данного типа (см. описание констант в интерфейсе ValueTypeConstants).

#### 4.1.5.1 Базовый тип

Простейшим расширением базового интерфейса Type является интерфейс BaseType, который реализуют все объекты, соответствующие базовым типам. Таких объектов в системе существует ровно столько, сколько существует базовых типов (см. описание констант базовых типов в интерфейсе BaseTypeConstants). Интерфейс BaseType имеет единственную дополнительную операцию.

short `getBaseType()` – получить константу базового типа, которому соответствует данный объект.

#### 4.1.5.2 Справочник

Каждый справочник реализует интерфейс Dictionary, являющийся расширением интерфейса Type и предоставляющий следующие операции по работе с содержимым справочника.

string `getString(long code, long moment)` – получить строковое значение указанного кода на текущем языке, действующее на некоторый момент времени. Действие строкового значения определяется по истории изменения кода (см. описание истории кода). Данная операция возвращает nil (null), если на текущем языке для указанного кода не задано строковое значение на указанный момент времени. Исключения:

- NoValue – указанного кода не существует в данном справочнике.

`void setString(long code, long moment, string str)` – установить новое строковое значение на текущем языке для указанного кода, начинающее свое действие с указанного момента. Указанный момент времени должен быть позднее или равен моменту начала действия последнего строкового значения указанного кода на текущем языке. В этом случае в указанный момент прежнее строковое значение прекращает свое действие и начинает действовать новое строковое значение. Исключения:

- `InterferedStringValue` – два строковых значения одного кода на одном языке не могут действовать одновременно.

`void removeString(long code)` – удалить последнее действующее строковое значение указанного кода на текущем языке. Если существует предыдущее строковое значение, то оно теряет дату завершения своего действия и становится новым последним действующим значением. Исключения:

- `NoString` – не существует строкового значения на текущем языке для данного кода.

`void cancelCode(long code, long moment)` – отменить действие всех строковых значений на любых языках для указанного кода, начиная с указанного момента времени. С этого момента все строковые значения становятся пустыми до тех пор, пока для них не будут установлены новые значения (в случае восстановления действия кода). При восстановлении операцией `Dictionary.setString` строковых значений для прежде отмененного кода, информация о периоде отмены теряется, вновь установленные строковые значения будут действовать весь период отмены. При удалении операцией `Dictionary.removeString` последнего действующего строкового значения для отмененного кода, происходит восстановление действия предыдущего строкового значения текущего языка так, как если бы удаления и не было. Указанный момент времени должен быть позднее или равен моменту начала действия последнего строкового значения указанного кода для всех языков. Исключения:

- `InterferedStringValue` – строковое значение кода не может действовать и быть отмененным одновременно.

***Операции `setString`, `cancelCode` и `removeString` действуют по принципу стека: `setString` помещает новое действующее строковое значение на вершину стека, `cancelCode` помещает на вершину стека признак отсутствия каких-либо действующего строкового значения, метод `removeString` удаляет с вершины стека любой из этих двух элементов, восстанавливая действие предыдущего. Особенностью данного стека является то, что помещение на вершину стека нового действующего строкового значения поверх признака отсутствия действующих значений сопровождается предварительным неявным снятием со стека последнего элемента [признака отсутствия действующих значений].***

`Iterator getDictionaryRecords()` – получить все записи справочника, как истории кодов, так и самих кодов, на текущем языке. Записи, действующие в настоящий момент, будут иметь момент окончания действия равный максимальному числу типа `long`. Итератор содержит объекты, реализующие интерфейс `DictionaryRecord` (см. его описание далее).

`Iterator getDictionaryRecords(long moment)` – получить все записи справочника, действовавшие (действующие) на заданный момент времени на текущем языке. Записи, действующие в настоящий момент, будут иметь момент окончания действия равный максимальному числу типа `long`. Итератор содержит объекты, реализующие интерфейс `DictionaryRecord`.

Доступ к содержимому записей справочников реализуется посредством интерфейса `DictionaryRecord`, предоставляющего следующие операции.

`long getCode()` – получить код данной записи справочника.

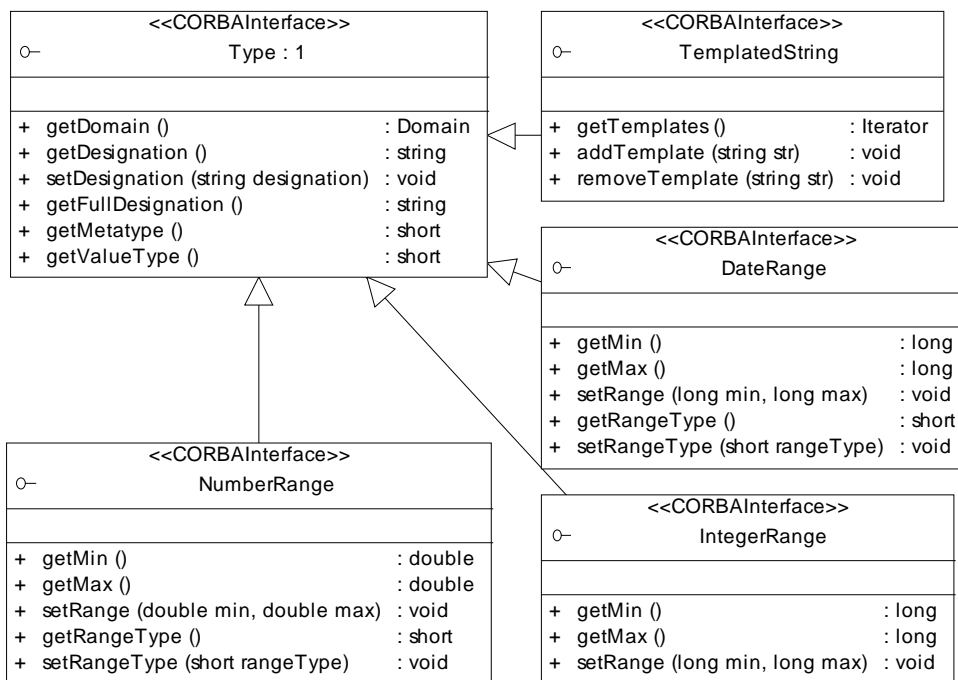
`string getValue()` – получить строковое значение данной записи справочника.

`long getLanguage()` – получить язык данной записи справочника.

long getEndMoment() – получить момент завершения действия данной записи справочника (в этот момент действие данной записи считается уже завершённым). Возвращает максимальное число тип long, если действие данной записи продолжается в настоящий момент.

### 4.1.5.3 Диапазон чисел

Рассмотрим следующую группу интерфейсов, расширяющих базовый интерфейс Type: NumberRange, IntegerRange, DateRange, TemplatedString. Некоторые из этих интерфейсов соответствуют только одному метатипу SCM, другие соответствуют многим метатипам, характеризующимся общим способом описания типов.



Интерфейс NumberRange является расширением базового интерфейса Type, соответствует единственному метатипу – диапазону чисел – и предоставляет следующие операции.

double getMin() – получить минимальную границу числового диапазона;

double getMax() – получить максимальную границу числового диапазона;

void setRange(double min, double max) – установить новую границу числового диапазона. Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- TypeInUse – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- BadRange – минимальная граница диапазона больше его максимальной границы;
- NotSupportedTypeChanging – изменение параметров типа не поддерживается в данной реализации SCM.

short getRangeType() – получить тип границ числового диапазона;

void setRangeType(short rangeType) – установить новый тип границ числового диапазона. Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- TypeInUse – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- NoRangeTypeConstant – не существует указанной константы типа границ;

- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

#### 4.1.5.4 Типы, описываемые целочисленным диапазоном

Интерфейс `IntegerRange` является расширением базового интерфейса `Type` и соответствует нескольким метатипам: диапазону целых чисел, диапазону кодов и строке ограниченной длины. Все эти метатипы описываются единым набором параметров – двумя числами типа `long`. Данный интерфейс предоставляет следующие операции.

`long getMin()` – получить минимальную границу числового диапазона или минимальный предел длины строки;

`long getMax()` – получить максимальную границу числового диапазона или максимальный предел длины строки;

`void setRange(long min, long max)` – установить новые границы числового диапазона или пределы длины строки. Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `BadRange` – минимальная граница диапазона больше его максимальной границы (генерируется только для диапазона целых чисел и диапазона кодов);
- `BadLimits` – минимальный предел больше максимального предела длины строки (генерируется только для строки ограниченной длины).
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

#### 4.1.5.5 Диапазон дат

Интерфейс `DateRange` является расширением базового интерфейса `Type`, соответствует единственному метатипу – диапазону дат – и предоставляет следующие операции.

`long getMin()` – получить минимальную границу диапазона дат;

`long getMax()` – получить максимальную границу диапазона дат;

`void setRange(long min, long max)` – установить новую границу диапазона дат.

Данная функциональность может не поддерживаться в начальных реализациях SCM.

Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `BadRange` – минимальная граница диапазона больше его максимальной границы;
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

`short getRangeType()` – получить тип границ диапазона дат;

`void setRangeType(short rangeType)` – установить новый тип границ диапазона дат.

Данная функциональность может не поддерживаться в начальных реализациях SCM.

Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `NoRangeTypeConstant` – не существует указанной константы типа границ;
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

#### 4.1.5.6 Строка по шаблону

Интерфейс `TemplatedString` является расширением базового интерфейса `Type`, соответствует единственному метатипу – строке по шаблону – и предоставляет следующие операции.

`Iterator getTemplates()` – получить совокупность шаблонов, на которых построена данная строка по шаблону. Итератор содержит только объекты типа `string`. Обязательно присутствует хотя бы один элемент в итераторе.

`void addTemplate(string str)` – добавить новый шаблон в данную строку по шаблону. Данная функциональность может не поддерживаться в начальных реализациях SCM.

Исключения:

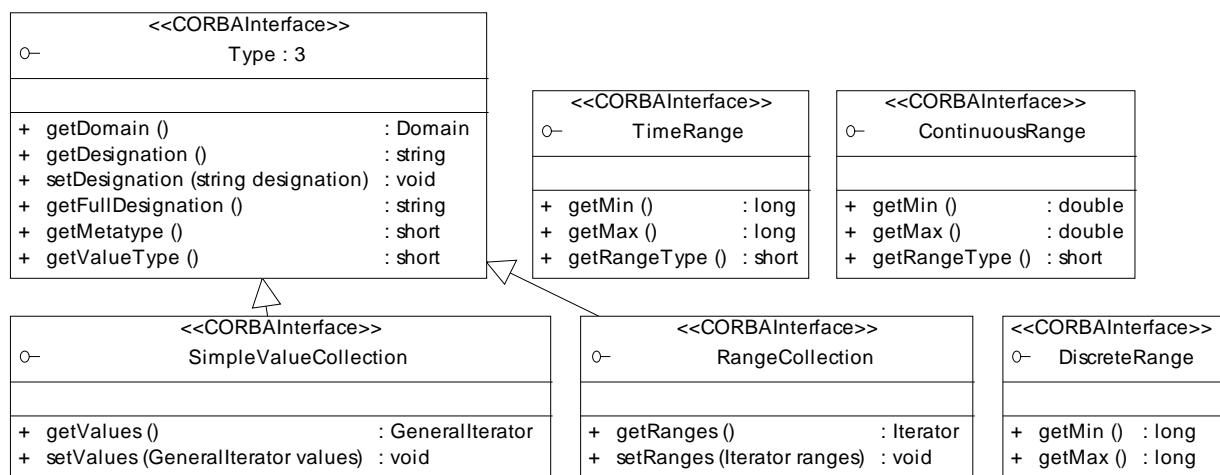
- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `DuplicateTemplate` – уже существует указанный шаблон в данной строке по шаблону;
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

`void removeTemplate(string str)` – удалить шаблон из данной строки по шаблону. Не допускается удаление последнего шаблона в данной строке. Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `NoTemplate` – указанного шаблона не существует в рамках данной строки по шаблону;
- `LastTemplate` – последний шаблон не может быть удален из данной строки по шаблону;
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

#### 4.1.5.7 Набор простых значений

Интерфейс `SimpleValueCollection` представляет собой набор простых значений, является расширением базового интерфейса `Type` и соответствует метатипам: набор чисел, набор целых чисел, набор дат, набор строк, набор кодов.



Интерфейс `SimpleValueCollection` предоставляет следующие операции.

`GeneralIterator getValues()` – получить совокупность значений данного набора. **Тип значений итератора соответствует исходному типу данного набора значений: для чисел – DOUBLE, для целых чисел – LONG, для дат – LONG, для строк – STRING, для кодов – LONG. То же самое верно для операции SimpleValueCollection.setValues (см. далее).**

`void setValues(GeneralIterator values)` – установить совокупность значений данного набора. Тип значений итератора соответствует исходному типу данного набора значений (см. предыдущую операцию). Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `BadElementType` – тип элемента итератора не соответствует исходному типу данного набора значений;
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

#### 4.1.5.8 Набор диапазонов

Интерфейс `RangeCollection` представляет собой набор диапазонов, является расширением базового интерфейса `Type` и соответствует метатипам: набор диапазонов чисел, набор диапазонов целых чисел, набор диапазонов дат, набор диапазонов кодов. Интерфейс `RangeCollection` предоставляет следующие операции.

`Iterator getRanges()` – получить совокупность диапазонов данного набора. Элементы итератора реализуют один из трех интерфейсов: `TimeRange`, `ContinuousRange` или `DiscreteRange`, в соответствии с исходным типом данного набора диапазонов. **Если данный тип является набором диапазонов чисел, то все элементы итератора реализуют интерфейс `ContinuousRange`. Если данный тип является набором диапазонов целых чисел или набором диапазонов кодов, то все элементы итератора реализуют интерфейс `DiscreteRange`. Если данный тип является набором диапазонов дат, то все элементы итератора реализуют интерфейс `TimeRange`. Те же правила действуют для итератора, являющегося входным параметром операции `RangeCollection.setRanges`.**

`void setRanges(Iterator ranges)` – установить совокупность диапазонов данного набора. Элементы итератора реализуют один из трех интерфейсов: `TimeRange`, `ContinuousRange` или `DiscreteRange`, в соответствии с исходным типом данного набора диапазонов (см. предыдущую операцию). Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `BadElementType` – тип элемента итератора не соответствует исходному типу данного набора диапазонов;
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

Дискретный диапазон должен реализовывать интерфейс `DiscreteRange`, предоставляющий следующие операции.

`long getMin()` – получить минимальную границу диапазона.

`long getMax()` – получить максимальную границу диапазона.

Непрерывный диапазон должен реализовывать интерфейс `ContinuousRange`, предоставляющий следующие операции.

`double getMin()` – получить минимальную границу диапазона.

`double getMax()` – получить максимальную границу диапазона.

`short getRangeType()` – получить тип границ диапазона.

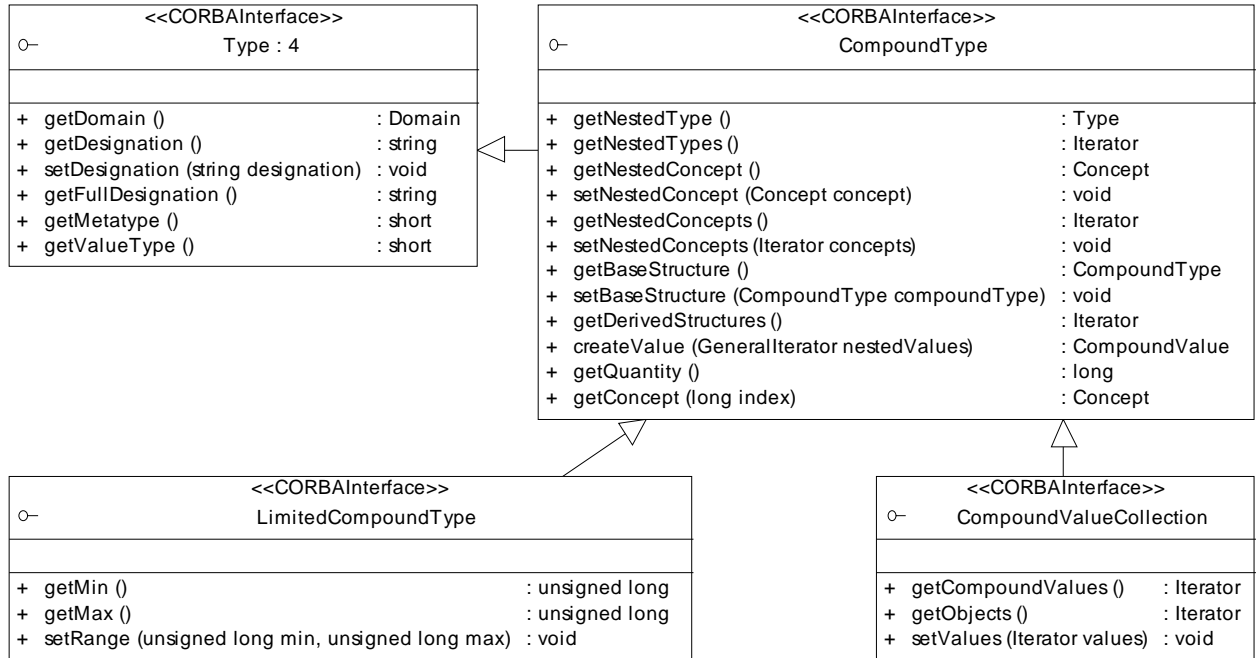
Диапазон дат должен реализовывать интерфейс `TimeRange`, предоставляющий следующие операции.

`long getMin()` – получить минимальную границу диапазона.

long getMax() – получить максимальную границу диапазона.  
short getRangeType() – получить тип границ диапазона.

#### 4.1.5.9 Составной тип

Составной тип реализует интерфейс CompoundType, являющийся расширением интерфейса Type и предоставляющий следующие операции.



Type getNestedType() – получить вложенный тип, если он один. Если вложенных типов несколько – генерирует исключение MultipleTypes. Исключения:

- MultipleTypes – данный тип имеет больше одного вложенного типа.

Iterator getNestedTypes() – получить совокупность всех вложенных типов (только собственных вложенных типов в случае структуры). Результирующий итератор содержит хотя бы один элемент. Все элементы результирующего итератора реализуют интерфейс Type.

Concept getNestedConcept() – получить вложенное понятие, если оно одно. Если вложенных понятий несколько – генерирует исключение MultipleConcepts. Исключения:

- MultipleConcepts – данный тип имеет больше одного вложенного понятия.

void setNestedConcept(Concept concept) – установить единственное вложенное понятие для данного типа. Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- TypeInUse – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- NoConcept – указанное понятие не существует в данном дереве предметных областей;
- NotSupportedTypeChanging – изменение параметров типа не поддерживается в данной реализации SCM.

Iterator getNestedConcepts() – получить совокупность всех вложенных понятий данного типа (только собственных вложенных понятий в случае структуры). Результирующий итератор содержит хотя бы один элемент. Все элементы результирующего итератора реализуют интерфейс Concept.

void setNestedConcepts(Iterator concepts) – установить вложенные понятия (только собственные вложенные понятия в случае структуры) данного типа в соответствии с указанной совокупностью. Итератор должен содержать только элементы, реализующие интерфейс Concept; итератор должен содержать хотя бы один элемент; элементы

итератора не должны повторяться. Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `NoConcept` – одно из понятий не существует в данном дереве предметных областей;
- `BadElementType` – неверный тип элемента итератора (ожидается тип, реализующий интерфейс `Concept`);
- `RepeatedElements` – итератор содержит повторяющиеся понятия;
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент);
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

`CompoundType getBaseStructure()` – получить базовую структуру для данного типа. Возвращает `nil (null)` в случае, если данный тип не является структурой или не имеет базовой структуры. Возвращаемый составной тип является структурой.

`void setBaseStructure(CompoundType compoundType)` – установить новую базовую структуру для данной структуры. И данный, и указанный составные типы должны быть типами структур, при нарушении данного ограничения выдается исключение `NoStructure` в обоих случаях. Если указанный в качестве параметра составной тип равен `nil (null)`, то данный тип перестает быть производным (если он не был производным, то ничего не происходит). Если обнаруживается цикл в дереве наследования структур, то выдается исключение `CycleDetected`.

- `NoStructure` – указанный составной тип не является структурой или типом данного дерева предметных областей;
- `CycleDetected` – обнаружен цикл в дереве наследования структур.

`Iterator getDerivedStructure()` – получить совокупность всех производных структур. Возвращает `nil (null)` в случае, если данный составной тип не является структурой. Возвращает пустой итератор, если производных структур нет. Все элементы результирующего итератора реализуют интерфейс `CompoundType` и являются структурами.

`CompoundValue createValue(GeneralIterator nestedValues)` – создать значение данного составного типа по вложенным значениям. Итератор должен возвращать вложенные значения в порядке их позиций внутри составного значения. Вложенные значения должны быть тех типов, которые ожидаются в каждой из позиций (см. описание составных типов). Итератор не может быть пустым. Исключения:

- `BadElementType` – неверный тип элемента итератора (ожидается тип, соответствующий позиции элемента);
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент).

`long getQuantity()` – получить количество вложенных типов/понятий данного составного типа.

`Concept getConcept(long index)` – получить вложенное понятие по номеру его позиции в рамках данного составного типа.

- `NoConcept` – указанный индекс не соответствует ни одному понятию в рамках данного составного типа.

#### 4.1.5.10 Ограниченный составной тип

Одним из расширений интерфейса `CompoundType` является интерфейс `LimitedCompoundType`, представляющий собой ограниченный составной тип. Данный интерфейс реализуют объекты, являющиеся ограниченными векторами или ограниченными множествами (они имеют общий интерфейс, так как описываются одинаковым набором параметров). Данный интерфейс предоставляет следующие операции.

`unsigned long getMin()` – получить минимальный предел количества элементов ограниченного составного типа.

`unsigned long getMax()` – получить максимальный предел количества элементов ограниченного составного типа.

`void setRange(unsigned long min, unsigned long max)` – установить новые пределы количества элементов составного типа. Данная функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `BadLimits` – минимальный предел больше максимального предела;
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM.

#### 4.1.5.11 Набор составных значений

Другим расширением интерфейса `CompoundType` является интерфейс `CompoundValueCollection`, представляющий собой набор составных значений. Данный интерфейс реализуется типами: набор значений структуры, набор значений вектора, набор значений множества, и предоставляет следующие операции.

`Iterator getCompoundValues()` – получить совокупность составных значений, определяющих данный набор. Все элементы итератора реализуют интерфейс `CompoundValue`.

`Iterator getObjects()` – получить совокупность составных значений, определяющих данный набор, в виде объектов. Все элементы итератора относятся к тому типу, в который отображается исходный тип (он является составным) данного набора значений. Данная функциональность может не поддерживаться в начальных реализациях SCM.

Исключения:

- `NoMapping` – исходный тип данного набора значений не имеет отображения в объектный тип;
- `NotSupportedCompoundValueMapping` – отображение составных значений в объекты не поддерживается в данной реализации SCM.

`void setValues(Iterator values)` – установить для данного набора новую совокупность составных значений. Все элементы итератора должны реализовывать интерфейс `CompoundValue` и принадлежать исходному типу данного набора, или являться объектами, отображаемыми к исходному типу данного набора. Итератор должен содержать по крайней мере один элемент и элементы не должны повторяться. Функциональность отображения и изменения типа могут не поддерживаться в начальных реализациях SCM.

Исключения:

- `TypeInUse` – тип используется существующими составными значениями, понятиями или составными понятиями, а поэтому не может быть изменен;
- `NoMapping` – тип, к которому относятся элементы итератора, не имеет отображения в исходный тип данного набора значений (генерируется только в случае наличия поддержки отображения объектов и составных значений);
- `BadElementType` – неверный тип элемента итератора (генерируется, если элемент итератора реализует интерфейс `CompoundValue`, но относится не к тому типу, к которому относится исходный тип данного набора);
- `RepeatedElements` – итератор содержит повторяющиеся элементы;
- `EmptyIterator` – пустой итератор (в итераторе должен быть хотя бы один элемент);
- `NotSupportedTypeChanging` – изменение параметров типа не поддерживается в данной реализации SCM;
- `NotSupportedCompoundValueMapping` – отображение объектов в составные значения не поддерживается в данной реализации SCM.

## 4.1.6 Составное значение

Каждое составное значение SCM реализует интерфейс `CompoundValue`, предоставляющий перечисленные далее операции.

<<CORBAInterface>>	
o-	CompoundValue
+ <code>getType ()</code>	: <code>CompoundType</code>
+ <code>asObject ()</code>	: <code>Object</code>
+ <code>asObject (long index)</code>	: <code>Object</code>
+ <code>asCompoundValue (long index)</code>	: <code>CompoundValue</code>
+ <code>asFloat (long index)</code>	: <code>float</code>
+ <code>asDouble (long index)</code>	: <code>double</code>
+ <code>asByte (long index)</code>	: <code>char</code>
+ <code>asShort (long index)</code>	: <code>short</code>
+ <code>asLong (long index)</code>	: <code>long</code>
+ <code>asLongLong (long index)</code>	: <code>long long</code>
+ <code>asUnsignedByte (long index)</code>	: <code>char</code>
+ <code>asUnsignedShort (long index)</code>	: <code>unsigned short</code>
+ <code>asUnsignedLong (long index)</code>	: <code>unsigned long</code>
+ <code>asUnsignedLongLong (long index)</code>	: <code>unsigned long long</code>
+ <code>asTime (long index)</code>	: <code>long</code>
+ <code>asBoolean (long index)</code>	: <code>boolean</code>
+ <code>asString (long index)</code>	: <code>string</code>

`CompoundType getType()` – получить тип, к которому относится данное составное значение.

`Object asObject()` – получить данное составное значение в виде объекта. Эта функциональность может не поддерживаться в начальных реализациях SCM.

Исключения:

- `NoMapping` – данный составной тип не имеет отображения в какой-либо объектный тип;
- `NotSupportedCompoundValueMapping` – отображение составных значений в объекты не поддерживается в данной реализации SCM.

`Object asObject(long index)` – получить вложенное составное значение, находящееся в указанной позиции, в виде объекта. Эта функциональность может не поддерживаться в начальных реализациях SCM. Исключения:

- `NoConcept` – указанный индекс не соответствует ни одному понятию в рамках данного составного типа;
- `NoMapping` – указанный вложенный тип (в том числе простой вложенный тип) не может быть отображен в какой-либо объектный тип;
- `NotSupportedCompoundValueMapping` – отображение составных значений в объекты не поддерживается в данной реализации SCM.

`CompoundValue asCompoundValue(long index)` – получить вложенное составное значение, находящееся в указанной позиции. Исключения:

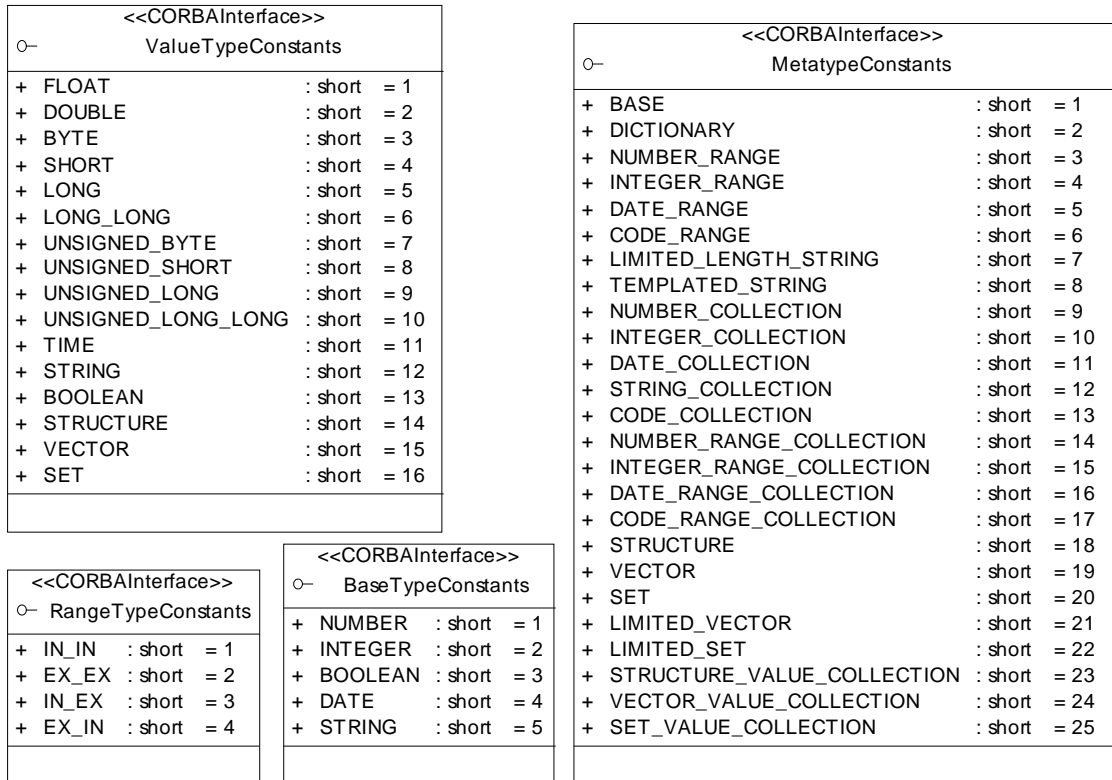
- `NoConcept` – указанный индекс не соответствует ни одному понятию в рамках данного составного типа;
- `NoMapping` – указан простой вложенный тип, который не может быть отображен в составной тип.

`float asFloat(long index)`, `double asDouble(long index)`, `char asByte(long index)`, `short asShort(long index)`, `long asLong(long index)`, `long long asLongLong(long index)`, `char asUnsignedByte(long index)`, `unsigned short asUnsignedShort(long index)`, `unsigned long asUnsignedLong(long index)`, `unsigned long long asUnsignedLongLong(long index)`, `long asTime(long index)`, `boolean asBoolean(long index)`, `string asString(long index)` – получить вложенное значение заданного простого типа, находящееся в указанной позиции. Исключения:

- NoConcept – указанный индекс не соответствует ни одному понятию в рамках данного составного типа;
- NoMapping – указанный вложенный тип не может быть отображен в заданный простой тип.

#### 4.1.7 Константы

SCM определяет ряд наборов констант, описанных при помощи отдельных интерфейсов.



Константы всех метатипов перечислены в интерфейсе MetatypeConstants:

- BASE – "Базовый";
- DICTIONARY – "Справочник";
- NUMBER\_RANGE – "Диапазон чисел";
- INTEGER\_RANGE – "Диапазон целых чисел";
- DATE\_RANGE – "Диапазон дат";
- CODE\_RANGE – "Диапазон кодов";
- LIMITED\_LENGTH\_STRING – "Строка ограниченной длины";
- TEMPLATED\_STRING – "Строка по шаблону";
- NUMBER\_COLLECTION – "Набор чисел";
- INTEGER\_COLLECTION – "Набор целых чисел";
- DATE\_COLLECTION – "Набор дат";
- STRING\_COLLECTION – "Набор строк";
- CODE\_COLLECTION – "Набор кодов";
- NUMBER\_RANGE\_COLLECTION – "Набор диапазонов чисел";
- INTEGER\_RANGE\_COLLECTION – "Набор диапазонов целых чисел";
- DATE\_RANGE\_COLLECTION – "Набор диапазонов дат";
- CODE\_RANGE\_COLLECTION – "Набор диапазонов кодов";
- STRUCTURE – "Структура";

- VECTOR – "Вектор";
- SET – "Множество";
- LIMITED\_VECTOR – "Ограниченный вектор";
- LIMITED\_SET – "Ограниченное множество";
- STRUCTURE\_VALUE\_COLLECTION – "Набор значений структуры";
- VECTOR\_VALUE\_COLLECTION – "Набор значений вектора";
- SET\_VALUE\_COLLECTION – "Набор значений множества".

Константы всех базовых типов перечислены в интерфейсе BaseTypeConstants:

- NUMBER – "Число";
- INTEGER – "Целое число";
- BOOLEAN – "Булево число";
- DATE – "Дата";
- STRING – "Строка".

Константы всех типов, которые используются вычислительными системами для хранения значений переменных, перечислены в интерфейсе ValueTypeConstants:

- FLOAT – вещественный;
- DOUBLE – вещественный двойной точности;
- BYTE – байт;
- SHORT – короткое целое;
- LONG – длинное целое;
- LONG\_LONG – длинное целое двойной точности;
- UNSIGNED\_BYTE – байт без знака;
- UNSIGNED\_SHORT – короткое целое без знака;
- UNSIGNED\_LONG – длинное целое без знака;
- UNSIGNED\_LONG\_LONG – длинное целое двойной точности без знака;
- TIME – время (как длинное целое);
- STRING – строка;
- BOOLEAN – булево число;
- STRUCTURE – структура;
- VECTOR – вектор;
- SET – множество.

Константы всех типов границ диапазонов перечислены в интерфейсе RangeTypeConstants:

- IN\_IN – обе границы включены: [];
- EX\_EX – обе границы исключены: ( );
- IN\_EX – левая граница включена, правая – исключена: [ );
- EX\_IN – левая граница исключена, правая – включена: ( ].

#### 4.1.8 Итераторы

SCM предоставляет два типа итераторов, представленных интерфейсами Iterator и GeneralIterator. Iterator предназначен для перебора совокупности элементов объектного (пользовательского) типа. GeneralIterator предназначен для перебора совокупности элементов любого типа: как объектного, так и простого. При работе с любым итератором необходимо осознавать, что он не содержит копий перебираемых объектов, и все изменения объекта-носителя данных немедленно сказываются на том, что возвращает итератор. Поэтому для того, чтобы гарантировать неизменность перебираемой совокупности, необходимо

- либо не выполнять операций над объектами, которые могут привести к изменению содержимого итератора;
- либо сделать копию содержимого итератора сразу после его получения.

<<CORBAInterface>> Iterator	
+ hasNext ()	: boolean
+ next ()	: Object

<<CORBAInterface>> GeneralIterator	
+ hasNext ()	: boolean
+ next ()	: void
+ getValueType ()	: short
+ asObject ()	: Object
+ asFloat ()	: float
+ asDouble ()	: double
+ asByte ()	: char
+ asShort ()	: short
+ asLong ()	: long
+ asLongLong ()	: long long
+ asUnsignedByte ()	: char
+ asUnsignedShort ()	: unsigned short
+ asUnsignedLong ()	: unsigned long
+ asUnsignedLongLong ()	: unsigned long long
+ asTime ()	: long
+ asBoolean ()	: boolean
+ asString ()	: string

Интерфейс Iterator предоставляет следующие операции:

boolean hasNext() – проверить, существует ли очередной элемент в итераторе.

Данная операция не приводит к переходу на очередной элемент, а поэтому может вызываться многократно без последствий. Если ее первое выполнение возвращает false, то это означает, что итератор пустой.

Object next() – получить очередной элемент итератора (возвращает nil, если очередного элемента нет). Первое выполнение данной операции приводит к возврату первого элемента, второе – второго элемента, и так далее.

Интерфейс GeneralIterator предоставляет следующие операции:

boolean hasNext() – проверить, существует ли очередной элемент в итераторе.

Данная операция не приводит к переходу на очередной элемент, а поэтому может вызываться многократно без последствий. Если ее первое выполнение возвращает false, то это означает, что итератор пустой.

void next() – делает очередной элемент итератора текущим. Данная операция генерирует исключение, если очередного элемента не существует (то есть все элементы перебраны, или их не было). До того, как данная операция будет вызвана в первый раз, текущий элемент считается пустым. Текущий элемент также становится пустым в момент, когда совершается попытка перейти к несуществующему очередному элементу. Все попытки получить доступ к пустому текущему элементу приведут к генерации соответствующего исключения (см. операции ниже). Исключения:

- NoNextElement – очередного элемента итератора не существует.

short getValueType() – получить константу типа, в виде которого представляются значения итератора (см. описание констант в интерфейсе ValueTypeConstants). Итератор не может содержать элементы, относящиеся к различным типам значений, поэтому результат данной операции является общим для всех элементов.

Object asObject(), float asFloat(), double asDouble(), char asByte(), short asShort(), long asLong(), long long asLongLong(), char asUnsignedByte(), unsigned short asUnsignedShort(), unsigned long asUnsignedLong(), unsigned long long asUnsignedLongLong(), long asTime(), boolean asBoolean(), string asString() – получить значение текущего элемента итератора в виде заданного типа.

- NoCurrentElement – текущий элемент итератора не выбран;
- NoMapping – текущий элемент не может быть отображен в заданный тип.

## 4.1.9 Ограничение

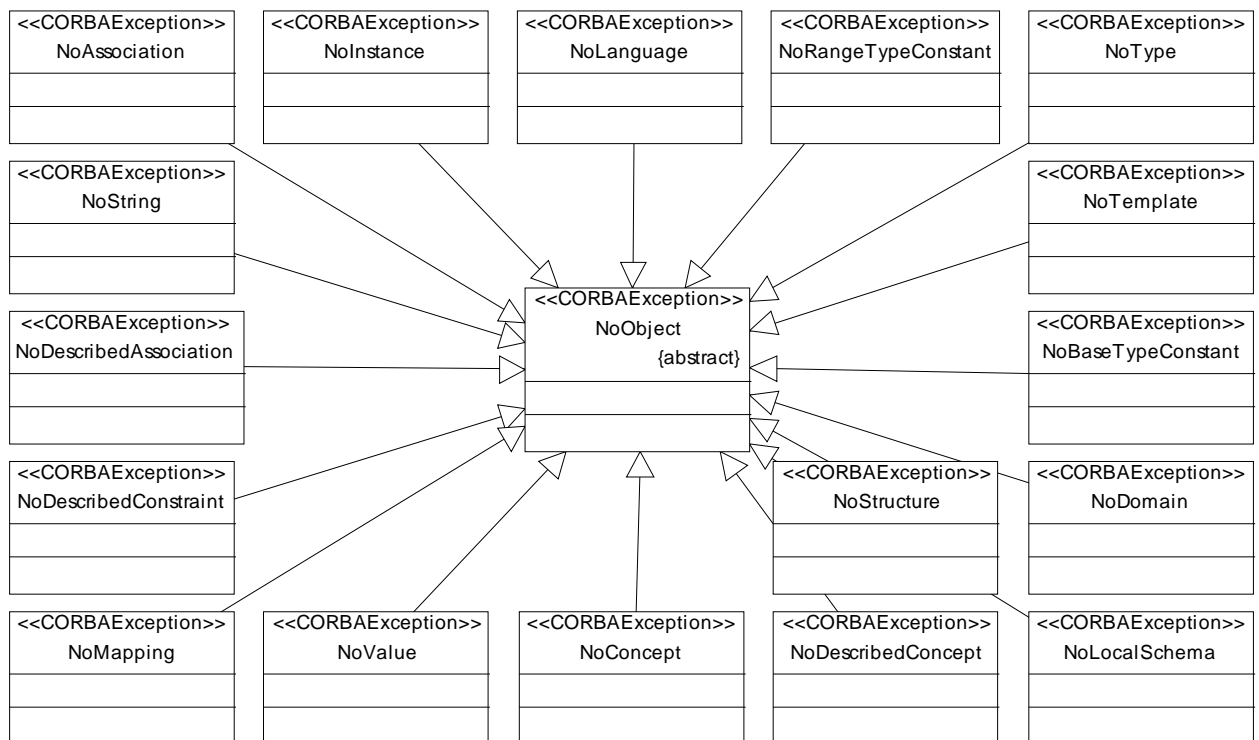
Использованные выше классы **Model.SCM.CL.Constraint** и **Model.SCM.CL.ConstraintBody** вводятся в отдельной спецификации SCM-CL-SPEC. См. данную спецификацию для получения полной информации о языке ограничений SCM CL.

## 4.1.10 Исключения

Исключения SCM делятся на две категории – базовые и генерируемые. Базовые исключения сами по себе не генерируются (являются абстрактными) и служат для обобщения ряда производных исключений по некоторому признаку. Базовые исключения:

- NoObject – искомый объект не найден;
- DuplicateObject – объект дублируется;
- NotSupportedFeature – функциональная возможность не поддерживается;
- ObjectInUse – удаляемый объект используется и не может быть удален;
- IncorrectIterator – нарушение требований к итератору;
- IncorrectRange – нарушение требований к диапазону;
- MultipleObjects – критерию поиска удовлетворяют множество объектов.

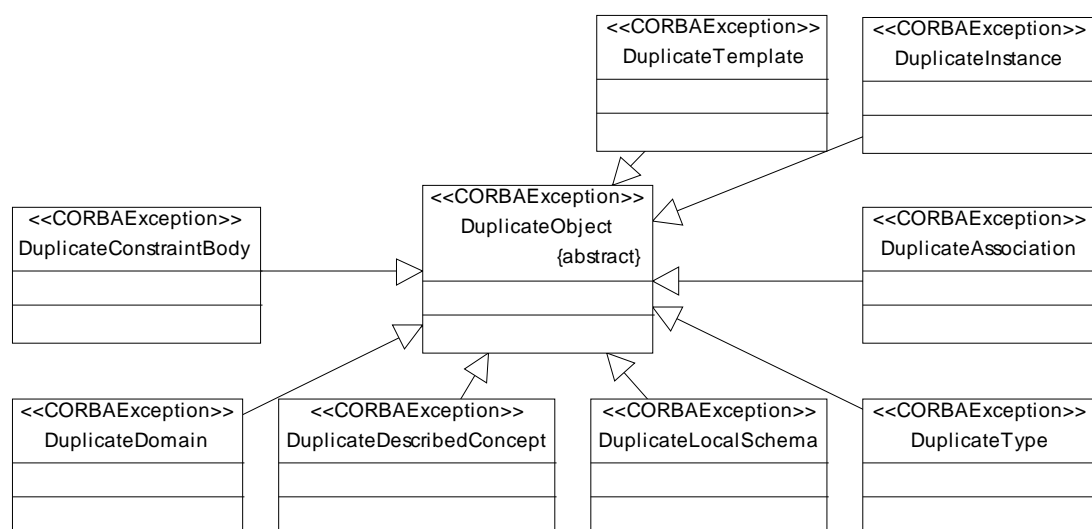
Далее перечислим генерируемые исключения, группируя их в рамках соответствующих базовых при необходимости.



Генерируемые исключения, производные от NoObject:

- NoDomain: "Domain %s does not exist" – указанная предметная область не существует в данном дереве предметных областей.
- NoConcept: "Concept %s does not exist" – указанное понятие не существует в данном дереве предметных областей или в рамках данной ассоциации.
- NoConcept: "Index %s does not suit any concept of %s" – указанный индекс не соответствует ни одному понятию ассоциации или составного типа.
- NoDescribedConcept: "Concept %s does not exist as being described" – указанное понятие не существует в качестве описанного.
- NoLocalSchema: "Local schema %s does not exist" – указанная локальная схема не существует в рамках данного дерева предметных областей.

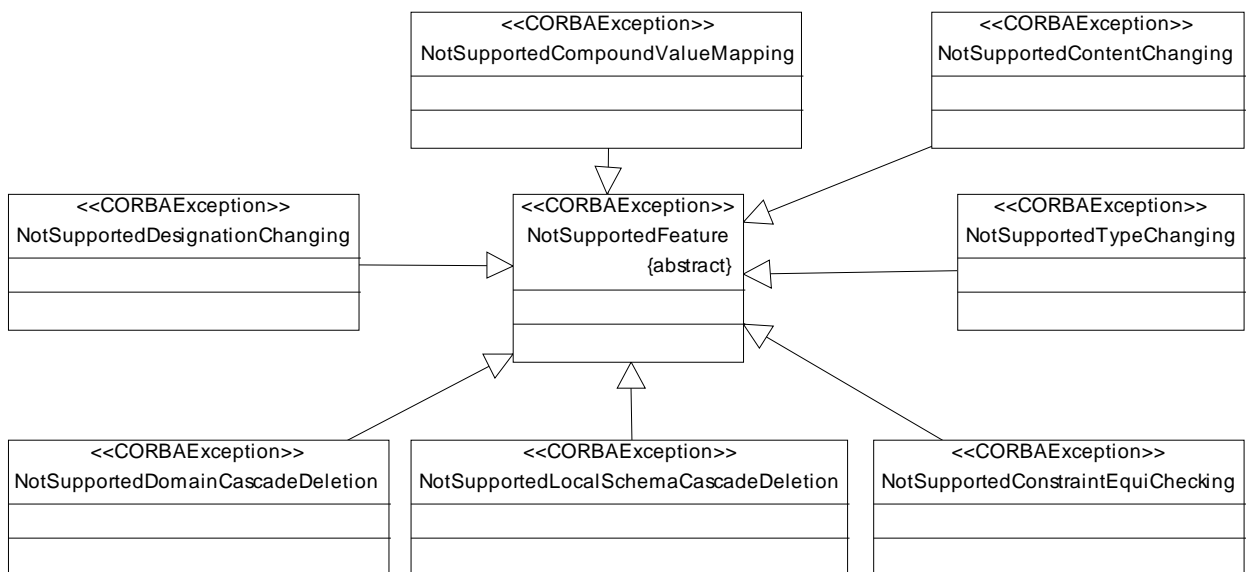
- NoBaseTypeConstant: "There is no a base type constant %s" – не существует указанной константы базового типа.
- NoType: "Type %s does not exist" – указанного типа не существует в рамках данного дерева предметных областей.
- NoRangeTypeConstant: "There is no a range type constant %s" – не существует указанной константы типа границ.
- NoLanguage: "There is no a language %s" – указанного языке не существует.
- NoAssociation: "There is no an association %s within the schema %s" – искомой ассоциации не существует в рамках данной схемы.
- NoDescribedAssociation: "There is no a described association %s within the schema %s" – искомой описанной ассоциации не существует в рамках данной схемы (при этом может существовать неописанная порожденная ассоциация).
- NoDescribedConstraint: "There is no a described constraint %s within the schema %s" – искомое описанное ограничение не существует в рамках данной схемы.
- NoMapping: "The object or value %s can not be mapped to the required type %s" – указанный объект или значение не может быть отображено в требуемый тип.
- NoMapping: "The type %s has no a mapping" – указанный тип не имеет отображения в какой-либо объектный тип.
- NoValue: "The value %s does not belong to the type %s" – указанное значение не является значением того типа, к которому относится указанное понятие.
- NoInstance: "There is no an association instance %s within the schema %s" – не существует экземпляра ассоциации с заданным набором значений понятий (экземпляров понятий) в рамках текущего состояния схемы.
- NoInstance: "There is no a next or selected association instance of the association %s within the schema %s" – не существует очередного или выбранного экземпляра данной ассоциации в состоянии схемы.
- NoString: "There is no a string value for the code %s" – не существует строкового значения для данного кода.
- NoTemplate: "There is no a template %s within the templated string %s" – указанного шаблона не существует в рамках заданной строки по шаблону.
- NoStructure: "There is no a structure %s" – указанный составной тип не является структурой или типом данного дерева предметных областей.



Генерируемые исключения, производные от DuplicateObject:

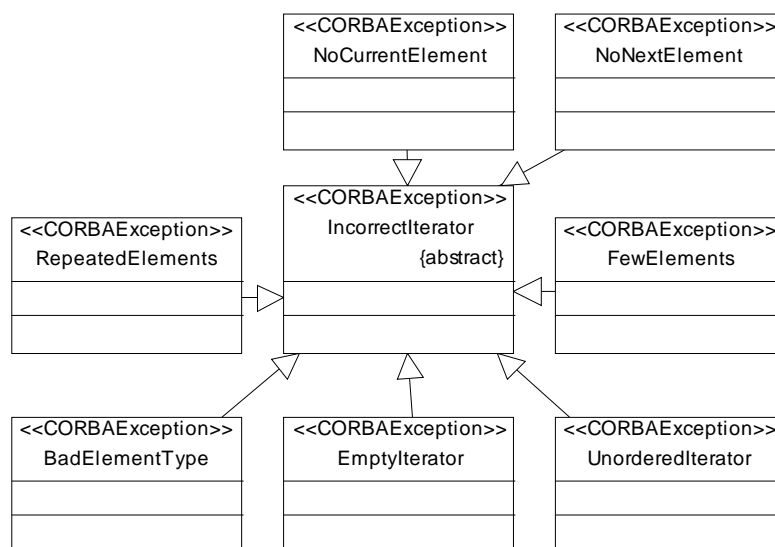
- DuplicateDomain: "Domain %s already exists" – уже существует вложенная предметная область с тем же обозначением в данном дереве предметных областей.

- DuplicateDescribedConcept: "Described concept %s already exists" – уже существует понятие с тем же обозначением в данном дереве предметных областей.
- DuplicateLocalSchema: "Local schema %s already exists" – уже существует локальная схема с тем же обозначением в данном дереве предметных областей.
- DuplicateType: "Type %s already exists" – тип с указанным обозначением уже существует в данном дереве предметных областей.
- DuplicateAssociation: "Association %s already exists" – ассоциация с указанным обозначением уже существует в данной схеме.
- DuplicateConstraintBody: "Constraint body %s already exists" – ограничение с указанной структурой уже существует в данной схеме.
- DuplicateInstance: "The association instance %s already exists" – уже существует экземпляр ассоциации с тем же набором значений понятий (экземпляров понятий) в рамках текущего состояния схемы.
- DuplicateTemplate: "The template %s already exists in the templated string %s" – уже существует указанный шаблон в данной строке по шаблону.



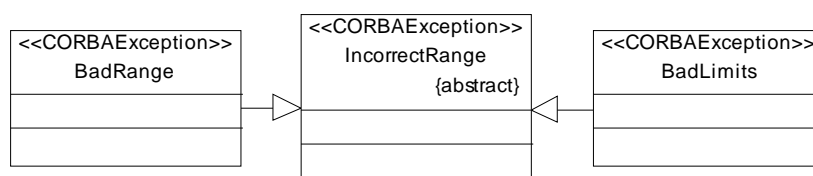
Генерируемые исключения, производные от NotSupportedFeature:

- NotSupportedDomainCascadeDeletion: "Cascade deletion of the domain %s and all its nested objects is not supported" – каскадное удаление предметной области и всех вложенных объектов не поддерживается.
- NotSupportedLocalSchemaCascadeDeletion: "Cascade deletion of the schema %s and all its nested objects is not supported" – каскадное удаление локальной схемы со всеми вложенными объектами не поддерживается.
- NotSupportedConstraintEquiChecking: "Constraint equivalence checking is not supported" – проверка ограничений на эквивалентность не поддерживается.
- NotSupportedTypeChanging: "Type changing is not supported" – изменение типа или параметров типа не поддерживается в данной реализации SCM.
- NotSupportedContentChanging: "Association content changing is not supported" – изменение содержимого ассоциаций не поддерживается в данной реализации SCM.
- NotSupportedCompoundValueMapping: "Mapping of objects/compound values to objects/compound values of other types is not supported" – отображение объектов/составных значений в составные значения других типов не поддерживается в данной реализации SCM.
- NotSupportedDesignationChanging: "Designation changing is not supported: %s" – изменение обозначений не поддерживается.



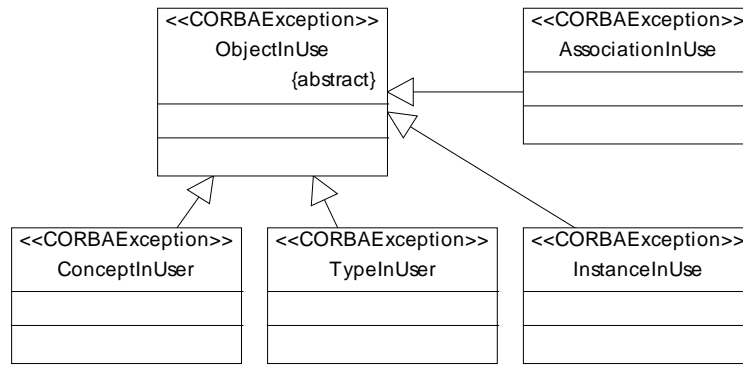
Генерируемые исключения, производные от IncorrectIterator:

- BadElementType: "Iterator element type %s conflicts with the due type %s" – тип элемента итератора не соответствует ожидаемому.
- EmptyIterator: "Iterator is empty" – итератор пуст, тогда как ожидается итератор, содержащий хотя бы один элемент.
- FewElements: "Quantity of iterator elements is less then %s" – итератор содержит недостаточное количество элементов.
- UnorderedIterator: "Iterator is unordered: the element %s incorrectly precedes to the element %s" – итератор не упорядочен, в то время как ожидается упорядоченность элементов в нем.
- RepeatedElements: "Iterator has repeated elements" – итератор содержит повторяющиеся элементы.
- NoNextElement: "There is no a next iterator element" – очередного элемента итератора не существует.
- NoCurrentElement: "There is no a current iterator element" – текущий элемент итератора не выбран.



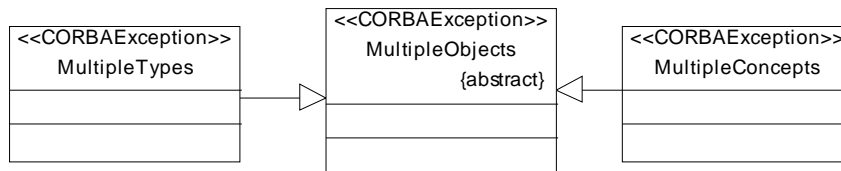
Генерируемые исключения, производные от IncorrectRange:

- BadRange: "Minimum %s of a range is greater than its maximum %s" – минимальная граница диапазона больше его максимальной границы.
- BadLimits: "Minimal limit %s is greater than the maximal limit %s" – минимальный предел больше максимального предела.



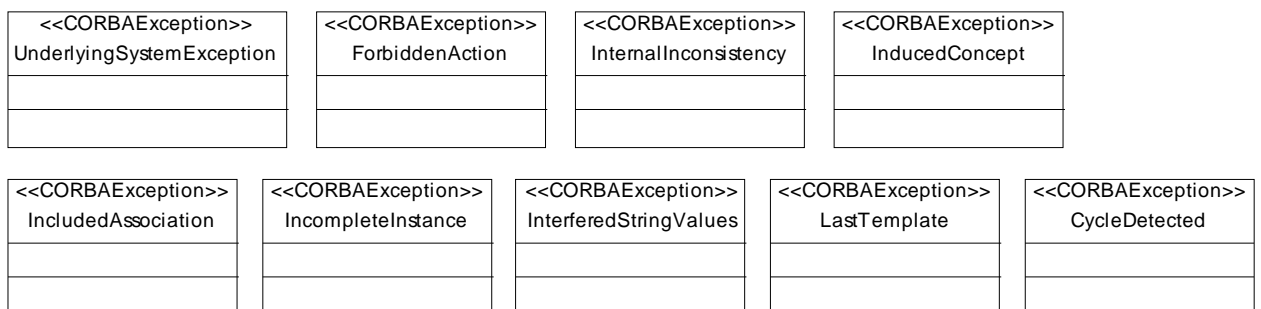
Генерируемые исключения, производные от ObjectInUse:

- TypeInUse: "Type %s is used by %s" – удаляемый тип используется понятием, составным типом или составными значениями в рамках данного дерева предметных областей и поэтому не может быть удален или изменен;
- ConceptInUse: "Concept %s is used in the schema %s" – удаляемое понятие используется для формулирования ассоциаций в хотя бы одной из схем и поэтому не может быть удалено.
- AssociationInUse: "Association %s is used by the constraint %s within the schema %s" – удаляемая ассоциация используется в одном из ограничений данной схемы и поэтому не может быть удалена.
- AssociationInUse: "Association %s of the schema %s is used for data retrieving or modifying" – удаляемая ассоциация используется для доступа к содержимому и поэтому не может быть удалена.
- InstanceInUse: "Association instance %s is already in use now" – экземпляр ассоциации используется в настоящий момент, доступ к нему заблокирован.



Генерируемые исключения, производные от MultipleObjects:

- MultipleTypes: "The type %s has more than one nested types" – данный тип имеет больше одного вложенного типа.
- MultipleConcepts: "The type %s has more than one nested concepts" – данный тип имеет больше одного вложенного понятия.



Генерируемые исключения, не являющиеся производными от каких-либо базовых:

- ForbiddenAction: "%s is not permitted" – действие запрещено;

- `UnderlyingSystemException`: "Underlying system exception: %s" – необрабатываемое исключение, возникшее при работе системы, используемой конкретной реализацией SCM.
- `InternalInconsistency`: "System internal inconsistency has arisen: %s. Inform developers please" – внутренняя противоречивость состояния системы, требуется обращение к разработчикам системы.
- `InducedConcept`: "Concept %s is induced" – данное понятие является порожденным, что приводит к невозможности некоторых действий над ним.
- `IncludedAssociation`: "There can not be an association %s since there exists the association %s within the schema %s" – множество понятий создаваемой ассоциации входит как часть или включает в себя как часть множество понятий уже существующей ассоциации в рамках данной схемы.
- `IncompleteInstance`: "The concept %s has no a specified value when managing content of the association %s" – значения понятий заданы не для всех понятий данной ассоциации в ходе управления ее содержимым.
- `InterferedStringValue`: "Two string values of the same code %s and the same language %s can not be in effect at one moment %s" – два строковых значения одного кода на одном языке не могут действовать одновременно.
- `InterferedStringValue`: "String value of the code %s and the language %s can not be in effect and be canceled at the same moment %s" – строковое значение кода не может действовать и быть отмененным одновременно.
- `LastTemplate`: "The last template %s can not be removed from the templated string %s" – последний шаблон не может быть удален из строки по шаблону.
- `CycleDetected`: "A cycle is detected when deriving the structure %s from the structure %s" – обнаружен цикл в дереве наследования структур.

## 4.2 Транзакционная работа

Управление транзакциями при работе через описанные интерфейсы соответствует спецификации `OMG Transaction Service`<sup>3</sup>. Это означает, что каждый вызов любой операции происходит в рамках некоторой транзакции, которые [транзакции]:

- изолированы друг от друга: в рамках каждой транзакции доступно то состояние объектов, которое было на момент начала транзакции плюс только те изменения, которые выполнены в рамках самой транзакции;
- атомарны: либо все изменения транзакции фиксируются как одно целое, либо все откатываются;
- последовательны: транзакции фиксируются последовательно, одна за другой, причем следующее состояние предыдущей транзакции является исходным для следующей транзакции (выполнение последнего требования может потребовать глобальной блокировки объектов или отката транзакций, конкурирующих за доступ к объектам).

Детализация управления транзакциями при работе через описанные интерфейсы должна осуществляться в рамках спецификаций, касающихся реализации данных интерфейсов на конкретных платформах.

## 4.3 Параллельная работа

Все описанные интерфейсы допускают параллельную работу из разных нитей. При этом происходит блокировка (синхронизация) нитей:

- по транзакциям (в каждый момент времени только одна нить может работать с каждой из транзакций);

<sup>3</sup> `OMG Transaction Service Specification V1.4` (<http://www.omg.org/cgi-bin/doc?formal/2003-09-02>), 2003.

- по разделяемым внутренним объектам (если один объект модифицируется в рамках нескольких транзакций, то последующие транзакции либо ожидают завершения первой модифицирующей транзакции, либо откатываются; для разведения транзакций может использоваться сохранение копий объектов в рамках каждой из конкурирующих транзакций).

#### 4.4 Управление доступом

Управление доступом при работе через описанные интерфейсы соответствует спецификации OMG Security Service<sup>4</sup>. Детализация управления доступом должна осуществляться в рамках спецификаций, касающихся реализации данных интерфейсов на конкретных платформах.

#### 4.5 Многоязыковая и терминологическая поддержка

Язык и терминология, используемая в рамках конкретных вызовов интерфейсных операций, определяется параметрами текущей нити, которые пользователь настраивает перед использованием соответствующих интерфейсных операций в соответствии со спецификацией MLANG-SPEC.

#### 4.6 Спецификация SCM в виде IDL

Ниже приводится спецификация SCM в виде IDL, полностью соответствующая вышеописанной объектной спецификации, формализованной при помощи UML.

```
#if !defined(__SCMSPEC_Model_SCM_idl)
#define __SCMSPEC_Model_SCM_idl

#include <Model/SCM/CL.idl>

module Model
{
    module SCM
    {
        exception DuplicateObject;
        exception NoObject;
        exception IncludedAssociation;
        exception ConceptInUser;
        exception DuplicateInstance;
        exception NoInstance;
        exception IncompleteInstance;
        exception NotSupportedFeature;
        exception UnderlyingSystemException;
        exception NoBaseTypeConstant;
        exception NoDomain;
        exception NoConcept;
        exception NoDescribedConcept;
        exception NoLocalSchema;
        exception DuplicateDomain;
        exception DuplicateDescribedConcept;
        exception DuplicateLocalSchema;
        exception NotSupportedDomainCascadeDeletion;
        exception NotSupportedLocalSchemaCascadeDeletion;
        exception NoType;
        exception DuplicateType;
        exception NoRangeTypeConstant;
        exception IncorrectIterator;
        exception EmptyIterator;
    }
}

```

<sup>4</sup> OMG Security Service Specification V1.8 (<http://www.omg.org/cgi-bin/doc?formal/2002-03-11>), 2002.

```

exception BadElementType;
exception BadRange;
exception BadLimits;
exception IncorrectRange;
exception TypeInUse;
exception ObjectInUse;
exception UnorderedIterator;
exception FewElements;
exception ForbiddenAction;
exception NoLanguage;
exception NoAssociation;
exception NoDescribedAssociation;
exception DuplicateAssociation;
exception AssociationInUse;
exception NotSupportedConstraintEquiChecking;
exception NoDescribedConstraint;
exception DuplicateConstraintBody;
exception InducedConcept;
exception NotSupportedTypeChanging;
exception NotSupportedContentChanging;
exception InternalInconsistency;
exception NoMapping;
exception NotSupportedCompoundValueMapping;
exception NoValue;
exception InstanceInUse;
exception NotSupportedDesignationChanging;
exception NoString;
exception InterferedStringValue;
exception DuplicateTemplate;
exception NoTemplate;
exception LastTemplate;
exception MultipleObjects;
exception MultipleTypes;
exception MultipleConcepts;
exception RepeatedElements;
exception NoNextElement;
exception NoCurrentElement;
exception NoStructure;
exception CycleDetected;
interface Domain;
interface Iterator;
interface Schema;
interface SchemaAssociation;
interface Concept;
interface Element;
interface AssociationChanger;
interface CompoundValue;
interface Type;
interface MetatypeConstants;
interface BaseTypeConstants;
interface Dictionary;
interface BaseType;
interface NumberRange;
interface RangeTypeConstants;
interface IntegerRange;
interface DictionaryRecord;
interface TemplatedString;
interface ValueTypeConstants;
interface ContinuousRange;
interface DiscreteRange;
interface DateRange;
interface CompoundValueCollection;
interface CompoundType;
interface LimitedCompoundType;

```

```

interface SimpleValueCollection;
interface RangeCollection;
interface AssociationIterator;
interface TimeRange;
interface GeneralIterator;

exception DuplicateObject {
};

exception NoObject {
};

exception IncludedAssociation {
};

exception ConceptInUser {
};

exception DuplicateInstance {
};

exception NoInstance {
};

exception IncompleteInstance {
};

exception NotSupportedFeature {
};

exception UnderlyingSystemException {
};

exception NoBaseTypeConstant {
};

exception NoDomain {
};

exception NoConcept {
};

exception NoDescribedConcept {
};

exception NoLocalSchema {
};

exception DuplicateDomain {
};

exception DuplicateDescribedConcept {
};

exception DuplicateLocalSchema {
};

exception NotSupportedDomainCascadeDeletion {
};

exception NotSupportedLocalSchemaCascadeDeletion {
};

exception NoType {
};

```

```
};

exception DuplicateType {
};

exception NoRangeTypeConstant {
};

exception IncorrectIterator {
};

exception EmptyIterator {
};

exception BadElementType {
};

exception BadRange {
};

exception BadLimits {
};

exception IncorrectRange {
};

exception TypeInUser {
};

exception ObjectInUse {
};

exception UnorderedIterator {
};

exception FewElements {
};

exception ForbiddenAction {
};

exception NoLanguage {
};

exception NoAssociation {
};

exception NoDescribedAssociation {
};

exception DuplicateAssociation {
};

exception AssociationInUse {
};

exception NotSupportedConstraintEquiChecking {
};

exception NoDescribedConstraint {
};

exception DuplicateConstraintBody {
};
```

```
exception InducedConcept {
};

exception NotSupportedTypeChanging {
};

exception NotSupportedContentChanging {
};

exception InternalInconsistency {
};

exception NoMapping {
};

exception NotSupportedCompoundValueMapping {
};

exception NoValue {
};

exception InstanceInUse {
};

exception NotSupportedDesignationChanging {
};

exception NoString {
};

exception InterferedStringValue {
};

exception DuplicateTemplate {
};

exception NoTemplate {
};

exception LastTemplate {
};

exception MultipleObjects {
};

exception MultipleTypes {
};

exception MultipleConcepts {
};

exception RepeatedElements {
};

exception NoNextElement {
};

exception NoCurrentElement {
};

exception NoStructure {
};
```

```

exception CycleDetected {
};

interface Domain {
    Domain getRootDomain();
    Schema getUnitedSchema();
    string getDesignation();
    string getFullDesignation();
    Domain getMainDomain();
    Iterator getNestedDomains();
    Domain getNestedDomain(in string designation);
    Domain createNestedDomain(in string designation);
    void removeNestedDomain(in Domain domain);
    boolean existsConcept(in string designation);
    Concept getConcept(in string designation);
    Concept getDescribedConcept(in string designation);
    Iterator getDescribedConcepts();
    Concept createDescribedConcept(in string designation, in Type type);
    void removeDescribedConcept(in Concept concept);
    Schema getLocalSchema(in string designation);
    Iterator getLocalSchemas();
    Schema createLocalSchema(in string designation);
    void removeLocalSchema(in Schema schema);
    BaseType getBaseType(in short baseType);
    Type getType(in string designation);
    Dictionary createDictionary(in string designation);
    NumberRange createNumberRange(in string designation, in double min,
        in double max, in short rangeType);
    IntegerRange createIntegerRange(in string designation, in long min,
        in long max);
    DateRange createDateRange(in string designation, in long min, in
        long max, in short rangeType);
    IntegerRange createCodeRange(in string designation, in long min, in
        long max);
    IntegerRange createLimitedLengthString(in string designation, in
        long min, in long max);
    TemplatedString createTemplatedString(in string designation, in
        GeneralIterator templates);
    SimpleValueCollection createNumberCollection(in string designation,
        in GeneralIterator values);
    SimpleValueCollection createIntegerCollection(in string designation,
        in GeneralIterator values);
    SimpleValueCollection createDateCollection(in string designation, in
        GeneralIterator values);
    SimpleValueCollection createStringCollection(in string designation,
        in GeneralIterator values);
    SimpleValueCollection createCodeCollection(in string designation, in
        GeneralIterator values);
    RangeCollection createNumberRangeCollection(in string designation,
        in Iterator ranges);
    RangeCollection createIntegerRangeCollection(in string designation,
        in Iterator ranges);
    RangeCollection createDataRangeCollection(in string designation, in
        Iterator ranges);
    RangeCollection createCodeRangeCollection(in string designation, in
        Iterator ranges);
    CompoundType createStructure(in string designation, in Iterator
        concepts, in CompoundType baseStructure);
    CompoundType createVector(in string designation, in Concept
        concept);
    CompoundType createSet(in string designation, in Concept concept);
    LimitedCompoundType createLimitedVector(in string designation, in
        Concept concept, in unsigned long min, in unsigned long max);

```

```

    LimitedCompoundType createLimitedSet(in string designation, in
        Concept concept, in unsigned long min, in unsigned long max);
    CompoundValueCollection createStructureValueCollection(in string
        designation, in CompoundType sourceType, in Iterator values);
    CompoundValueCollection createVectorValueCollection(in string
        designation, in CompoundType sourceType, in Iterator values);
    CompoundValueCollection createSetValueCollection(in string
        designation, in CompoundType sourceType, in Iterator values);
    void removeType(in Type type);

};

interface Iterator {
    boolean hasNext();
    Object next();
};

interface Schema {
    string getDesignation();
    void setDesignation(in string designation);
    Domain getDomain();
    string getFullDesignation();
    boolean existsSchemaAssociation(in Iterator concepts);
    SchemaAssociation getSchemaAssociation(in Iterator concepts);
    SchemaAssociation getDescribedSchemaAssociation(in Iterator
        concepts);
    Iterator getDescribedSchemaAssociations();
    Iterator getDescribedSchemaAssociations(in Concept concept);
    Iterator getDescribedSchemaAssociations(in ::Model::SCM::Constraint
        constraint);
    Iterator getSchemaAssociations(in ::Model::SCM::Constraint
        constraint);
    SchemaAssociation createDescribedSchemaAssociation(in Iterator
        concepts);
    void removeDescribedSchemaAssociation(in SchemaAssociation
        association);
    boolean fulfillsConstraint(in ::Model::SCM::ConstraintBody
        constraintBody);
    ::Model::SCM::Constraint getDescribedConstraint(in string
        designation);
    ::Model::SCM::Constraint getDescribedConstraint(in
        ::Model::SCM::ConstraintBody constraintBody);
    Iterator getDescribedConstraints();
    Iterator getDescribedConstraints(in SchemaAssociation association);
    Iterator getDescribedConstraints(in Concept concept);
    ::Model::SCM::Constraint createDescribedConstraint(in
        ::Model::SCM::ConstraintBody constraintBody);
    void removeDescribedConstraint(in ::Model::SCM::Constraint
        constraint);
};

interface SchemaAssociation : Element {
    Schema getSchema();
    Iterator getConcepts();
    AssociationChanger createAssociationChanger();
    AssociationIterator createAssociationIterator();
    long getConceptQuantity();
    long getIndex(in Concept concept);
    string synthesizeDesignation();
};

```

```

interface Concept : Element {
    Domain getDomain();
    string getDesignation();
    void setDesignation(in string designation);
    string getFullDesignation();
    Type getType();
    void setType(in Type type);
};

interface Element {
    boolean isDescribed();
    boolean isInduced();
};

interface AssociationChanger {
    SchemaAssociation getSchemaAssociation();
    void setValue(in long index, in Object value);
    void setValue(in long index, in CompoundValue value);
    void setValue(in long index, in float value);
    void setValue(in long index, in double value);
    void setValue(in long index, in char value);
    void setValue(in long index, in short value);
    void setValue(in long index, in long value);
    void setValue(in long index, in long long value);
    void setValue(in long index, in unsigned char value);
    void setValue(in long index, in unsigned short value);
    void setValue(in long index, in unsigned long value);
    void setValue(in long index, in unsigned long long value);
    void setValue(in long index, in boolean value);
    void setValue(in long index, in string value);
    boolean existsAssociationInstance();
    void createAssociationInstance(in boolean wait);
    void removeAssociationInstance(in boolean wait);
    void clearValues();
};

interface CompoundValue {
    CompoundType getType();
    Object asObject();
    Object asObject(in long index);
    CompoundValue asCompoundValue(in long index);
    float asFloat(in long index);
    double asDouble(in long index);
    char asByte(in long index);
    short asShort(in long index);
    long asLong(in long index);
    long long asLongLong(in long index);
    char asUnsignedByte(in long index);
    unsigned short asUnsignedShort(in long index);
    unsigned long asUnsignedLong(in long index);
    unsigned long long asUnsignedLongLong(in long index);
    long asTime(in long index);
    boolean asBoolean(in long index);
    string asString(in long index);
};

interface Type {
    Domain getDomain();
    string getDesignation();
    void setDesignation(in string designation);
};

```

```

    string getFullDesignation();
    short getMetatype();
    short getValueType();
};

interface MetatypeConstants {
    attribute short BASE;
    attribute short DICTIONARY;
    attribute short NUMBER_RANGE;
    attribute short INTEGER_RANGE;
    attribute short DATE_RANGE;
    attribute short CODE_RANGE;
    attribute short LIMITED_LENGTH_STRING;
    attribute short TEMPLATED_STRING;
    attribute short NUMBER_COLLECTION;
    attribute short INTEGER_COLLECTION;
    attribute short DATE_COLLECTION;
    attribute short STRING_COLLECTION;
    attribute short CODE_COLLECTION;
    attribute short NUMBER_RANGE_COLLECTION;
    attribute short INTEGER_RANGE_COLLECTION;
    attribute short DATE_RANGE_COLLECTION;
    attribute short CODE_RANGE_COLLECTION;
    attribute short STRUCTURE;
    attribute short VECTOR;
    attribute short SET;
    attribute short LIMITED_VECTOR;
    attribute short LIMITED_SET;
    attribute short STRUCTURE_VALUE_COLLECTION;
    attribute short VECTOR_VALUE_COLLECTION;
    attribute short SET_VALUE_COLLECTION;
};

interface BaseTypeConstants {
    attribute short NUMBER;
    attribute short INTEGER;
    attribute short BOOLEAN;
    attribute short DATE;
    attribute short STRING;
};

interface Dictionary : Type {
    string getString(in long code, in long moment);
    void setString(in long code, in long moment, in string str);
    void removeString(in long code);
    void cancelCode(in long code, in long moment);
    Iterator getDictionaryRecords();
    Iterator getDictionaryRecords(in long moment);
};

interface BaseType : Type {
    short getBaseType();
};

interface NumberRange : Type {
    double getMin();
    double getMax();
    void setRange(in double min, in double max);
    short getRangeType();
};

```

```

    void setRangeType(in short rangeType);
};

interface RangeTypeConstants {
    attribute short IN_IN;
    attribute short EX_EX;
    attribute short IN_EX;
    attribute short EX_IN;
};

interface IntegerRange : Type {
    long getMin();
    long getMax();
    void setRange(in long min, in long max);
};

interface DictionaryRecord {
    long getCode();
    string getValue();
    long getLanguage();
    long getEndMoment();
};

interface TemplatedString : Type {
    Iterator getTemplates();
    void addTemplate(in string str);
    void removeTemplate(in string str);
};

interface ValueTypeConstants {
    attribute short FLOAT;
    attribute short DOUBLE;
    attribute short BYTE;
    attribute short SHORT;
    attribute short LONG;
    attribute short LONG_LONG;
    attribute short UNSIGNED_BYTE;
    attribute short UNSIGNED_SHORT;
    attribute short UNSIGNED_LONG;
    attribute short UNSIGNED_LONG_LONG;
    attribute short TIME;
    attribute short STRING;
    attribute short BOOLEAN;
    attribute short STRUCTURE;
    attribute short VECTOR;
    attribute short SET;
};

interface ContinuousRange {
    double getMin();
    double getMax();
    short getRangeType();
};

interface DiscreteRange {
    long getMin();
    long getMax();
};

```

```

};

interface DateRange : Type {
    long getMin();
    long getMax();
    void setRange(in long min, in long max);
    short getRangeType();
    void setRangeType(in short rangeType);
};

interface CompoundValueCollection : CompoundType {
    Iterator getCompoundValues();
    Iterator getObjects();
    void setValues(in Iterator values);
};

interface CompoundType : Type {
    Type getNestedType();
    Iterator getNestedTypes();
    Concept getNestedConcept();
    void setNestedConcept(in Concept concept);
    Iterator getNestedConcepts();
    void setNestedConcepts(in Iterator concepts);
    CompoundType getBaseStructure();
    void setBaseStructure(in CompoundType compoundType);
    Iterator getDerivedStructures();
    CompoundValue createValue(in GeneralIterator nestedValues);
    long getQuantity();
    Concept getConcept(in long index);
};

interface LimitedCompoundType : CompoundType {
    unsigned long getMin();
    unsigned long getMax();
    void setRange(in unsigned long min, in unsigned long max);
};

interface SimpleValueCollection : Type {
    GeneralIterator getValues();
    void setValues(in GeneralIterator values);
};

interface RangeCollection : Type {
    Iterator getRanges();
    void setRanges(in Iterator ranges);
};

interface AssociationIterator {
    SchemaAssociation getSchemaAssociation();
    boolean hasNext();
    void next();
    Object asObject(in long index);
    CompoundValue asCompoundValue(in long index);
    float asFloat(in long index);
    double asDouble(in long index);
    char asByte(in long index);
    short asShort(in long index);
};

```

```

    long asLong(in long index);
    long long asLongLong(in long index);
    char asUnsignedByte(in long index);
    unsigned short asUnsignedShort(in long index);
    unsigned long asUnsignedLong(in long index);
    unsigned long long asUnsignedLongLong(in long index);
    long asTime(in long index);
    boolean asBoolean(in long index);
    string asString(in long index);

};

interface TimeRange {
    long getMin();
    long getMax();
    short getRangeType();
};

interface GeneralIterator {
    boolean hasNext();
    void next();
    short getValueType();
    Object asObject();
    float asFloat();
    double asDouble();
    char asByte();
    short asShort();
    long asLong();
    long long asLongLong();
    char asUnsignedByte();
    unsigned short asUnsignedShort();
    unsigned long asUnsignedLong();
    unsigned long long asUnsignedLongLong();
    long asTime();
    boolean asBoolean();
    string asString();
};
};

#endif // __SCMSPEC_Model_SCM_idl

```