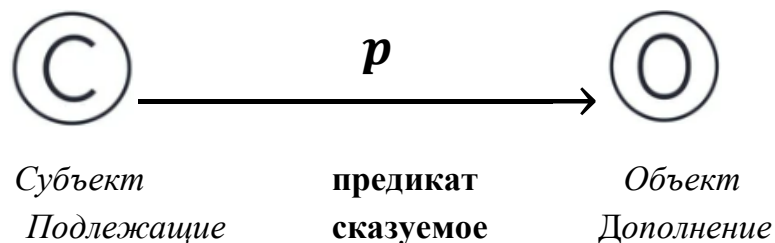


Лабораторная работа 1

Создание онтологий в Protégé

Понятие об онтологиях

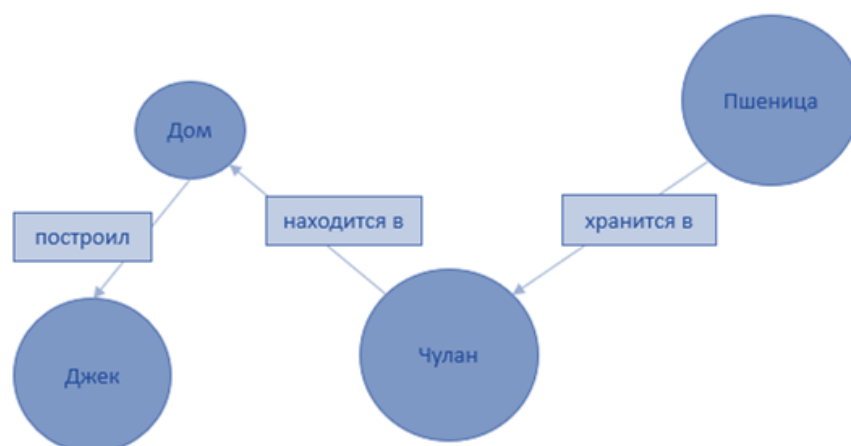
В математическом основании онтологии лежит так называемая дескриптивная логика (раздел математики), предполагающая, что любая информация, высказанная на естественном языке, может быть представлена без потери смысла в виде цепочки утверждений – *триплетов*. Каждый из триплетов состоит из *субъекта*, *предиката* и *объекта* (подлежащие, сказуемое, дополнение).



Любое сложное предложение можно свести к цепочки простых предложений, состоящих подлежащего, сказуемого и дополнения. Отрывок из стихотворения «Дом, который построил Джек...» (пер. С.Я. Маршака):

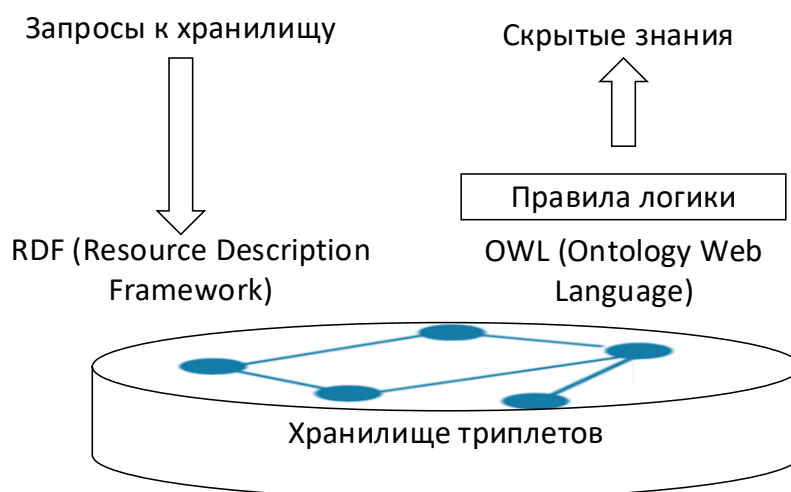
Вот дом,
 Который построил Джек.
 А это пшеница,
 Которая в тёмном чулане хранится
 В доме,
 Который построил Джек.

Описанные в стихах отношения между различными сущностями можно представить в виде онтологии.



Онтология представляется в виде графа, вершины которого это сущности, а ребра – отношения между сущностями. Считается, что любое утверждение на естественном языке можно представить в виде простых предложений, из которых можно извлечь сущности и отношения между ними.

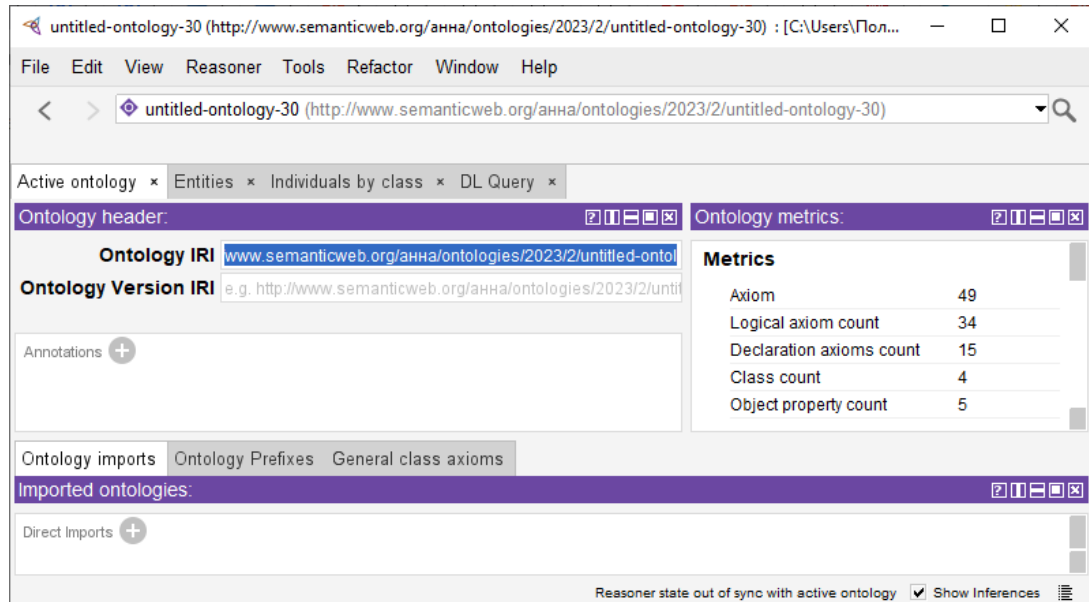
Есть два основных языка: RDF (Resource Description Framework) и OWL (Ontology Web Language). RDF позволяет записывать триплеты и накапливать их в определенном хранилище, имеющий текстовый формат. К этому хранилищу можно делать запрос. OWL позволяет дополнительно описывать логические правила над данными. Онтологии (в отличие от обычных баз данных) позволяют находить скрытые знания. Обычные хорошо подходят для поиска конкретной информации, а базы знаний нужны там, где надо выявлять новые знания, например, в системах поддержки принятия решений (экспертных системах).



Сила онтологии проявляется в том случае, если подробно и качественно описаны взаимосвязи между ее элементами, с использованием математического аппарата дескриптивной логики. Например, для отношений можно задать их свойства (функциональное, транзитивное, рефлексивное). И тогда можно автоматически из онтологии извлекать факты, этот процесс называется ризонинг (reasoning), есть типовые алгоритмы ризонинга, основанные на графах. Возможные применения: уточнение характеристик объекта и выделение из набора похожих объектов уникального, поиск похожих объектов, «понимание текста» и отнесение текста к определенному классу, помощь в NLP задачах (NER, Relation Extraction), анализ корневых причин, выявление паттернов в данных. Наиболее популярный редактор онтологий, поддерживающий ризонинг, это Protégé. Существуют и другие инструменты, например: IBM Watson, Wolfram Alpha.

Основы работы в Protégé

После запуска программы мы видим вот такое стартовое окно.



Для каждого проекта существует уникальный идентификатор IRI (Internationalized Resource Identifier). Protégé позволяет записывать триплеты, то есть тройки вида «субъект-предикат-объект». Раздел **Entities (Сущности)** позволяет описывать субъекты и объекты. Важные вкладки в этом разделе:

- **Экземпляры классов (Individuals)** это объекты классов, как в объектно-ориентированном программировании. Например, класс «Сервер», объект «prod-serv-002».
- **Свойства (object properties или data properties)**, похоже на свойства классов в ООП, однако в онтологии свойства имеют независимую природу, свойство не является неотделимым от класса (как в ООП).

Для предикатов можно задавать разные свойства, например: функциональный; обратный; транзитивный. По описанной онтологии можно применять ризонер, который дает предложения по найденным фактам (можно принять или отказаться).

Онтологию можно сохранить во внешний файл, в формате owl, который можно открыть в текстовом редакторе и увидеть, что это xml.

Эти и другие понятия разберем на конкретном примере.

Предметная область

Создадим онтологию клиентской базы некой компании. Сегментация — это выбор части клиентской базы по определенным признакам.

Сегментировать клиентов можно по-разному, например:

- По жизненному циклу: на каком этапе воронки находится клиент.
- По личным данным клиента: полу, возрасту, географии.
- С помощью RFM-анализа: по давности, частоте и сумме покупок.

- По поведению: как часто потенциальный покупатель заходит на сайт, открывает рассылки, какие товары или категории смотрит.

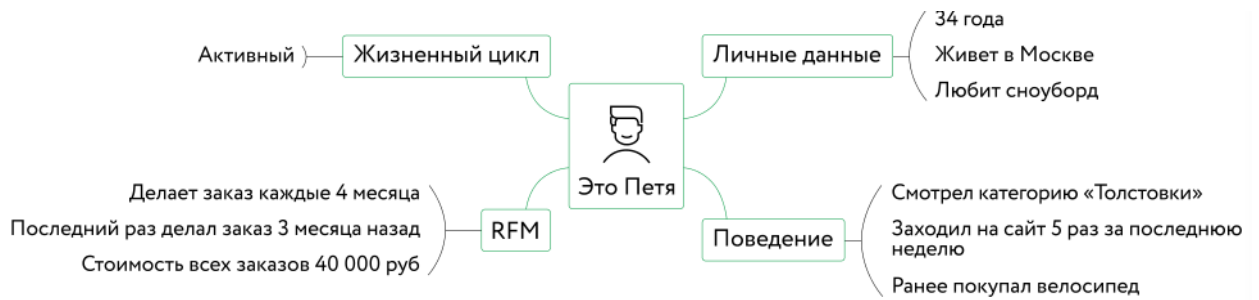


Рис. 1. Характеристика клиента с помощью различных видов сегментации

Остановим свое внимание на сегментации по жизненному циклу. Взаимодействие клиента и компании начинается с привлечения потенциального покупателя. Далее клиент знакомится с брендом, выбирает товар и совершает целевое действие — заказ. После такого опыта взаимодействия с компанией он либо переходит на новый цикл «выбор товара → целевое действие», либо уходит в отток. Таким образом, клиентов можно разделить на 3 группы: новички, активные, отток.

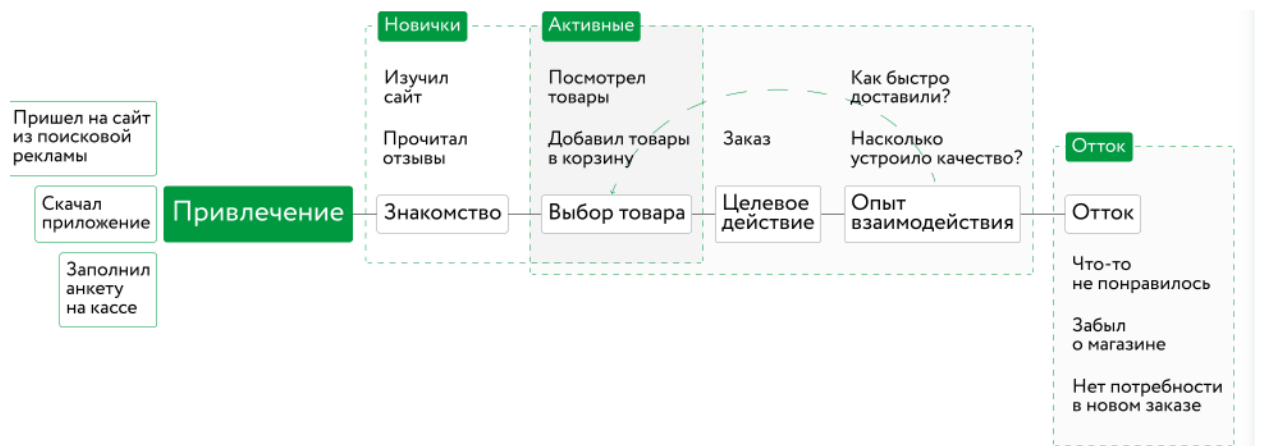


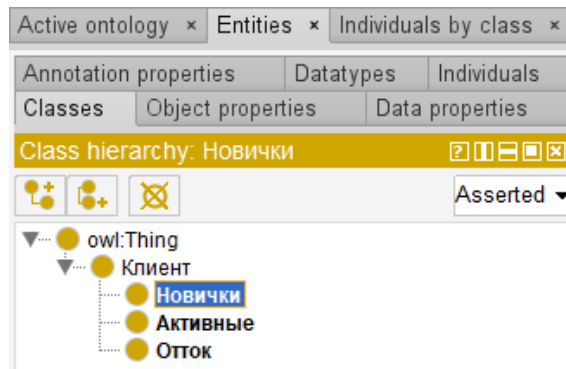



Рис. 2. Пример жизненного цикла клиента интернет магазина

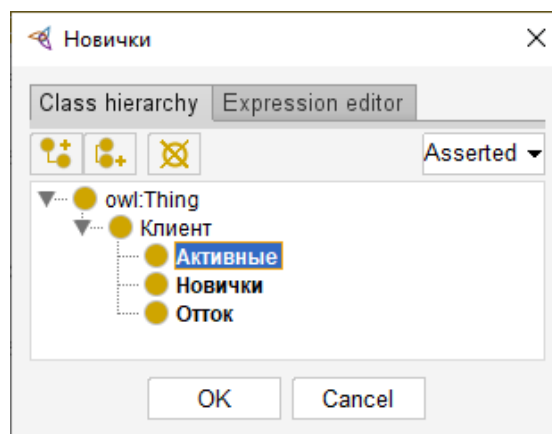
1. Создание простой онтологии

Выберите вкладку **Classes**. На вкладке **Classes** верхнем уровнем является предопределённый класс *owl:Thing*. Нажмите на кнопку **Add subclass**  и создайте класс *Клиент*. В поле **Name** диалогового окна **Create a new Class** введите имя класса *Клиент*.

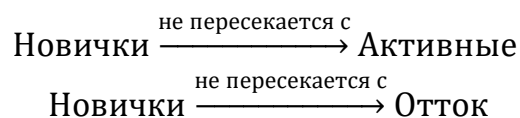
Класс *Клиент* разделите на подклассы. С помощью кнопки **Add subclass**  создайте подклассы в классе *Клиент*: *Отток*, *Активные*, *Новички*. В результате получилась следующая иерархия классов клиентов компании:



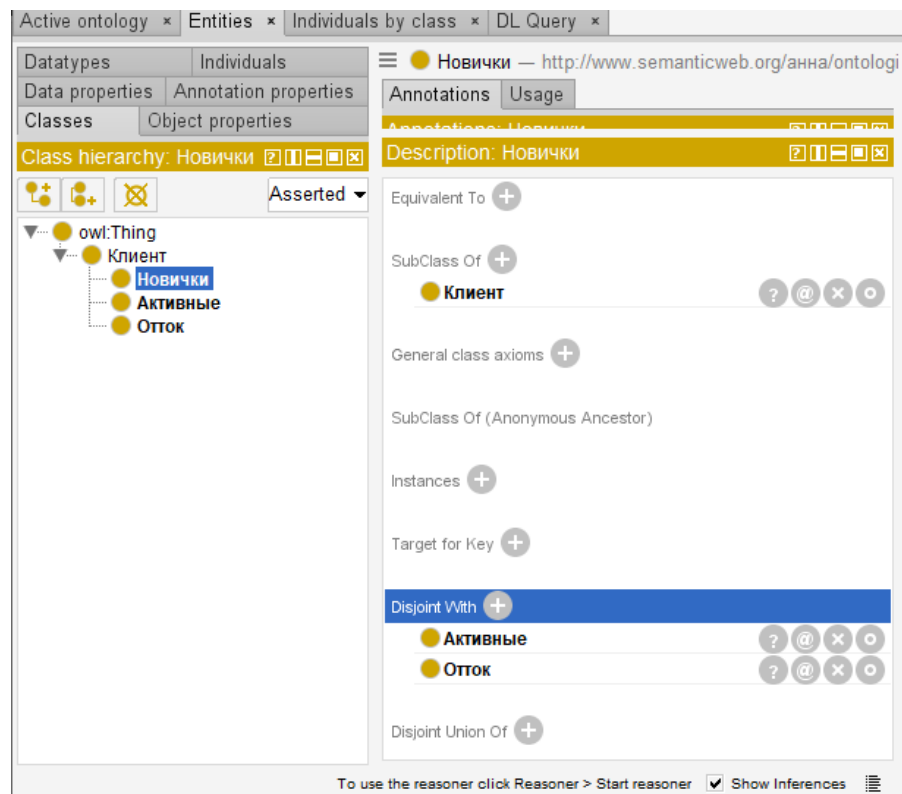
Будем считать, что заданные классы клиентов являются не пересекающимися (например, новичок не может быть одновременно активным клиентом и т.п.). По умолчанию в Protégé заложено, что классы могут пересекаться. Поэтому необходимо явно указать, что классы клиентов не пересекаются. В разделе **Description** для этого есть предопределенный предикат **Disjoint With (не пересекается с)**. В разделе **Class hierarchy** поставьте курсор мыши на *Новички*. В разделе **Description** около **Disjoint With** нажмите на  и в окно *Новички* последовательно введите утверждения, что подкласс *Новички* не пересекается с подклассом *Активные* и *Отток*.



По сути определили следующие триплеты:



В результате окно Protégé будет выглядеть следующим образом:

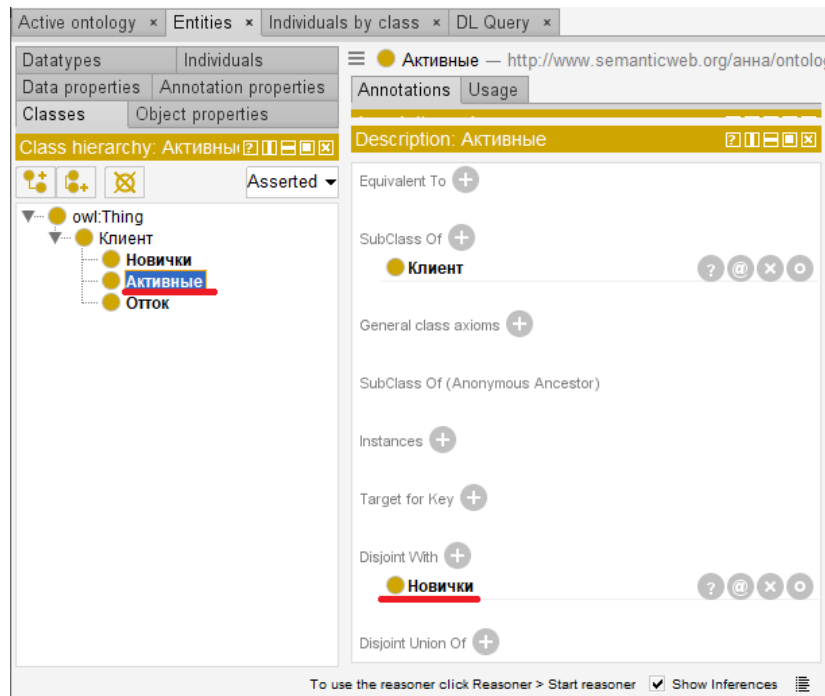


Поставьте курсор мыши на подкласс *Активные*. В разделе **Description** в **Disjoint With** часть информации уже есть, то есть если задано утверждение:

Новички $\xrightarrow{\text{не пересекается с}}$ Активные

то и утверждение «наоборот» тоже работает и уже появилось в **Description** в **Disjoint With** для подкласса *Активные*:

Активные $\xrightarrow{\text{не пересекается с}}$ Новички



Поэтому для подкласса *Активные* необходимо ввести оставшееся утверждение:

Активные $\xrightarrow{\text{не пересекается с}}$ Отток.

Для последнего класса *Отток* вся информация будет уже автоматически введена.

Итак, если в онтологии классы задуманы как не пересекающиеся, этот факт в явной форме нужно сохранить. При чем все высказывания (утверждения), которые были построены, имеют форму триплета, еще раз пример:


Новички $\xrightarrow{\text{не пересекается с}}$ Активные

В данном примере сформированы также еще следующие триплеты:

Клиент $\xrightarrow{\text{имеет подкласс}}$ Новички
 Клиент $\xrightarrow{\text{имеет подкласс}}$ Активные
 Клиент $\xrightarrow{\text{имеет подкласс}}$ Отток

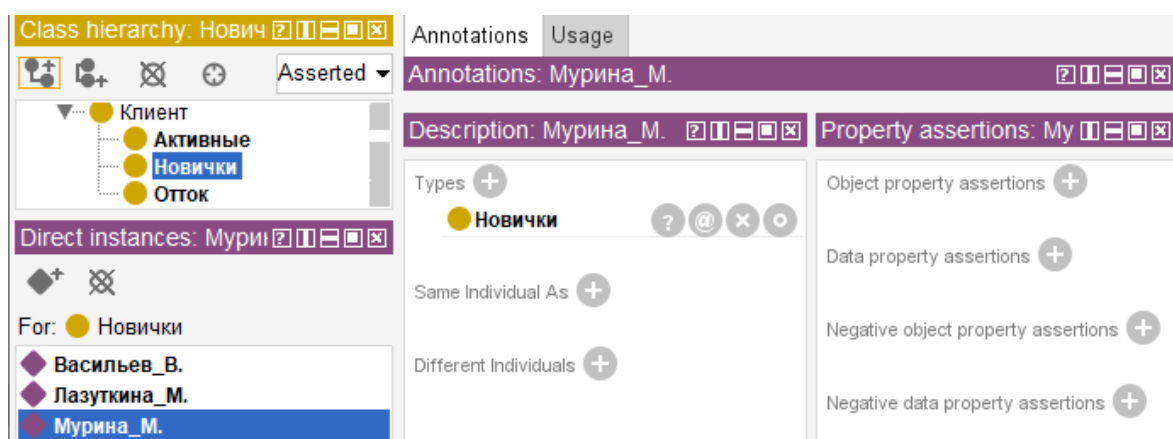
Пока не будем усложнять иерархию. Кроме классов субъектами и объектами могут выступать также экземпляры классов. Их можно определить на вкладке **Entities** → **Individuals** или непосредственно на вкладке **Individuals by class**, на которой видны и классы, и экземпляры. Класс (*Клиент*) является абстрактным понятием. Если нужно внести информацию о конкретном клиенте, как о ресурсе

компании, в базу знаний, то это будет экземпляр класса. При чем ресурсами выступают как классы, так и экземпляры классов.

На вкладке **Classes** в разделе **Class hierarchy** поставьте курсор мыши на подкласс *Новички*. На вкладке **Individuals by class** в разделе **Direct Instances** нажмите на кнопку **Add Individual** . В открывшемся окне **Create a new Named individual** введите фамилию и имя клиента – Васильев В. Сразу видим, что в разделе **Description** в **Types** *Васильев В.* определился как новичок. То есть снова создан триплет:

Васильев_В. ^{имеет тип} → Новички

В этом же подклассе (*Новички*) создайте еще клиенток Мурину_М. и Лазуткину_М.




2. Reasoner. Поиск скрытых знаний


До этого момента в создаваемой онтологии использовали предопределенные предикаты (**не пересекается с**, **имеет подкласс**, **имеет тип**). Но в Protégé есть возможность создавать свои предикаты. Для этого есть вкладки **Entities** → **Object properties** и **Entities** → **Data properties**. Предикаты в Protégé называются свойствами (**properties**). Свойства в онтологии похожи на свойства класса в ООП, но ООП свойства привязаны к классу, в Protégé такой привязки нет. Свойства существуют сами по себе – это так называемая свойствоцентричная модель.

Свойства-отношения (Object properties) - те свойства, у которых объектом может выступать готовый класс или экземпляр. Определяют некоторые отношения между двумя индивидами (экземплярами классов).

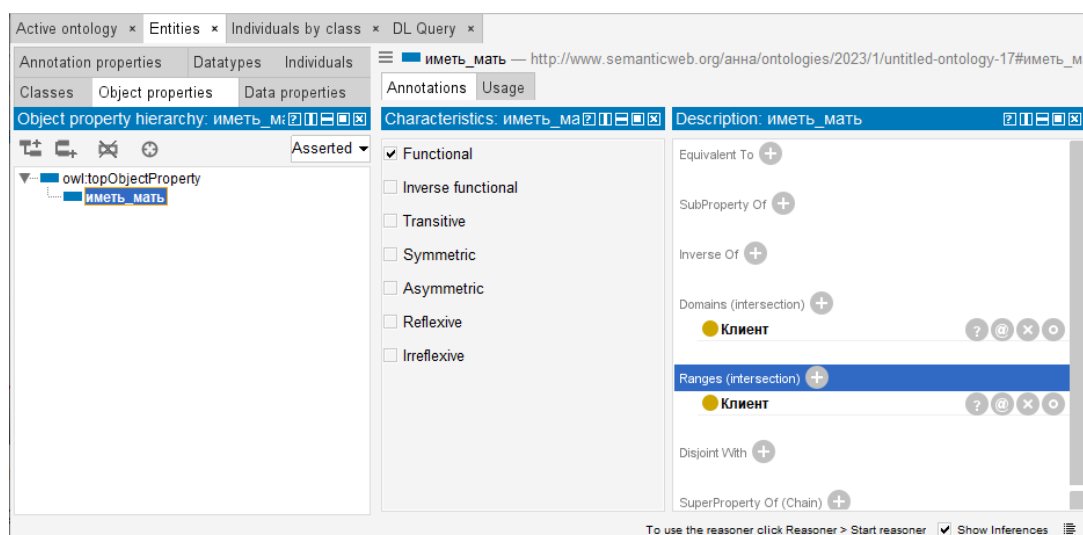
Свойства-данных (Data properties) - объектом может выступать простой примитивный тип данных (число, строка, дата и т.п.). определяют некоторые фактические характеристики индивидов (экземпляров классов)

2.1. Свойства-отношения (Object properties)


Рассмотрим вкладку **Object properties**, здесь тоже есть верхнее свойство *topObjectproperty* и все остальные свойства являются его подсвойствами. Создайте свойство *иметь_мать*. Давайте обозначим, что свойство *иметь_мать* может действовать из класса *Клиент* в класс *Клиент*. То есть матерью клиента является экземпляр класса *Клиент*. Для этого на вкладке **Object properties** выберите свойство *иметь_мать*, а в разделе **Description** обратите внимание на 2 важных предиката **Domains** и **Ranges**. Щелкните мышью на кнопке  предиката **Domains**, убедитесь, что в окне *иметь_мать* выбрана вкладка **Class hierarchy** и укажите, что для свойства *иметь_мать* субъекты берутся из класса *Клиент*.

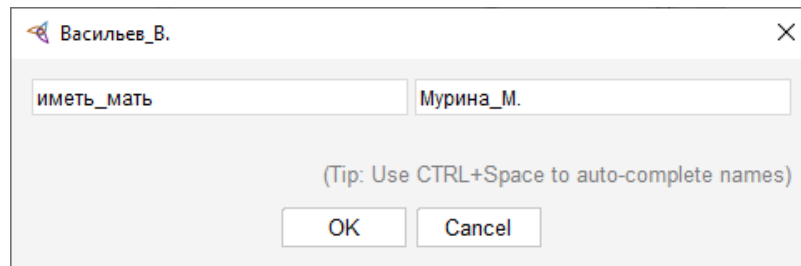
Щелкните мышью на кнопке  предиката **Ranges** (область допустимых значений), откуда берутся объекты данного свойства *иметь_мать*. Здесь тоже укажите, что субъекты берутся из класса *Клиент*. Свойство не является не отделимым свойством какого-то одного класса – в домен можно привязать набор классов, например если рассматривать не только клиентов, но и поставщиков компании. Для свойства важно указать **Domains** (откуда действует свойство) и **Ranges** (куда действуют свойство).

Добавим небольшой элемент логики. Свойство *иметь_мать* является функциональным свойством, так как у человека мать может быть только одна (при этом детей может быть несколько). В разделе **Characteristics**, напротив **Functional** поставьте галочку. Результаты настройки свойства *иметь_мать* приведены на рисунке ниже:

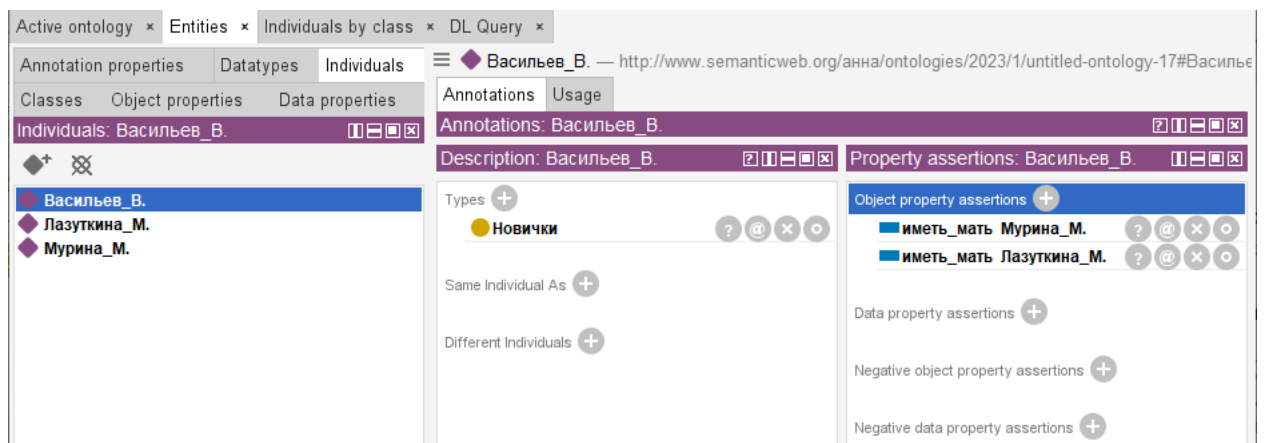


На вкладке **Individuals by class** в разделе **Direct Instances** и создайте факт о том, что матерью Васильева В. является Мурина М. Для этого в разделе **Direct**

Instances выберите *Васильев_В.*, а в разделе **Property assertions** щелкните мышью на кнопке  **Object property assertions** и вручную введите информацию:



Сохраните этот факт. Теперь сохраните еще один факт: Васильев В. имеет мать Лазуткину М. Так не может быть ни вроде бы с житейской точки зрения, ни с позиции Protégé, так как было отмечено, что свойство *иметь_мать* функционально (**Functional**). Обратите внимание, что Protégé позволяет сохранить оба этих факта.

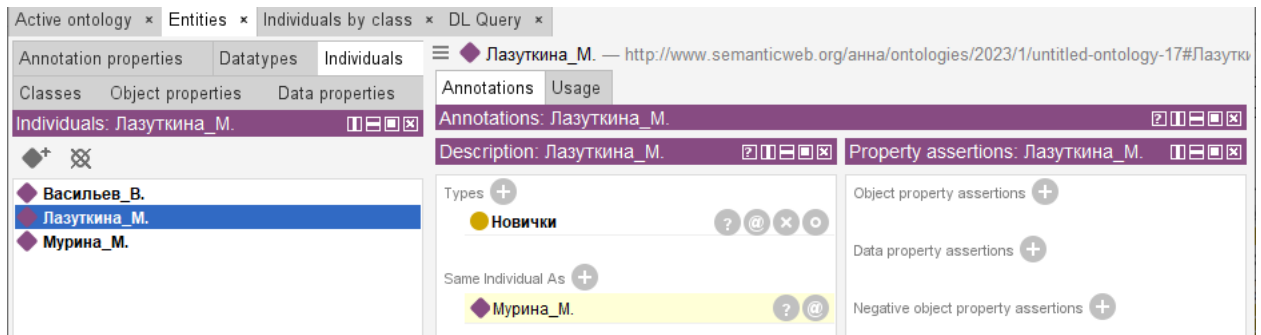


Подключаем **Reasoner** – это плагин (их существует несколько, например в данной версии: **ELK 0.5.0**, **HermiT 1.4.3.456**), который встраивается в Protégé и работает с логикой. К логике в рассматриваемом абстрактном примере относится только то, что свойство *иметь_мать* является *функциональным*. Но уже это позволит получить новые знания. В меню выберите **Reasoner** → **HermiT 1.4.3.456** → **Start Reasoner**.

Теперь, если вы перейдете в **Direct Instances** на клиента *Лазуткина_М.*, то вы увидите предположение, что *Лазуткина_М.* и *Мурина_М.* один и тот же экземпляр класса:

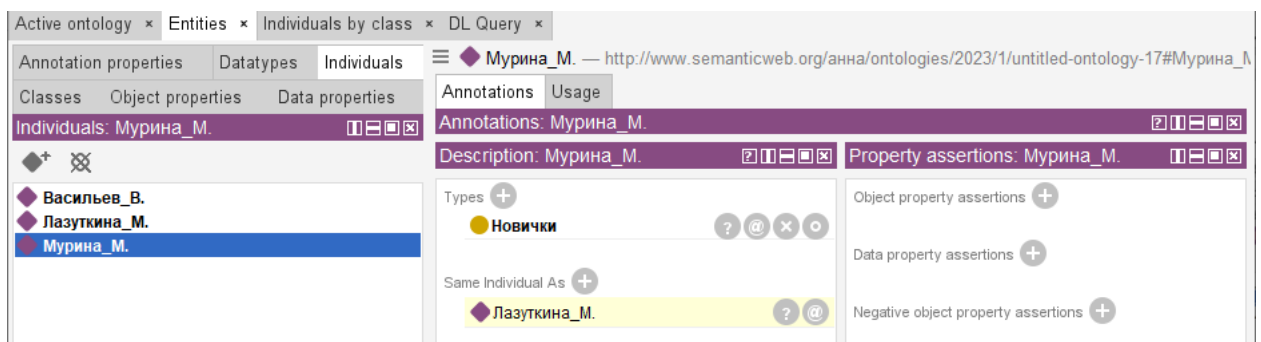
Лазуткина_М. $\xrightarrow{\text{Same Individual As}}$ Мурина_М.


Можно сделать предположение, что возможно эта клиент сменила фамилию и запись о ней дублируется в базе данных клиентов.

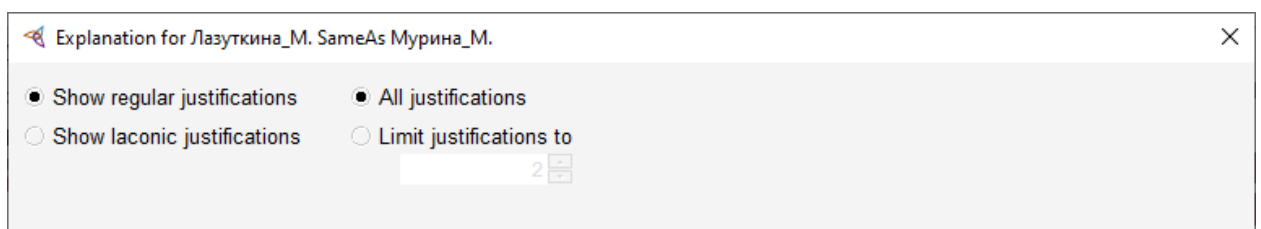


Про клиента Мурина М. тоже появилось предположение, что это тот же самый экземпляр класса, что и Лазуткина_М.:

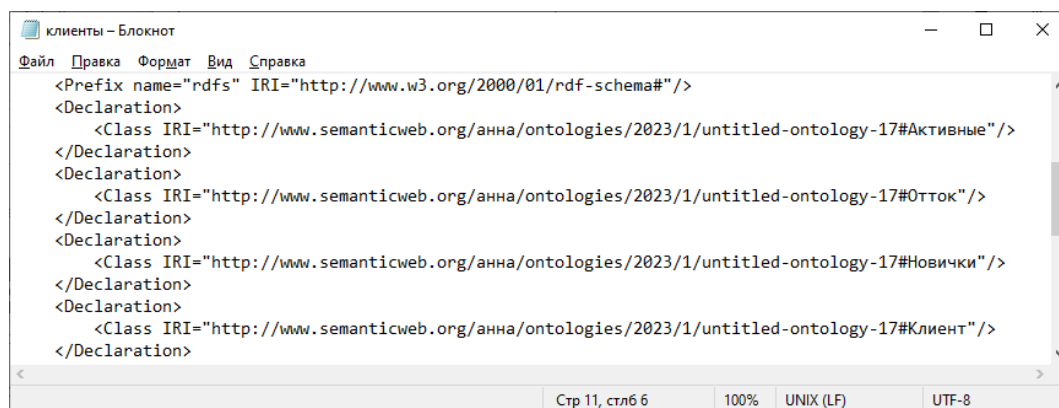
Мурина_М. $\xrightarrow{\text{Same Individual As}}$ Лазуткина_М.



Если исходить из того, что сохраненные факты истинны, а также учитывать функциональность свойства *иметь_мать*, онтология в Protégé, вместо того, что бы запретить вводить второй факт, как было бы в реляционной БД, высказывает предположение, которое изначально не утверждали. Это по сути и есть новое знание (учитывая легкость задачи – знание не очень важное, но здесь главное, что оно было сформулировано на основе вводимых утверждений). Если в разделе **Description** → **Same Individual As** в строке *Лазуткина_М.* нажать на , то откроется окно **Explanation for....**, в котором могут выводиться логические утверждения, на основе которых **Reasoner** сделал вывод, что *Мурина_М.* тот же самый экземпляр, что и *Лазуткина_М.* (в нашем случае утверждение слишком простое и пояснения не вывелись, рассмотрим их на следующих примерах). В зависимости от вида плагина **Reasoner** объяснения могут выглядеть по-разному:






Полученные **Reasoner** знания можно принимать или не принимать. Если полученные утверждения не нужно принимать, то выберите **Reasoner** → **Stop reasoner**. Если полученные утверждения нужно сохранить, то можно экспортировать найденные аксиомы как онтологию с помощью меню **File** → **Export inferred axioms as ontology**. На шагах **Select axioms to export**, **Include asserted axioms**, **Ontology ID** оставьте все без изменений. На шаге **Physical Location** укажите требуемое размещение и имя файла. На шаге **Ontology format** выберите *OWL/XML Syntax*. Сохраненный файл можно посмотреть в текстовом редакторе.



2.2. Свойства-данные (Data Properties)

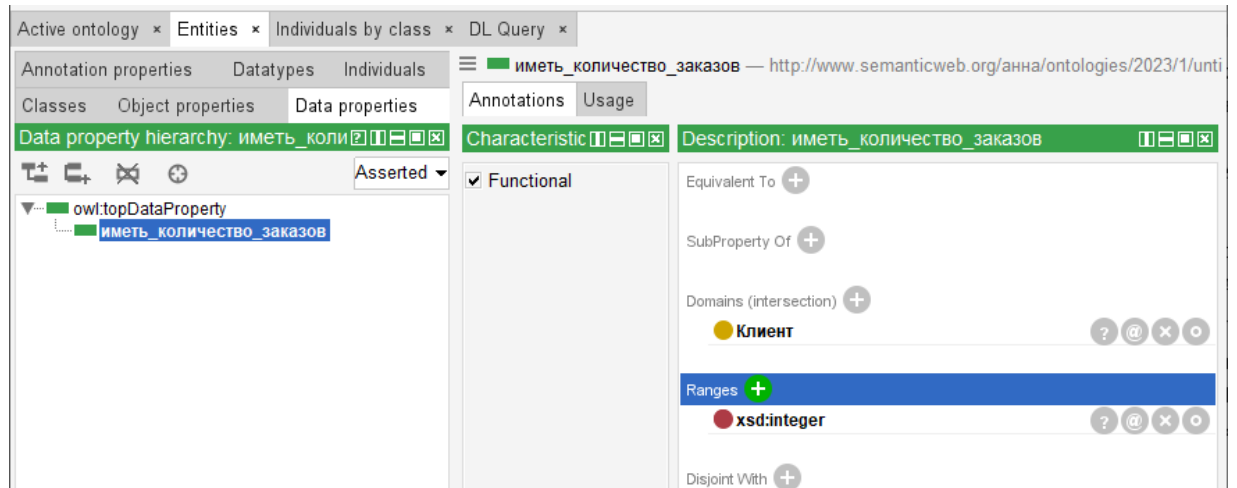
Рассмотрим свойство **Data Properties**, для которого субъектом может быть класс онтологий или экземпляр онтологии, а объектом являются примитивные типы данных (число, строка, переменная булевского типа и пр.).

Хранить значение по количеству заказов клиента, как отдельный объект или экземпляр класса, не имеет смысла. Создадим свойство *иметь_количество_заказов*, для этого нужно зайти в **Data Properties hierarchy** и нажать . У этого свойства доменом будет класс *Клиенты*, поэтому в разделе **Description** щелкните мышью на кнопке  предиката **Domains** и в **Class hierarchy** укажите, что для свойства *иметь_количество_заказов* субъекты берутся из класса *Клиент*. Щелкните мышью на кнопке  предиката **Ranges** (область допустимых значений) и здесь на вкладке **Built in datatypes** в списке примитивных типов данных выберите *xsd:integer*. То есть *Клиент* *иметь_количество_заказов* равное какому-то целому числу *integer*.

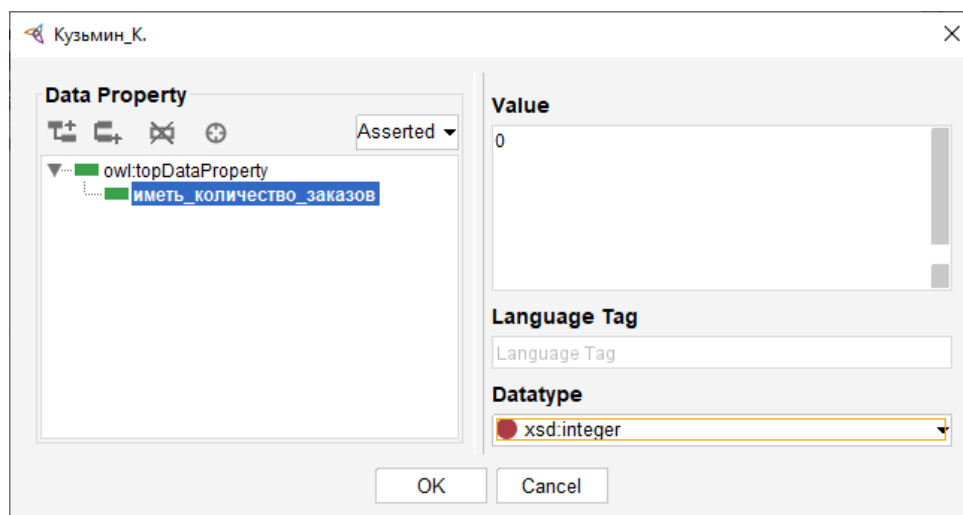
Клиент $\xrightarrow{\text{имеет количество заказов}}$ число *integer*.

Из характеристик свойств, которые относятся к **Data Properties**, есть только характеристика **Functional**. У клиента может быть какое-то одно определённое количество заказов в настоящий момент времени. Поэтому это свойство можно

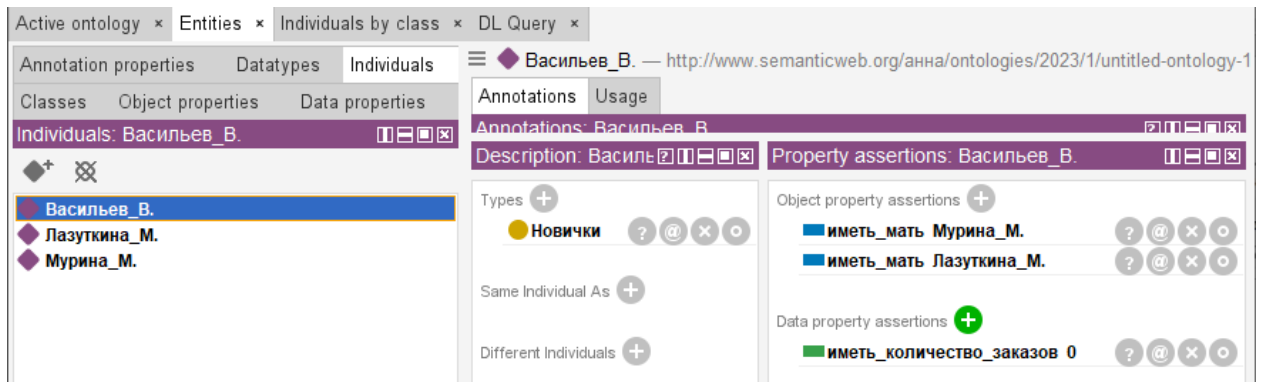
считать функциональным. В разделе **Characteristics**, напротив **Functional** поставьте галочку.



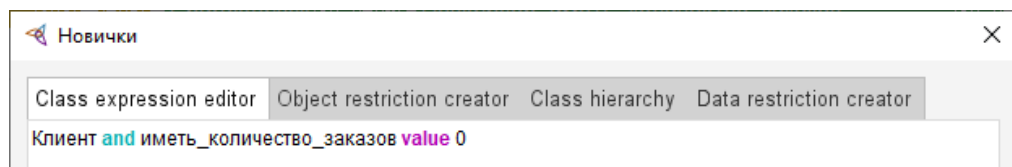
На вкладке **Entities** → **Individuals** выберите клиента Васильева В. В разделе **Property assertions** щелкните мышью на кнопке **Data property assertions**. В открывшемся окне указать, у клиента Васильева В. количество заказов **0** (он новичок), тип **xsd:integer**.



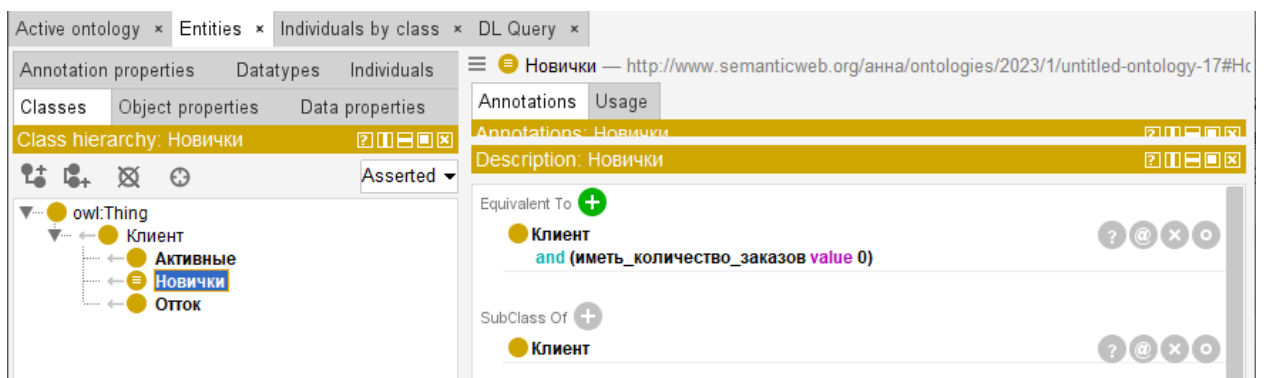
Информация о клиенте *Васильев В.* теперь представлена в таком виде:




Рассмотрим как это можно использовать. На вкладке **Classes** → **Description** есть раздел **Equivalent To**. С помощью логических правил можно описать какие конкретные экземпляры относятся к тому или иному классу. Так, например, любой клиент у которого количество заказов равно **0** автоматически должен относиться к классу *Новички*. Выберите класс *Новички* и нажмите на кнопке **+ Equivalent To**. В открывшемся окне на вкладке **Class expression editor** необходимо ввести логическое выражение:



В этом логическом выражении есть два утверждения. Первое – экземпляр должен быть из класса *Клиент* и второе, которое выполняется одновременно с первым, этот клиент должен *иметь_количество_заказов* равным **0**. Это выражение сохраняется в отношениях эквивалентности.

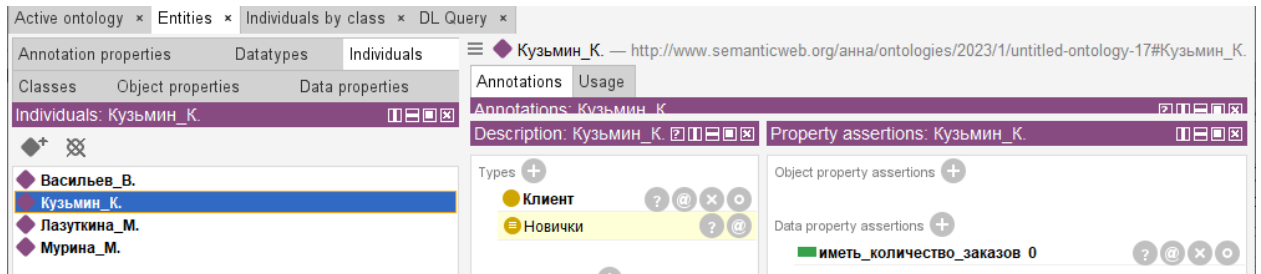



На вкладке **Entities** → **Individuals** создайте новый экземпляр класса *Кузьмин_К*. В **Description** в разделе **Types** определите, что *Кузьмин К*. относится к классу *Клиент*. Неизвестно к какому классу именно клиентов он относится. Однако,

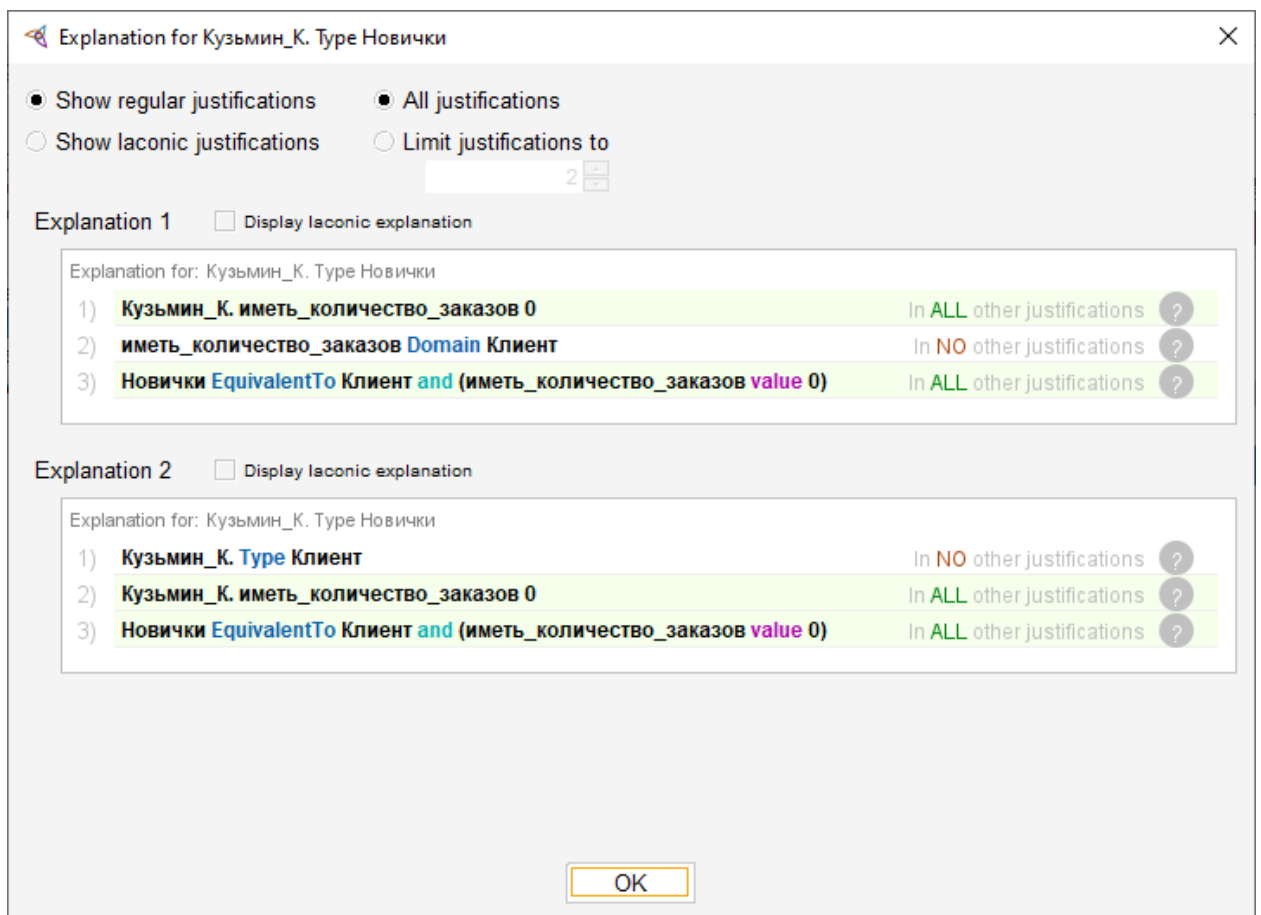
известно, что заказов *Кузьмин_К.* еще не делал. В разделе **Property assertions** щелкните мышью на кнопке  **Data property assertions** и введите утверждение:

Кузьмин_К. $\xrightarrow{\text{имеет количество заказов}}$ 0

В меню выберите **Reasoner** → **HermiT 1.4.3.456** → **Start Reasoner**. Reasoner предположил, что *Кузьмин_К.* относится к классу *Новички*:



Нажмите на  напротив, выданного Reasoner предположения, что *Кузьмин_К.* относится к классу *Новички* (подсвечено желтым цветом). На этот раз сформировались цепочки рассуждений:



Reasoner действительно подробно описывает и приводит 2 цепочки рассуждений. Рассмотрим первую цепочку:

- 1) *Кузьмин_К.* имеет количество заказов 0 - это утверждение было задано;
- 2) *иметь_количество_заказов* это свойство, которое в качестве домена имеет класс *Клиент* - и получается даже не нужно утверждать *Кузьмин_К.* – это *Клиент*. Это утверждение, что *Кузьмин_К.* имеет тип *Клиент*, в этой цепочке не понадобилась. Это можно предположить на основе того, что у него могут быть заказы. А в этой онтологии заказы бывают только к клиентов. Следовательно *Кузьмин_К.* – клиент;
- 3) *Новички* эквивалентно *Клиентам*, которые имеют количество заказов равным 0.

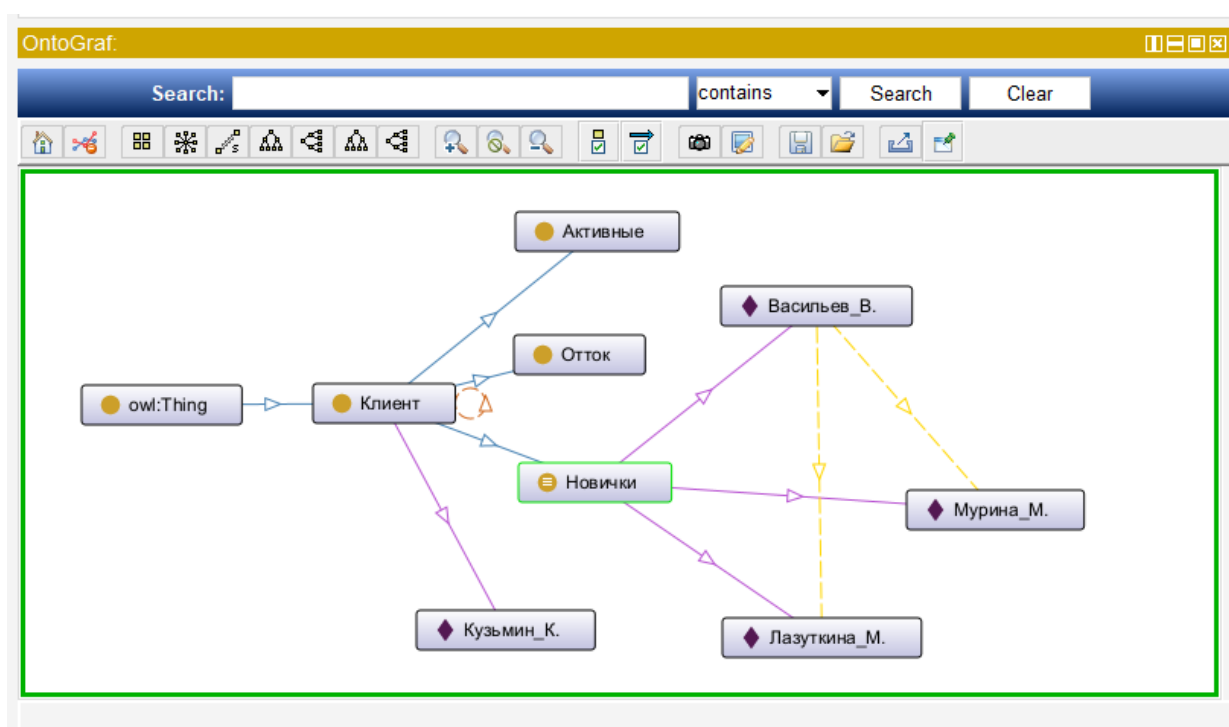
Рассмотрим вторую цепочку рассуждений:

- 1) *Кузьмин_К.* это *Клиент* - это было задано;
- 2) *Кузьмин_К.* имеет количество заказов 0 – это тоже было задано;
- 3) *Новички* эквивалентно классу *Клиент*, которые имеют количество заказов равным 0.

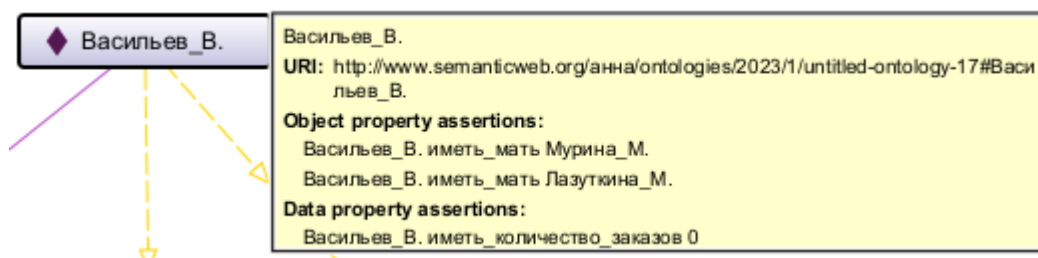
Кузьмин_К. подходят под пункт 1, под пункт 2 и, следовательно, под пункт 3. Рассуждение не гениальное, но это пример.

3. Визуализация онтологий

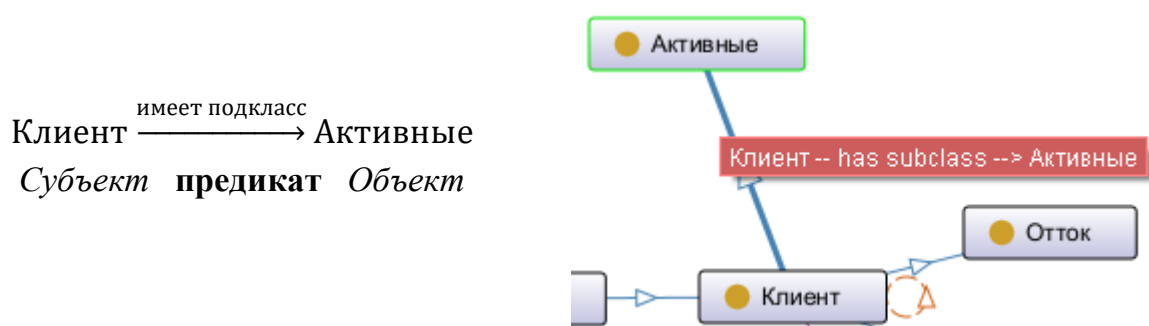
Рассмотрим графическое представление онтологии. Для этого в Protégé есть плагин **OntoGraf**. Выберите в меню **Windows** → **Class Views** → **OntoGraf**. двойным щелчком мышки на объектах онтологии раскройте ее, как показано ниже:



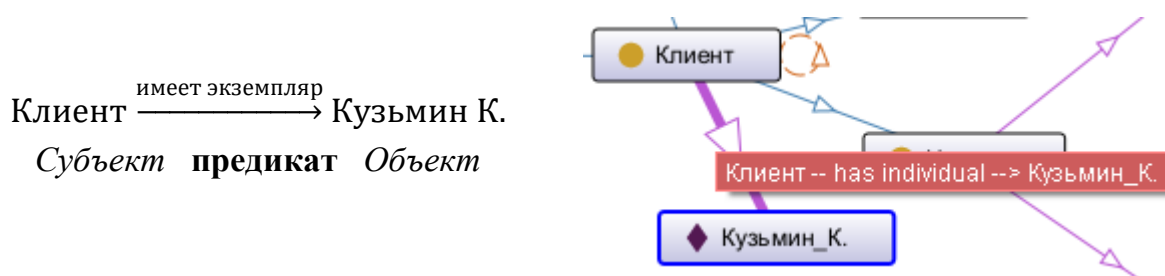
Желтыми кружками обозначены классы и субклассы, ромбиками обозначены экземпляры классов. У каждого экземпляра класса, наведя на него курсор мыши, можно посмотреть аннотацию:



Все элементы на схеме онтологической модели связаны между собой стрелками. Стрелки представляют собой графическое отображение свойств **Object properties**. Если навести курсор мыши на стрелку отобразится триплет:

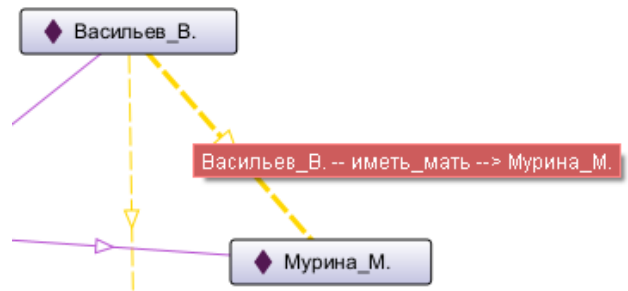




Предикат *has_subclass* (*имеет_подкласс*) является предопределенным (синяя стрелка). Предопределенным является так же предикат *has_individual* (*имеет_экземпляр*), обозначен сиреневой стрелкой:

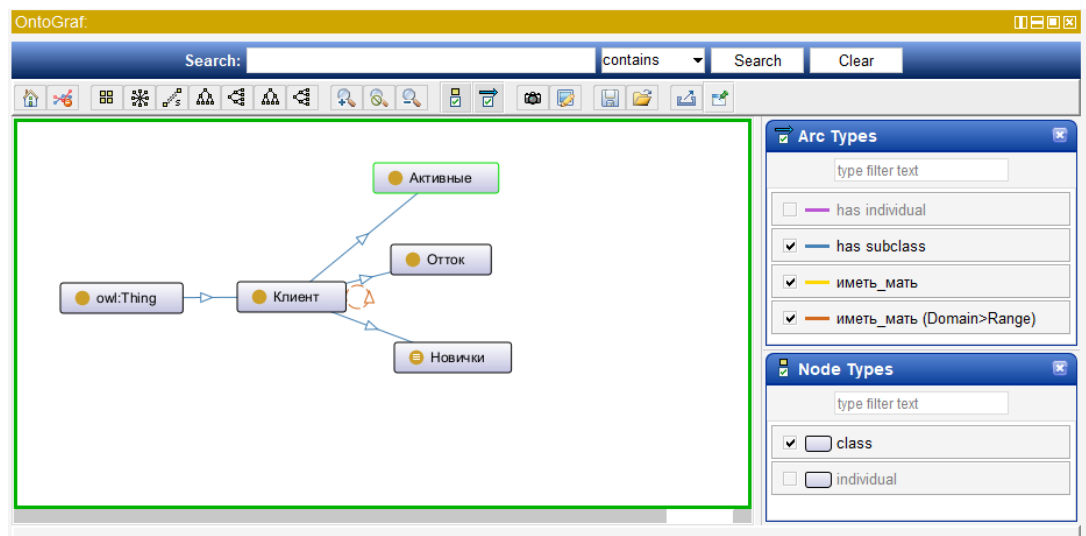



Свойства, введенные в онтологию пользователем, обозначены желтыми стрелками, они связывают экземпляры между собой

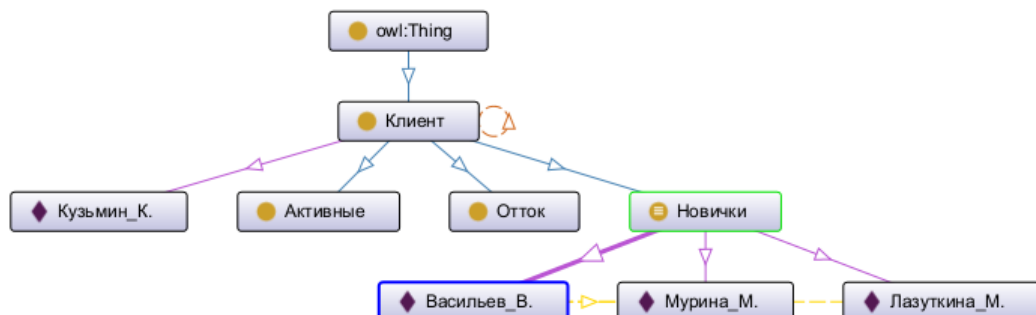
Васильев_В. $\xrightarrow{\text{иметь_мать}}$ Мурина_М.
 Субъект предикат Объект

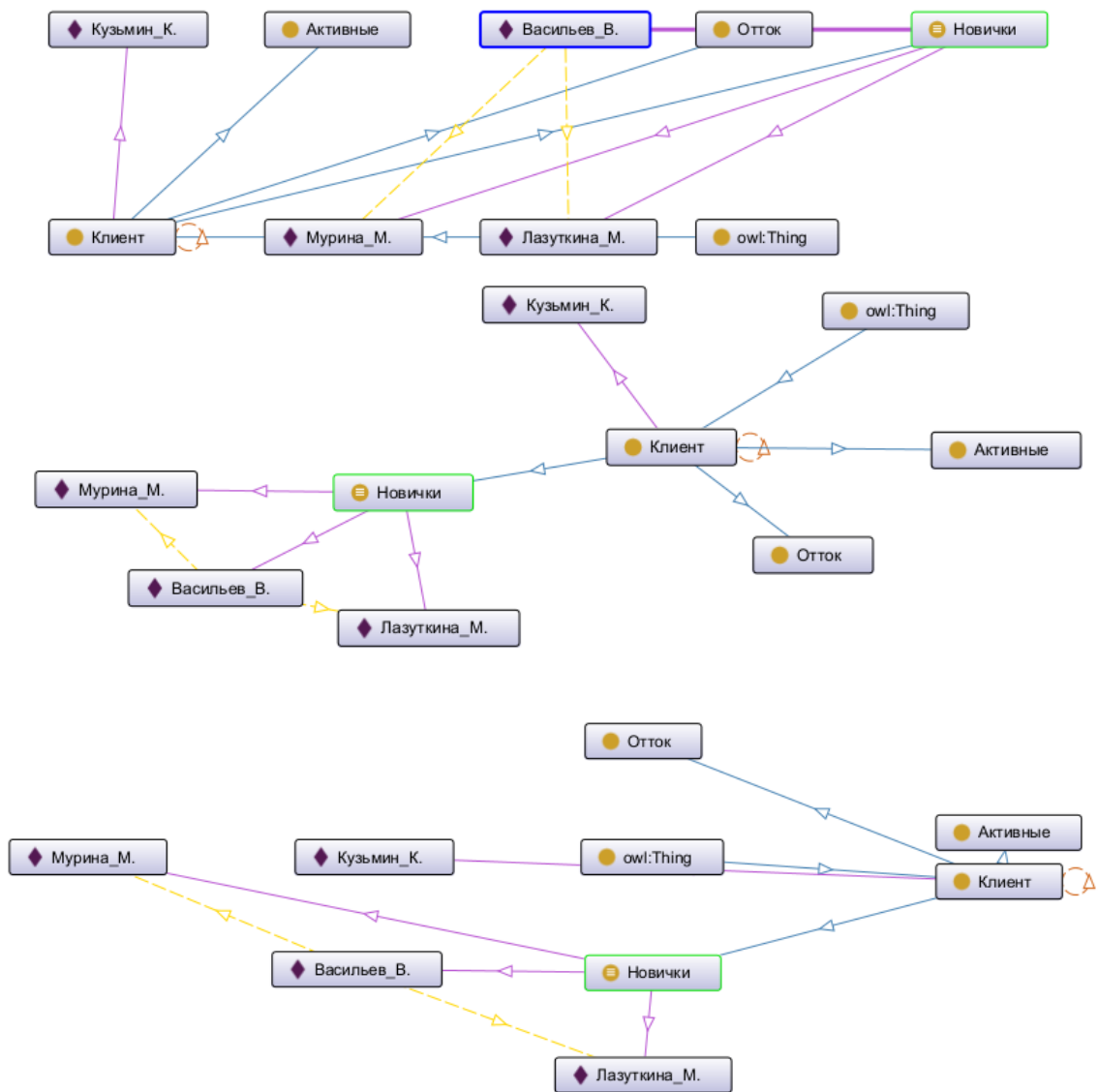


Вся информация, которая заложена в онтологию может быть представлена в виде графа. Но как только онтология начинает расти, она становится трудно просматриваемой. С помощью команды **Arc Types**  можно управлять видимостью связей онтологий. С помощью команды **Node Types**  можно управлять видимостью классов и экземпляров классов.



Поэкспериментируйте с панелью , посмотрите визуализацию онтологии в разных представлениях:



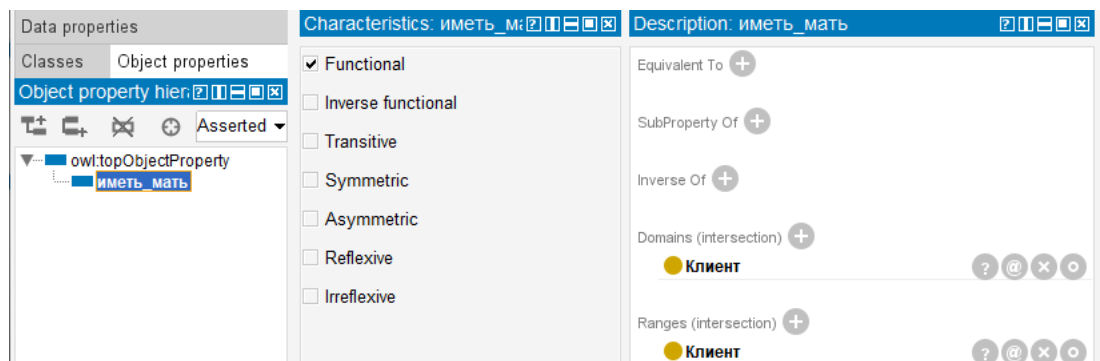


Закройте **OntoGraf**.

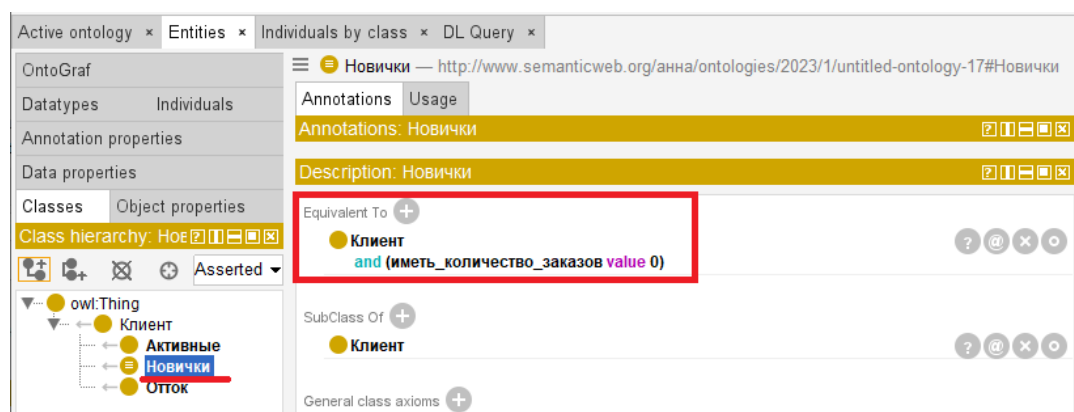
4. Обратное свойство (Inverse Of) и обратно-функциональное свойство (Inverse functional)

Рассмотренный граф представляет собой исходную информацию, заложенную в систему. Над графом выстраивается логика. Логiku можно выстраивать разными способами, рассмотрим некоторые:

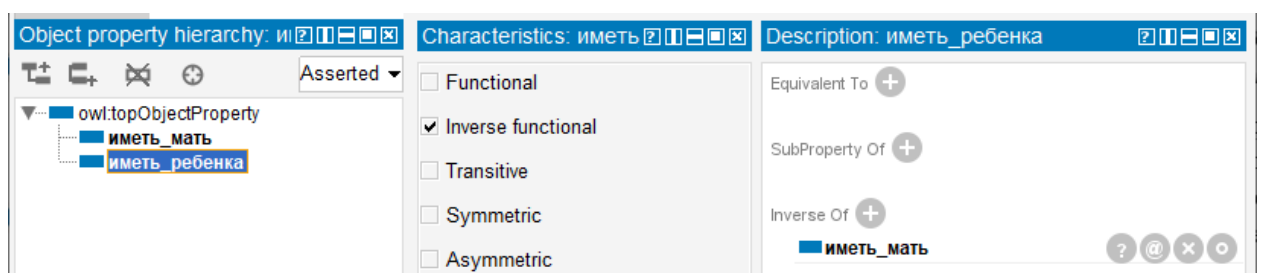
- 1) С помощью характеристик свойств. Одно из этих свойств **Functional** уже рассмотрели ранее.



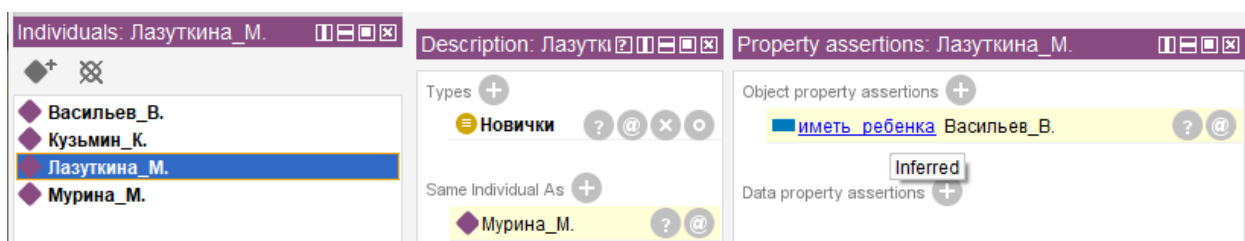
- 2) Построение выражения, которое определяет какие именно экземпляры попадают в данный класс (**Equivalent To** - эквивалентность).



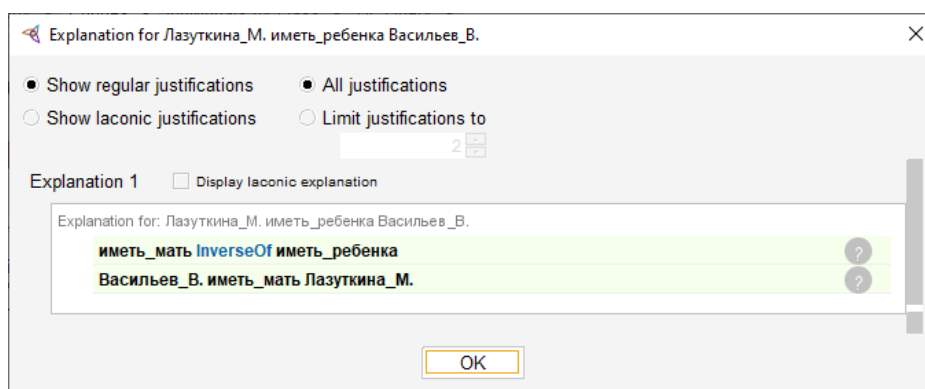
Создадим еще одно свойство **Object Properties**. Назовем его *иметь_ребенка*. Отметим, что свойство *иметь_ребенка* является обратным для свойства *иметь_мать*. Для этого поставьте курсор мыши на свойство *иметь_ребенка*, в разделе **Description** выберите **Inverse Of** . Свойство *иметь_мать* ранее определили как функциональное (**Functional**) – это значит, что детей может быть много, а свойство *иметь_мать* ставят им в соответствие одну-единственную мать. Свойство *иметь_ребенка* является обратным или обратно функциональным (**Inverse Functional**) *иметь_мать* – это означает, что у матери может быть много детей, но мать у детей всегда одна.



Перейдите на вкладку **Entities** → **Individual** и запустите **Reasoner**.



Предположение, что Лазуткина М. и Мурина М. один и тот же экземпляр класса: Лазуткина_М. $\xrightarrow{\text{Same Individual As}}$ Мурина_М. было обнаружено ранее. А вот предположение, что Лазуткина_М. связана свойством *иметь_ребенка* с Василийев_В. обнаружено впервые: Лазуткина_М. $\xrightarrow{\text{иметь_ребенка}}$ Василийев_В. Если навести курсор мыши на *иметь_ребенка* появится слово **Inferred** (найдено **Reasoner**). Запустим **Explain Inference** ?.



Свойство *иметь_мать* обратно к свойству *иметь_ребенка*, следовательно Василий В. связан свойством *иметь_мать* с Лазуткиной М. Тоже самое получится если посмотреть на клиента Мурина М.

5. Транзитивность свойств (Transitive)

Закон транзитивности:

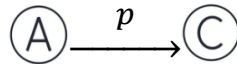
$$a = b, b = c \text{ следовательно } a = c$$

Оператор сравнения $=$ обладает характеристикой транзитивности, он транзитивен. Тоже самое можно говорить про предикаты, которые связывают субъект и объект. a равно b ($a = b$) по сути является триплетом, где a – субъект, b – объект, p – предикат.

Если имеются произвольные экземпляры или классы a и b , и они связаны каким-то свойством p (p – предикат), то если p обладает характеристикой *транзитивности*, то из того, что a связано с b свойством p и b связано с c свойством p :

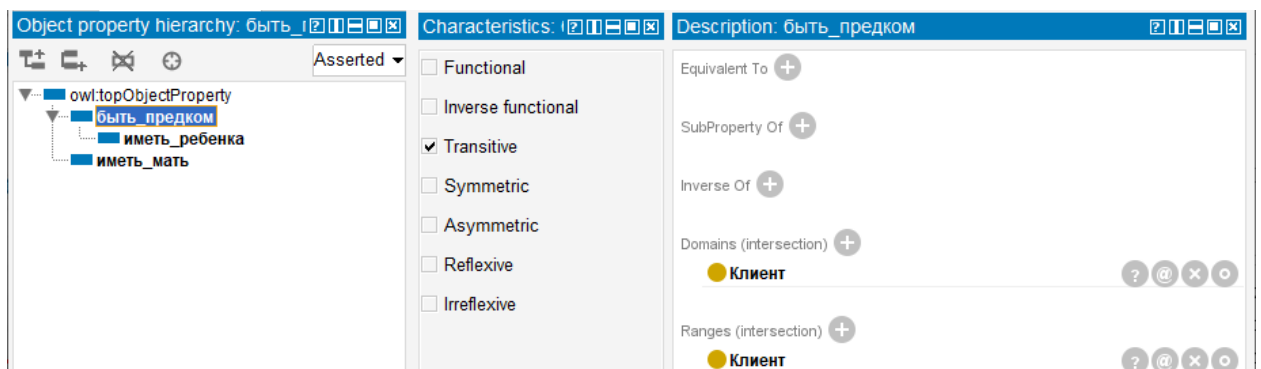


из этих двух утверждений следует, что **a** и **c** тоже связаны свойством **p**:

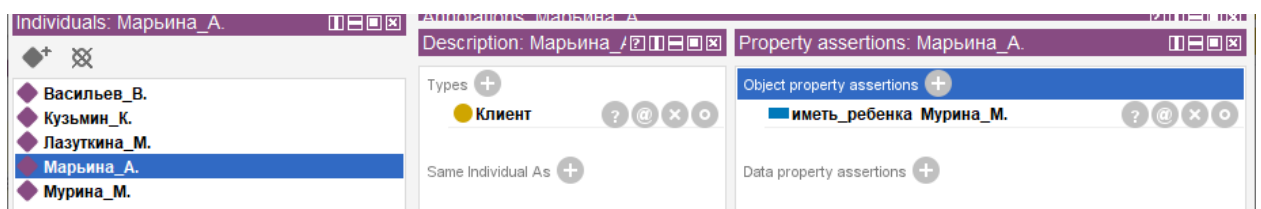


Все по аналогии с $=$, только **p** в данном случае представляет собой произвольное свойство.

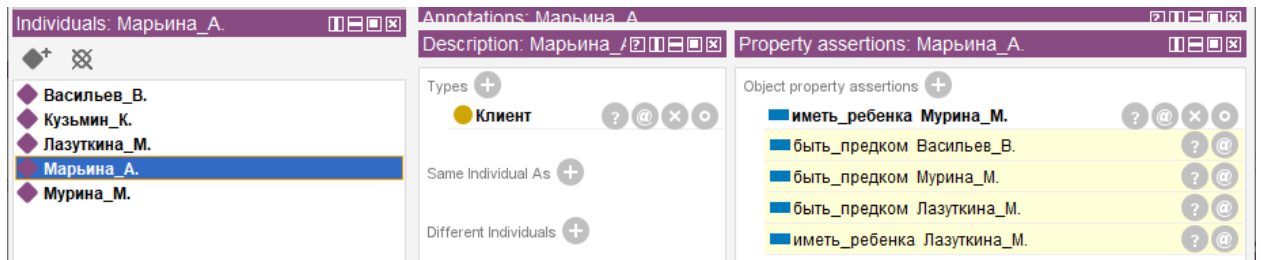
Возвращаемся к онтологии. Проверим как работает *транзитивность* на новом свойстве. Создайте свойство *быть_предком* (быть мамой, папой, дедушкой, бабушкой, прадедушкой, прабабушкой). Свойство *иметь_ребенка* можно считать подсвойством, частным случаем свойства *быть_предком*. Если у человека есть ребенок, то этот человек является предком ребенка, но не наоборот. Быть предком - это необязательно быть родителем, можно быть и бабушкой, и дедушкой и т.д. Далее отметим, что свойство *быть_предком* является *транзитивным*, потому что если **a** является предком **b**, а **c** является предком **a**, то **c** является предком **b** (просто более дальним). Введите информацию как показано на рисунке:




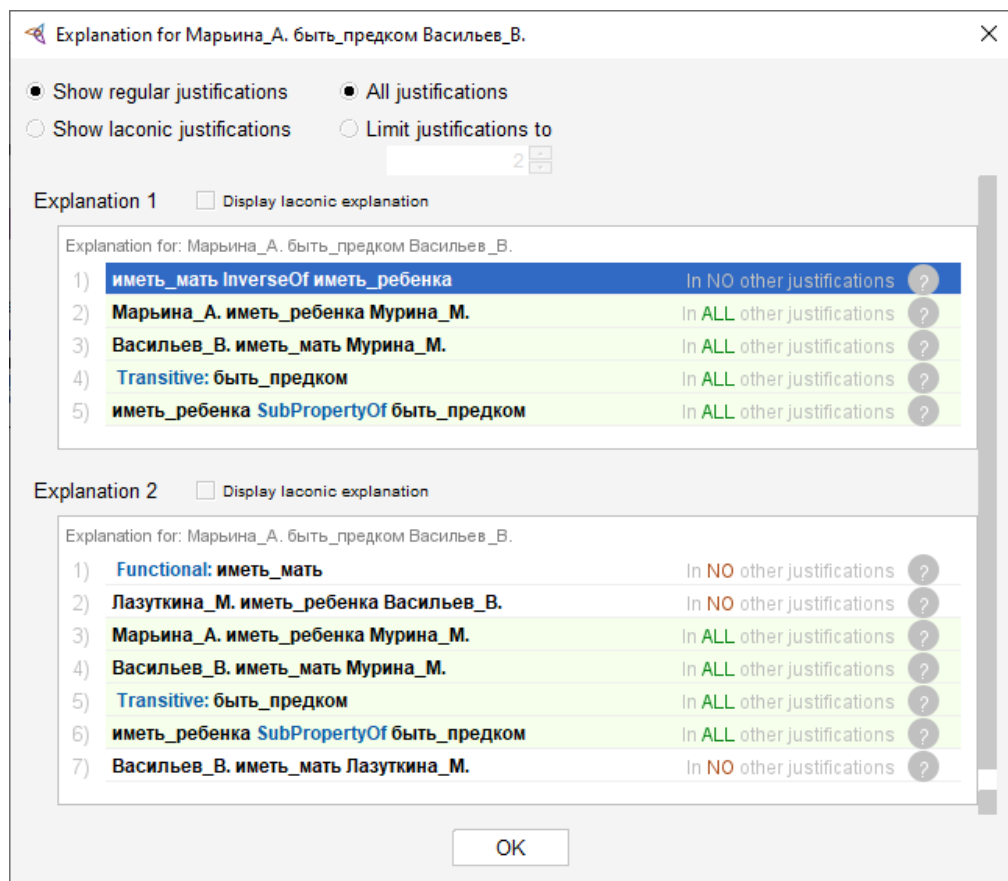
На вкладке **Entities** → **Individuals** создадим нового клиента Марьину А. Введите информацию как показано на рисунке:



Запустите **Reasoner** и получите много новой информации:



Посмотрите, как **Reasoner** объясняет, что *Марьяна_А.* $\xrightarrow{\text{быть_предком}}$ *Васильев_В.*.
 Нажмите в этой строке  и посмотрите как свойство *транзитивности* отразилось в цепочках рассуждений:



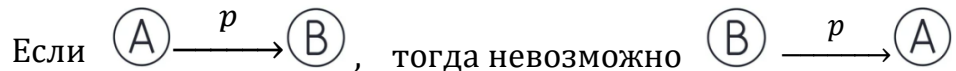
6. Характеристики свойств: Symmetric, Asymmetric

Продолжаем рассматривать характеристики свойств: симметричность, асимметричность, рефлексивность и иррефлексивность.

Симметричность. Если свойство **p** является симметричным, то из того, что **a** связано с **b** свойством **p** автоматически следует, что **b** тоже связано с **a** свойством **p**. При условии, что **p** симметричное свойство, из левого утверждения автоматически следует правое утверждение.



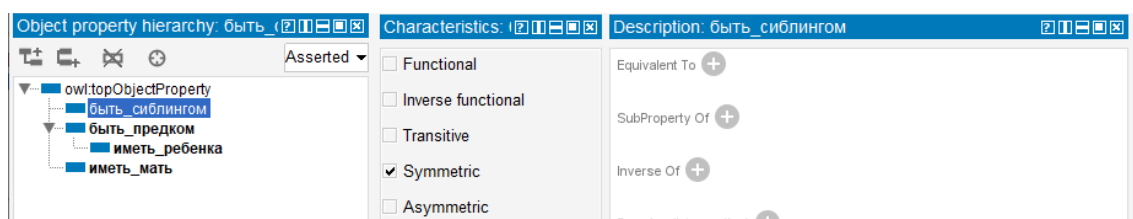
Асимметричность. Обратная ситуация - когда из левого утверждения следует, что правое утверждение невозможно. Если **a** связано с **b** свойством **p**, то ситуация когда **b** связано с **a** свойством **p** не может появиться.



Свойства могут быть симметричными, асимметричными или ни теми и ни другими. Такое возможно в случае, когда триплет, который является следствием из исходного триплета (в данном случае это $(B) \xrightarrow{p} (A)$), может появиться в онтологии, а может не появиться. В этом случае свойство **p** является ни симметричным и ни асимметричным.

При симметричности триплет $(B) \xrightarrow{p} (A)$ обязательно присутствует, при асимметричности $(B) \xrightarrow{p} (A)$ обязательно отсутствует.

Рассмотрим эти свойства в рамках рассматриваемой онтологии. Введем свойство *быть_сиблингом* - быть либо братом, либо сестрой. Что бы не вводить в онтологию различие по полу, выбрали собирательное свойство. *Быть_сиблингом* типичное симметричное свойство. Если **a** является сиблингом **b**, то обратное утверждение обязательно тоже верно - **b** тоже является сиблингом **a**. Отметим это свойство как симметричное:

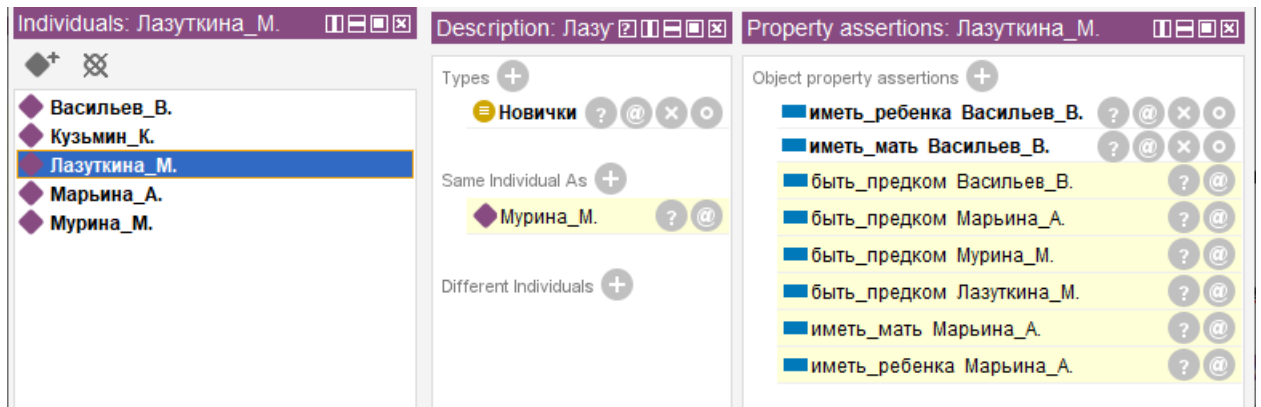


На вкладке **Entities** → **Individuals** создайте новое утверждение *Васильев_В.*
быть_сиблингом → *Кузьмин_К.*

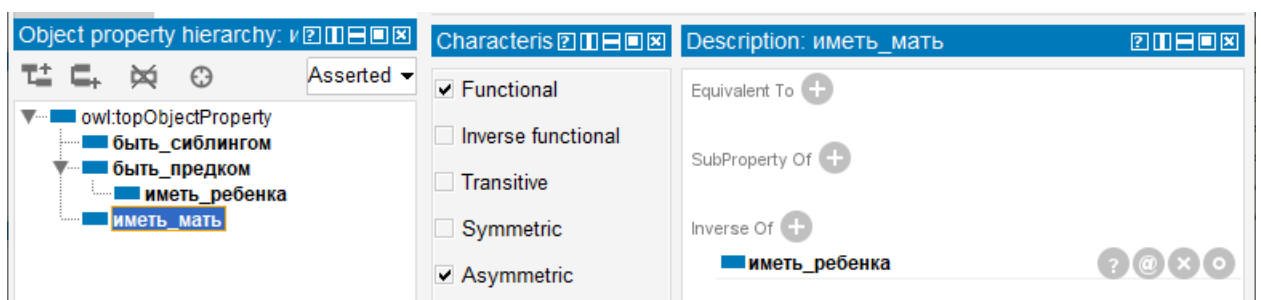
Запустите **Reasoner**. Получаем, что *Кузьмин_К.* тоже является сиблингом *Васильев_В.* Это следствие из симметричности свойства *быть_сиблингом*.

Типичное асимметричное свойство - это свойство *иметь_мать*. Это свойство асимметрично, поскольку из того, что **a** имеет мать **b** следует, что **b** никогда не может быть матерью **a**. Поэтому свойство *иметь_мать* нужно было ранее отметить, как асимметричное, но такой отметки не было сделано. Посмотрим, что получилось. Остановите **Reasoner**. На вкладке **Entities** → **Individuals** отмечено, что *Васильев_В.* имеет мать *Лазуткина_М.* Ранее мы также выяснили что *Лазуткина_М.* и *Мурина_М.* это одна и та же клиентка, которая, например, сменила фамилию. Давайте создадим обратное утверждение *Лазуткина_М.* $\xrightarrow{\text{иметь_мать}}$ *Васильев_В.*

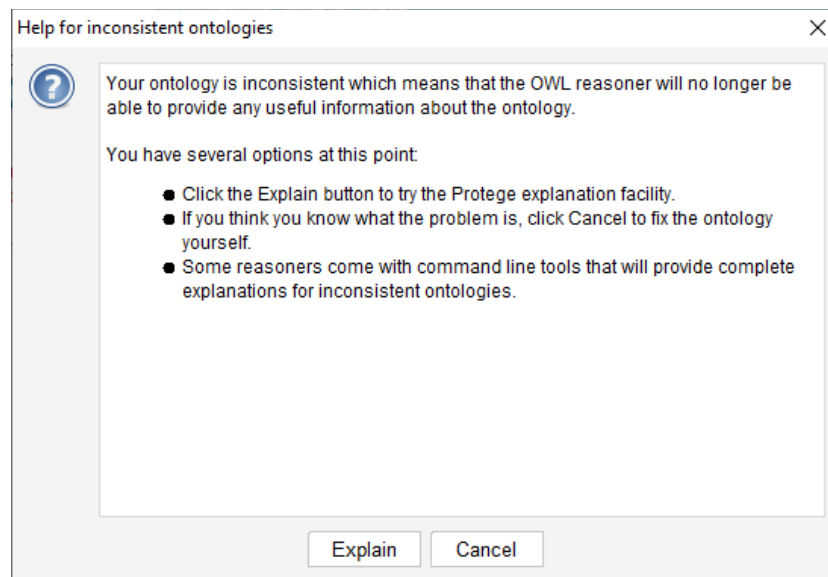
Запустите **Reasoner**. Поскольку *асимметричность* свойства не была задана, выдается много разных не правильных предположений, потому что изначально логика была нарушена.



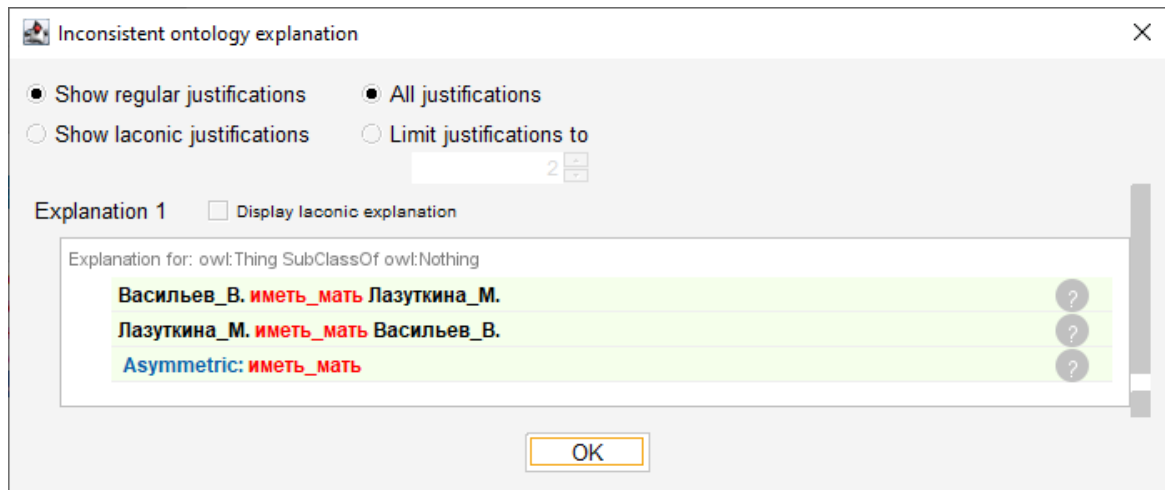
Назовем свойство *иметь_мать* ассиметричным.



Запустите **Reasoner**.



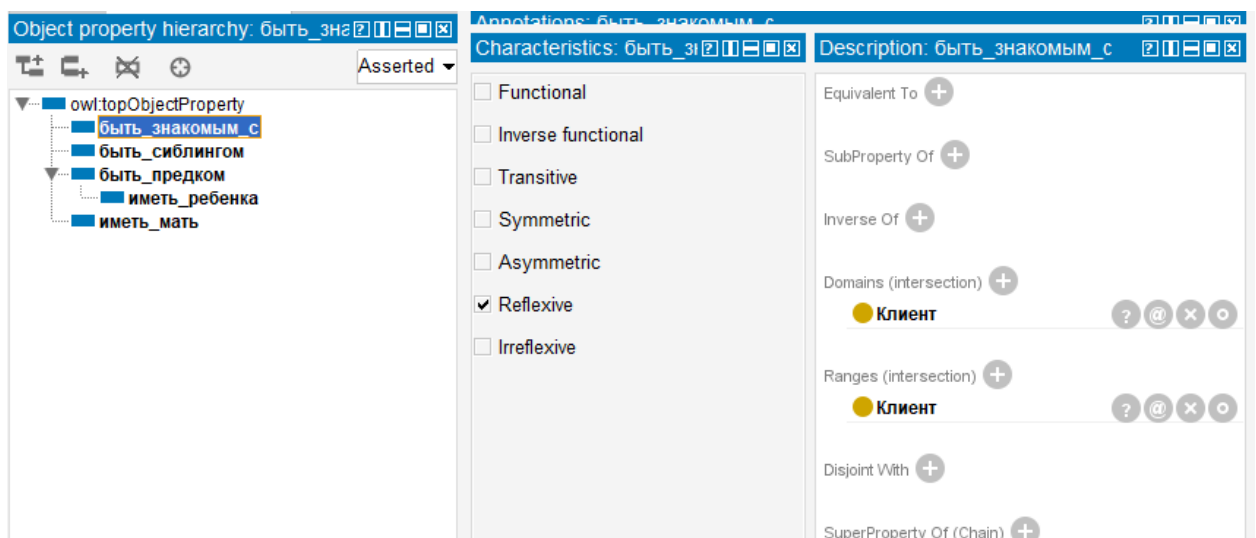
Появилось сообщение об ошибке, что означает - ваша онтология является несовместимой, то есть содержит внутри себя утверждения которые противоречат друг другу. Нажмите **Explain** и можно будет узнать в чем состоит проблема.



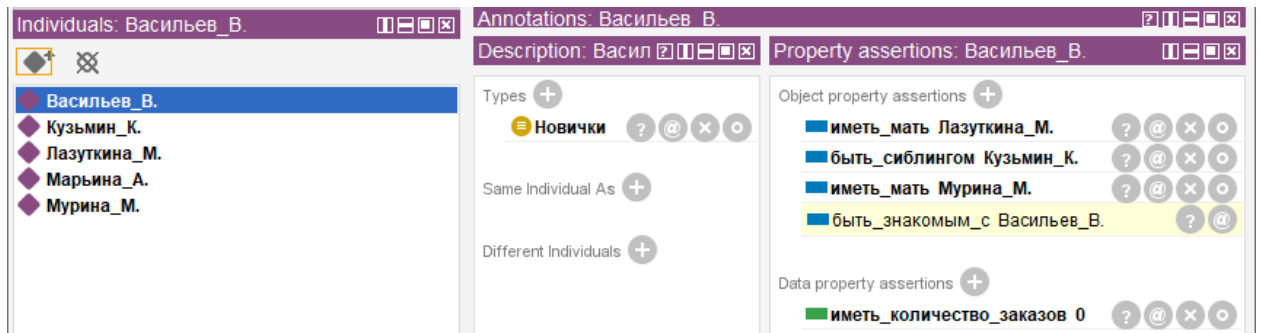
В такой ситуации необходимо остановить **Reasoner**, хорошо подумать и удалить триплет $\text{Лазуткина_М.} \xrightarrow{\text{иметь_мать}} \text{Васильев_В.}$, который противоречит нормальной ситуации.

7. Характеристики свойств: Symmetric, Asymmetric, Reflexive, Irreflexive

Рефлексивное свойство - это такое свойство, для которого всегда любой экземпляр или класс на котором это свойство определено, связан сам с собой этим свойством. Создадим рефлексивное свойство *быть_знакомым_с* и отметим для него характеристики как на рисунке:



При этом каждый клиент знаком сам с собой. Запустите **Reasoner**. И вы увидите, Васильев В. знаком с Васильевым В. и т.д.



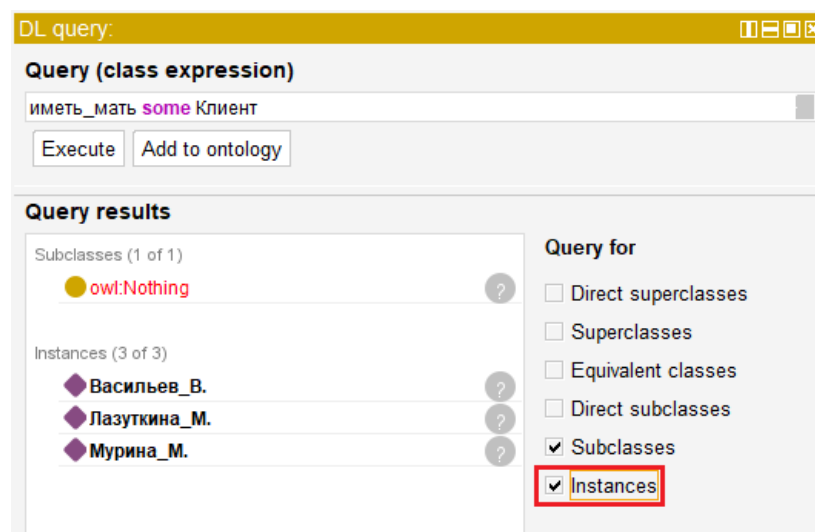
Иррефлексивность - ситуация обратная рефлексивности. Это означает, что если свойства иррефлексивно, то экземпляр или класс никогда не может быть связан сам собой этим свойством. Свойства *иметь_мать* и *иметь_ребенка* являются иррефлексивными и асимметричными.

8. Запросы к онтологии

Вкладка DL Query, предоставляет интерфейс для запроса и поиска в онтологии. Онтология должна быть классифицирована разработчиком, прежде чем ее можно будет запросить на вкладке DL query.

Запрос DL — это выражение класса, построенное с использованием таких конструкций, как *and* (соответствует заданному пересечению) и *some*. Запросы DL - это выражения класса Manchester syntax. Manchester syntax — это удобный синтаксис для онтологий OWL. OWL позволяет определять обратные свойства. иже представлены запросы, которые были написаны с помощью DL Query.

Запрос 1. Введите запрос и нажмите **Execute**.



Данный запрос выводит все **Individuals**, у которых есть **Object Properties** *иметь_мать*. Этими **Individuals** являются: *Васильев_В.*, *Лазуткина_М.*, *Мурина_М.*

Запрос 2. Введите запрос и нажмите **Execute**.

The screenshot shows a web interface for a DL query. At the top, there's a yellow header with 'DL query:'. Below it, a text box contains the query: 'inverse быть_знакомым_с some {Мурина_М.}'. There are two buttons: 'Execute' and 'Add to ontology'. Below the query box, the 'Query results' section is visible. It shows 'Subclasses (1 of 1)' with 'owl:Nothing' and 'Instances (2 of 2)' with 'Лазуткина_М.' and 'Мурина_М.'. On the right, a 'Query for' section has checkboxes for 'Direct superclasses', 'Superclasses', 'Equivalent classes', 'Direct subclasses', 'Subclasses' (checked), and 'Instances' (checked).

Данный запрос выводит все **Individuals**, которые имеют **Object Properties** *быть_знакомым_с* *Мурина_М.* Этими *Individuals* являются: *Лазуткина_М.*, *Мурина_М.* **Reasoner** автоматически определяет, что *Мурина_М* знакома сама с собой.

Запрос 3. Введите запрос и нажмите **Execute**.

The screenshot shows a web interface for a DL query. At the top, there's a yellow header with 'DL query:'. Below it, a text box contains the query: 'inverse быть_предком some {Марьяна_А.} and (быть_знакомым_с some {Лазуткина_М.})'. There are two buttons: 'Execute' and 'Add to ontology'. Below the query box, the 'Query results' section is visible. It shows 'Subclasses (1 of 1)' with 'owl:Nothing' and 'Instances (2 of 2)' with 'Лазуткина_М.' and 'Мурина_М.'. On the right, a 'Query for' section has checkboxes for 'Direct superclasses', 'Superclasses', 'Equivalent classes', 'Direct subclasses', 'Subclasses' (checked), and 'Instances' (checked).

Данный запрос выводит все **Individuals**, которые имеют **Object Properties** *быть_предком* *Марьяна_А* и с помощью оператора *and* **Object Properties** *быть_знакомым_с* *Лазуткина_М.* Этими **Individuals** являются: *Лазуткина_М.*, *Мурина_М.*

Запрос 4. Введите запрос и нажмите **Execute**.

Query (class expression)

```
inverse быть_сестрингом some {Кузьмин_К.} and (иметь_мать some {Лазуткина_М.} and (быть_знакомым_с some {Васильев_В.}))
```

Query results

Subclasses (1 of 1)

- owl:Nothing

Instances (1 of 1)

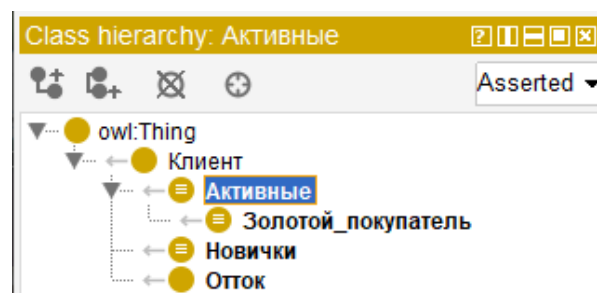
- Васильев_В.

Еще один пример запроса, в котором участвуют уже два оператора **and**. Данный запрос выводит все **Individuals**, которые имеют **Object Properties** *быть_сестрингом* с *Кузьмин_К.*, *иметь_мать* *Лазуткина_М.* и *быть_знакомым_с* *Васильев_В.* Этим **Individuals** является: *Васильев_В.* Тут также как и с *Мурина_М* **Reasoner** автоматически определяет, что *Васильев_В* знаком сам с собой.

Практическое задание

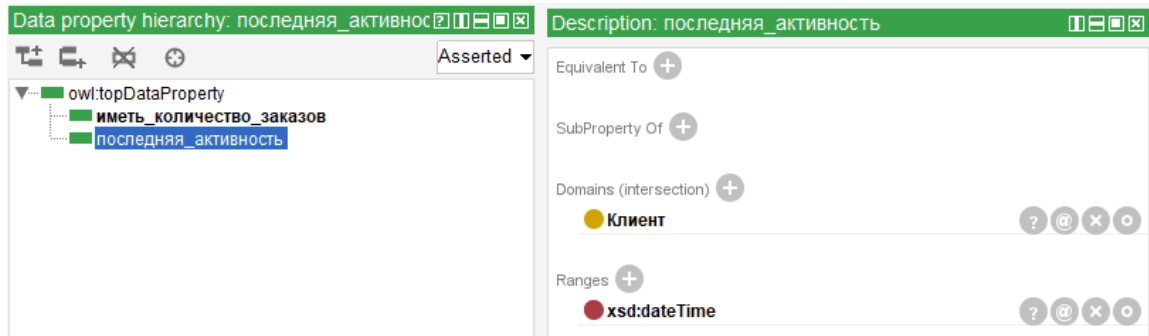
В разработанной онтологии есть *Новички*, который выводится из свойств экземпляров классов. Он показывает, что заказов у пользователя немного. Помимо классов, в онтологии уже есть несколько экземпляров, связанных отношениями «родитель-ребёнок», «брат», «знаком-с». Есть также классы *Активный* и *Отток*, однако в рассматриваемой реализации они не использовались. Классы *Активный* и *Отток* не являются определёнными, а просто являются наследниками класса *Клиент*, что не позволяет использовать их для расширения текущей онтологии. Доработайте текущую онтологию.

Создайте новый класс *Золотой_покупатель*, который будет входить в класс *Активные*:

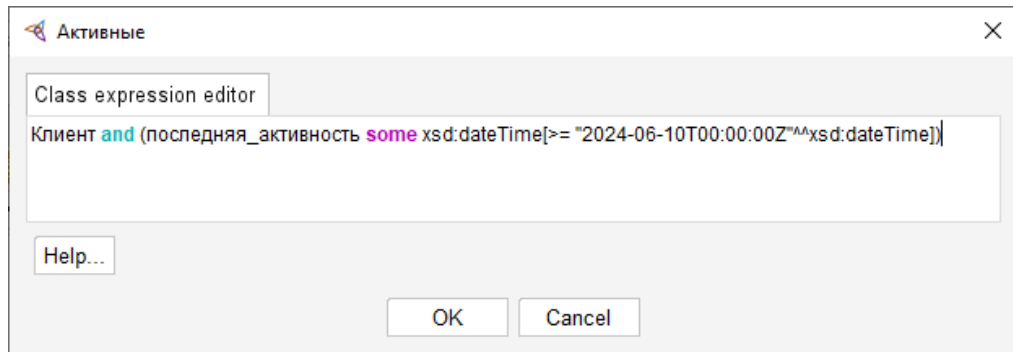


Будем считать, что активными клиентами являются клиенты, которые сделали заказ недавно (до 10 июня 2024 г.). А золотые покупатели это те, которые делали заказ недавно (то есть активный клиент) и имеют более 50 заказов.

Создайте новые **Data Properties** *последняя_активность*. В **Description** введите следующие параметры:

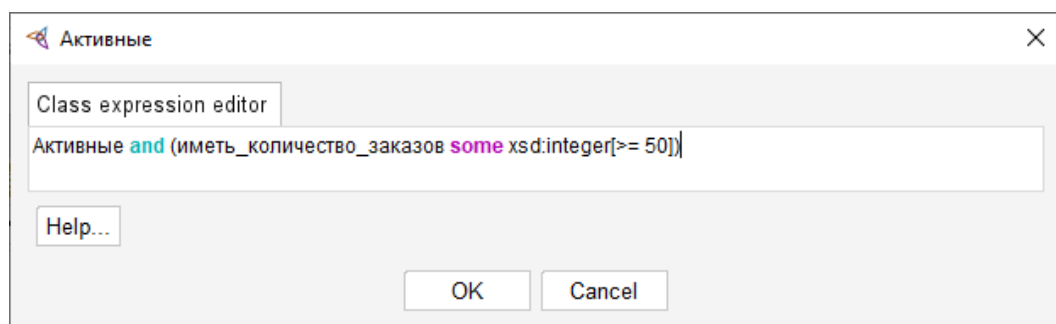


Класс *Активные* определите запросом:



Здесь сначала выбираются все клиенты и с помощью импликации ограничивается множество лишь теми пользователями, у которых свойство *последняя_активность* является датой из промежутка $[2024-06-10; \text{inf}]$. Для представления даты нужно использовать приведение типов (« $^{\wedge}$ »). Для Ввода дат используйте следующий формат ISO 8601: **YYYY-MM-DDTHH:MM:SSZ**.

Соответственно класс *Золотой_покупатель* определяется следующим запросом:



Таким образом объекты этого класса должны в первую очередь удовлетворять условиям класса *Активные*, но помимо этого их свойство *иметь_количество_заказов* должно быть числом из промежутка $[50; \text{inf}]$.

Чтобы проверить эту онтологию на непротиворечивость добавьте ещё 6 новых индивидов, их **Data Properties** приведены в таблице:

Индивид	Количество заказов	Последняя активность
Петрова_А.	52	2024-06-20
Петрова_Е.	67	2024-09-10
Федоров_А.	23	2024-08-12
Федоров_И.	15	2022-06-15
Смирнов_С.	75	2023-03-11
Кузнецова_М.	35	2024-08-31

Пример ввода значения **Data Properties** *последняя_активность*:

The screenshot shows a dialog box titled 'Петрова_Е.' with a 'Data Property' section. In the tree view, 'owl:topDataProperty' is expanded, showing 'иметь_количество_заказов' and 'последняя_активность'. The 'Value' field contains '2024-09-10T00:00:00Z', the 'Language Tag' is empty, and the 'Datatype' is set to 'xsd:dateTime'. There are 'OK' and 'Cancel' buttons at the bottom.

Информацию об остальных клиентах введите самостоятельно. После ввода информации о клиентах **Individuals** будет выглядеть так:

The screenshot shows the 'Individuals' panel with a list of individuals. 'Петрова_Е.' is selected. The 'Property assertions' section shows two assertions: 'иметь_количество_заказов 67' and 'последняя_активность "2024-09-10T00:00:00Z"^^xsd:dateTime'. There are buttons for adding and removing assertions.

Запустите **Reasoner** и проверьте работоспособность онтологии. Убедитесь, что клиенты отображаются в соответствующих классах.

The screenshot displays the DL Query interface with three main panels:

- Individuals: Петрова_Е**: A list of individuals including Васильев_В., Кузнецова_М., Кузьмин_К., Лазуткина_М., Марьина_А., Мурина_М., Петрова_А., **Петрова_Е.** (highlighted), Смирнов_С., Федоров_А., and Федоров_И.
- Description: Петрова_Е**: Shows the type **Золотой_покупатель** and options for 'Same Individual As' and 'Different Individuals'.
- Property assertions: Петрова_Е.**: Contains sections for:
 - Object property assertions**: **быть_знакомым_с Петрова_Е.**
 - Data property assertions**: **иметь_количество_заказов 67** and **последняя_активность "2024-09-10T00:00:00Z"^^xsd:dateTime**.
 - Negative object property assertions**: (empty section)
 - Negative data property assertions**: (empty section)

В **DL Query** сделайте выборку клиентов *Активные* и *Золотой покупатель*.
Внесите, если необходимо изменения в онтологию и определите класс *Отток*.