```
(cl-add-rule 'two-bedrooms-and-family-size @two-bedroom
                                       '(all possible-clients
                                               (at-most 4  family-members)))
```

There is a problem in defining such an upper bound for the class *more-that-three-bedroom*. It is certainly possible to write a rule that uses a Test-C function to check whether the family size is not too large respect to the number of bedrooms of the house being analized. This rule can be used to check if a certain client is a possible client for a certain house, but not for retrieving all the individuals satisfing the restriction that the rule creates.

This problem rises since the restriction is based on TEST-C. Every time a house is inserted in the KB, a "prototypical" client is assigned as a filler of the *possible-client* role. To retrieve all the possible clients for a house, we first generate a derived concept that contains all the derived information in the prototypical client and then we ask for all the instance of this derived concept. The problem is that the derived concept cannot contain the restriction created with a TEST-C, since TEST-C restrictions are treated like black boxes by the classifier.

This problem is easily solvable creating a class for each possible number of bedrooms, if we assume that a Real Estate agency in Pittsburgh doesn't deal with castles and therefore we can satisfactory stop creating new classes at 5 or 6 bedrooms. Nonetheless, the problem has been addressed to show the kind of problems that the lack of structural descriptions in Classic can generate.

- A second set of rules defines a lower bound on the possible clients' income depending on the house's cost.
- A rule takes care of restricting the possible clients of a house for rent to be interested in renting (an there is a similar rule for houses on sale).

There are no rules simmetric to the ones that constraint the proposable houses' locations for a client if the client is a student. Infact these rules would constrain the possible clients of an house in the university areas to be students only, which is definitely a too strong constraint. For the same reason there are no rules that constraint possible clients of an house with a primary school in its vicinities to have kids in scholar age.

The complete process to find out what are the most probable clients interested in an new house available for renting or selling is the following. First, the agency queries the knowledge base to obtain the possible clients. Then, it checks for each client if the house belongs to the client's list of proposable house, obtaining the final the selection of clients to notify. It is during the second phase, for example, that a client is discarded (or just considered with lower priority) if it is a student and the house just inserted is not in the university area.

```
     (2B3 (STATUS FOR-RENT) (LOCATION OAKLAND) (NUMBER-OF-BEDROOMS 2)(PETS-ALLOWED YES)...)
> (all-role-fillers @2b6)
     (2B6 (STATUS FOR-RENT) (LOCATION SHADYSIDE) (NUMBER-OF-BEDROOMS 2) (RENT 700.0)
     (AVAILABLE-SERVICES PRIMARY-SCHOOL SECONDARY-SCHOOL) (PETS-ALLOWED YES) ...)
> (all-role-fillers @2b7)
     (2B7 (STATUS FOR-RENT) (LOCATION SQUIRREL-HILL) (NUMBER-OF-BEDROOMS 2)
     (AVAILABLE-SERVICES NONE)(PETS-ALLOWED YES) (RENT 500.0)...)
```

As a second step, the agency can enter the description of the new client in the client taxonomy and ask for proposable houses. In the example shown in the previous section, the proposable houses for *@ind20* contained just *@2b3*. The intersection between the houses satisfying the query and the proposable-houses gives the set of houses that the agency can submit for evaluation to *@ind20*.

Another inference the system is able to make is what individuals in the Clients Taxonomy could be interested in a certain house. In this way, every time a new house becomes available in the database, the agency can immediately find out which clients to notify. The inference is based on the same mechanism as the one for retrieving the proposable houses for a client.

The role *possible-client* has been defined for the concepts *house-for-sale* and *house-for-rent* and there are rules that impose restrictions on this role depending on houses' properties (they are basically simmetric to the rules for the *proposable-house*).

- A first set of rules imposes an upper bound on the client's family size depending on the house's number of bedrooms, such as

```
                @c{PERSON-WITH-FAMILY} @c{PERSON} @c{CLIENT}
                @c{PERSON-WITH-INCOME})
    Role Fillers and Restrictions
      Pets![1 ; 1] -> (YES)
      Interested-in![1 ; 1] -> (RENTING)
                            => (ONE-OF RENTING BUYING)
      Family-members[2 ; 2] -> (@i{W2} @i{KID9})
      Income![1 ; 1] -> (900)
                     => Tests: ((INTEGERP) (ATOM) (NUMBERP))
                        Interval: [-INF ; 1000]
      Proposable-house[1 ; INF] -> (@i{PH20})
                                => Primitive ancestors: (@c{INANIMATE-THING}
                                                         @c{CLASSIC-THING})
                                   Role Restrictions:
                                     Status![1 ; 1] -> (FOR-RENT)
                                     Possible-clients => @c{CLIENT}
                                     Rent! => Tests: ((ATOM) (NUMBERP) (FLOATP))
                                              Interval: [-INF ; 500.0]
                                     Basic-rent! => @c{FLOAT}
                                     Pets-allowed![1 ; 1] -> (YES)
                                     Location! => @c{URBAN-UNIVERSITY-AREA}
                                     Number-of-bedrooms! => Tests: ((INTEGERP))
                                                            Interval: [1 ; INF]
                                     Date-of-construction! => @c{INTEGER}
                                     Available-services[1 ; 9] -> (@i{SECONDARY-SCHOOL})
                                                               => @c{SERVICE}
      Age! => @c{INTEGER}
      Profession![1 ; 1] -> (@i{STUDENT})
@i{IND20}

> (list-proposable-houses @ind20)
(1B3 (STATUS FOR-RENT) (LOCATION OAKLAND) (NUMBER-OF-BEDROOMS 1) (RENT 300.0)
     (AVAILABLE-SERVICES PRIMARY-SCHOOL SECONDARY-SCHOOL...)(PETS-ALLOWED YES) ...)
(2B3 (STATUS FOR-RENT) (LOCATION OAKLAND) (NUMBER-OF-BEDROOMS 2) (PETS-ALLOWED YES)
     (AVAILABLE-SERVICES PRIMARY-SCHOOL SECONDARY-SCHOOL) (BASIC-RENT 300.0) (RENT 450.0)...)
```

Note that, in the last example, all the houses proposed for @ind20, which is a student, are located in Oakland, which is an university area. Furthermore, they allows pets and have a secondary school available

Queries to The KB

A typical operation supported by terminological languages is the retrieval of all the individuals that satisfy a given description. Therefore, our KB is able to accept a description of a house from a client and to return all the available houses that satisfy the description.

Let's suppose that the client that corresponds to the individual *@ind20* in the above example requests to the agency a two-bedrooms house for rent in the university area, where pets are allowed. At first, the agengy can tranform the request in a query for the Classic KB

```
> (find-instances-of '(and two-bedroom
                           house-in-university-area
                           (fills pets-allowed 'yes)))

(@i{2B3} @i{2B6} @i{2B7})

> (all-role-fillers @2b3)
```

```
                                            Status![1 ; 1] -> (FOR-RENT)
                                            Possible-clients => @c{CLIENT}
                                            Rent! => Tests: ((ATOM) (NUMBERP) (FLOATP))
                                                     Interval: [-INF ; 1500.0]
                                            Basic-rent! => @c{FLOAT}
                                            Location! => @c{GEOGRAPHIC-LOCATION}
                                            Number-of-bedrooms! => Tests: ((INTEGERP) (ATOM) (NUMBERP))
                                                                   Interval: [1 ; INF]
                                            Date-of-construction! => @c{INTEGER}
                                            Available-services[1 ; 9] -> (@i{PRIMARY-SCHOOL})
                                                                      => @c{SERVICE}
      Family-members[2 ; 2] -> (@i{W1} @i{KID1})
      Age! => @c{INTEGER}
@i{IND1}
```

The KB retrieves the following proposable houses for @ind1[8]

```
> (list-proposable-houses @ind1)

(1B3 (STATUS FOR-RENT) (LOCATION OAKLAND) (NUMBER-OF-BEDROOMS 1)
     (AVAILABLE-SERVICES PRIMARY-SCHOOL SECONDARY-SCHOOL ...) (PETS-ALLOWED YES) (RENT 300.0)...)

(2B2 (STATUS FOR-RENT) (LOCATION DOWNTOWN) (NUMBER-OF-BEDROOMS 2)
     (AVAILABLE-SERVICES PRIMARY-SCHOOL SECONDARY-SCHOOL) (PETS-ALLOWED NO) (RENT 850.0) ...)

(2B3 (STATUS FOR-RENT) (LOCATION OAKLAND) (NUMBER-OF-BEDROOMS 2) (PETS-ALLOWED 'yes)
     (AVAILABLE-SERVICES PRIMARY-SCHOOL SECONDARY-SCHOOL...) (rent 450.0) ...)

(2B4 (STATUS FOR-RENT) (LOCATION SOUTH-SIDE) (NUMBER-OF-BEDROOMS 2)
     (AVAILABLE-SERVICES PRIMARY-SCHOOL SECONDARY-SCHOOL) (PETS-ALLOWED NO) (RENT 1500.0))
```

Let's now define another client that is a student, has a kid of age 11, an income of 900$ and has pets.

```
>(cl-create-ind 'kid9 '(and person (fills age 11)))
@kid9

> (cl-create-ind 'ind20 '(and client student (fills pets 'yes) (fills interested-in 'renting)
                                     (fills family-members w2 kid9)
                                     (fills income 900)
   (fills proposable-house ph20)))

>(cl-ind-close-role @ind20 @family-members)
@ind20

>(cl-print-ind @ind20)

@i{IND20} ->
  Derived Information:
    Primitive ancestors: (@c{PERSON} @c{LIVING-THING} @c{CLASSIC-THING})
    Parents: (@c{PERSON-WITH-LOW-INCOME} @c{CLIENT-FOR-RENT}
              @c{PERSON-WITH-ANIMAL} @c{PERSON-WITH-CHILDREN-IN-SECONDARY}
              @c{STUDENT})
    Ancestors: (@c{THING} @c{CLASSIC-THING} @c{LIVING-THING}
```

[8]The function "list-proposable-houses" is defined in "kb.lisp"

- Another constraint reflects the fact that a person with kids in scholar age necessitates to have a school in the neighborhood. Three categories of persons with kids in scholar age have been created: *person-with-children-in-high*, *person-with-children-in-secondary*, *person-with-children-in-primary*. The latter class, for example, is defined as

  ```
  (cl-define-concept 'person-with-children-in-secondary
                     '(and person-with-family
                     (TEST-C has-children? child-in-secondary-school-age)))
  ```

  The TEST-C in the above definition tests the family members of the client to check whether there is any of them with age between six and ten.

  The fact that a house has a primary school in its vicinities is represented inserting *primary-school* as a filler of the *available-services* role. The rule that relates the two elements is

  ```
  (cl-add-rule 'primary-school-available  @client '(all proposable-house
                                  (fills available-services primary-school))
         :filter 'person-with-children-in-primary)
  ```

- An additional rule states that if a person is a student, the proposable house must be in a university area, while another rule requires that the houses proposable to a person with animals must have the attribute *pets-allowed* set to 'yes.

Working all together, the restrictions defined above allow to eliminate from the set of available houses those whose features are incompatible with the client's feature. Some examples follows.
Let's define a client interested in renting, with a kid of age six and no pets

```
> (cl-create-ind 'kid1 '(and person (fills age 6)))
@kid1

> (cl-create-ind 'ind1 '(and client (fills interested-in 'renting)
                                  (fills income 2500)
                                  (fills pets 'no)
   (fills proposable-house ph3)
  (fills family-members w1 kid1)))
(cl-ind-close-role @ind1 @family-members)

>(cl-print-ind @ind1)
@i{IND1} ->
  Derived Information:
    Primitive ancestors: (@c{PERSON} @c{LIVING-THING} @c{CLASSIC-THING})
    Parents: (@c{PERSON-WITH-AVERAGE-INCOME} @c{CLIENT-FOR-RENT}
              @c{PERSON-WITH-CHILDREN-IN-PRIMARY})
    Ancestors: (@c{THING} @c{CLASSIC-THING} @c{LIVING-THING} @c{PERSON}
                @c{PERSON-WITH-FAMILY} @c{CLIENT} @c{PERSON-WITH-INCOME})
    Role Fillers and Restrictions
      Interested-in![1 ; 1] -> (RENTING)
                          => (ONE-OF RENTING BUYING)
      Income![1 ; 1] -> (2500)
                     => @c{INTEGER}
      Pets![1 ; 1] -> (NO)
      Proposable-house[1 ; INF] -> (@i{PH3})
                              => Primitive ancestors: (@c{INANIMATE-THING}
                                                       @c{CLASSIC-THING})
                                 Role Restrictions:
```

On the other end, creating an individual with three members in his family gives:

```
(cl-create-ind 'my-client1 '(and client-for-rent
                     (fills family-members fm2 fm3 fm4) (fills proposable-house h2)))
@my-client1

> (list-proposable-houses @my-client1)

(2B2 (STATUS FOR-RENT) (LOCATION DOWNTOWN) (NUMBER-OF-BEDROOMS 2)...)
```

Another reasonable constraint could be to define an upper bound on the number of bedrooms depending on the family size. For example, if the family size is at most 1, it is not worth it proposing a house that has more that, let's say, 3 bedrooms. Nonetheless, if we assume that the only reason that could induce a person to refuse a house with too many bedrooms is money, we can integrate this constraint in the next restriction.

- The client must have an income large enough to afford the rent (or the selling price). The best way to implement this constraint would be to define a relation between the amount of income and the rent, but this impossible in Classic.

  This cannot be done with a rule of the following type

```
(cl-add-rule 'salary-adeguate-for-rent @client
                    '(all proposable-house (TEST-C rent-adeguate-for-salary?)))
```

  since the TEST-C function cannot access the client individual, and therefore it is impossible to relate her income with the cost of the proposable-house.

  Another solution could be to add an "add-filler-rule" such as

```
(cl-add-filler-rule 'income-adeguate-for-rent @client @proposable-house
                    'income-adeguate-for-rent? :filter '(TEST-C
                                   cl-role-filled? price))
```

  When a new client (let's call it *new-c*) is created, this rule tests all the available houses and returns as fillers for *proposable-house* those houses whose cost is adeguate for *new-c*'s income. The problem with this approach is that the rule checks just for the income constraint, creating inconsistencies when, for example, we have

```
(cl-create-ind 'ind4 '(and client (fills income 2000) (fills family-members ind2)))

(cl-create-ind 'h3 '(and efficency (fills price 600)))
```

  As far as cost concern, *h3* would be fine as proposable house for *ind4*, but it is an efficency, therefore it is incompatible with the constraint that the proposable house for a client with one family member must have at least one bedroom. To pursue this last approach, it would be necessary to define a rule for each combination of house's and client's feature.

  The solution I have adopted is to define four subclasses of *person-with-income*, based on income ranges: *person-with-low-income*, *person-with-average-income*, *person-with-high-income*, *person-with-very-hig-income* along with rules that define an upper bound on the house cost for each class, such as:

```
(cl-add-rule 'low-income-and-rent @person-with-low-income
'(all proposable-house (and house-for-rent
(all rent (max 500.0))))
:filter '(fills interested-in 'renting))
```

  This solution is evidently not very satisfactory, since the grain size of the match its creates is quite coarse and the only way to improve it is to create subclasses based on smaller ranges of income. Nonetheless, it is the only one allowed by Classic's expressive power.

```
                 => @c{FLOAT}
      Basic-rent![1 ; 1] -> (300.0)
                         => @c{FLOAT}
                         .
                         .
      Status![1 ; 1] -> (FOR-RENT)
@i{2B1}
```

1 The Clients' Taxonomy

This taxonomy organizes the information about the clients of the agency. The taxonomy representes a "waiting list" of clients willing to rent or to buy a house and waiting for the right offer to become available. The classes in this hierarchy are specializations of the *person* superclass and they are built on properties that create constraints between a client and the kind of house that can be offered to him.

For example, the size of the client's family creates a lower bound on the number of bedrooms of the house he could be interested in. The client's income defines an upper bound on the cost of the house. The link between the client and houses that can suit him resides in the *proposable-house* role of the *client* concept, defined as follows [5]

```
(cl-define-concept 'client '(and person-with-income
(all proposable-house house)
             (all interested-in (one-of 'buying 'renting))))
```

The first restriction on the proposable houses' type depends upon whether the client is interested in buying or renting. This restriction is integrated in the definition of *client-for-rent* and *client-for-sale*. The latter concept, for example, is defined as

```
(cl-define-concept 'client-for-sale '(and client (fills interested-in 'buying)
                                     (all proposable-house house-on-sale)))
```

Other restrictions are create by rules that fire every time a new client enters the taxonomy[6]

- The house must have enough bedrooms to fit all the members of the client's family. This constraint is encoded in a set of rules parameterized over the family size. The following rule, for example, states that if the client has at least 1 member in her family, the proposable house must have at least 1 bedroom (that is, efficencies cannot be proposable houses for families of two persons or more)

  ```
  (cl-add-rule 'family-members-1 @client '(all proposable-house
                                       (all number-of-bedrooms (min 1)))
  :filter '(at-least 1 family-members))
  ```

  Therefore, if we create a client that has one family member and is interested in renting

  ```
  (cl-create-ind 'my-client '(and client-for-rent
                     (fills family-members fm1) (fills proposable-house h1)))
  ```

  and we ask for all the instances that satisfy the definition of the sample proposable house *h1*, we obtain[7]

  ```
  > (list-proposable-houses @my-client1)

  (1B1 (STATUS FOR-RENT) (NUMBER-OF-BEDROOMS 1) (LOCATION SHADYSIDE)(RENT 400.0)...)
  (1B2 (STATUS FOR-RENT) (LOCATION DOWNTOWN) (NUMBER-OF-BEDROOMS 1)...)
  (2B2 (STATUS FOR-RENT) (LOCATION DOWNTOWN) (NUMBER-OF-BEDROOMS 2)...)
  ```

[5] While the definition of all concepts regarding persons are in the "person-concept.cl", the definition of *client* is the the "house-concept.cl".
[6] These rules are defined in the "person-rules.cl"file
[7] All the examples presented in this work have been obtained having loaded the files "person-istances.cl" and "house-instances.cl"

let's insert in the KB a two-bedroom, average-age house locaded in Squirrel Hill.

```
>(cl-create-ind '2b1 '(and house-for-rent two-bedroom average-age-house
(fills location squirrel-hill)))
@i{2B1}
```

The new house gets classified as *house-in-urban-residential-area* and its rent is set to 1650$.

```
> (cl-print-ind @2b1)
@i{2B1} ->
  Derived Information:
    Primitive ancestors: (@c{INANIMATE-THING} @c{CLASSIC-THING})
    Parents: (@c{HOUSE-FOR-RENT} @c{TWO-BEDROOM}
              @c{HOUSE-IN-URBAN-RESIDENTIAL-AREA} @c{HOUSE-IN-UNIVERSITY-AREA}
              @c{AVERAGE-AGE-HOUSE})
    Ancestors: (@c{THING} @c{CLASSIC-THING} @c{INANIMATE-THING}
                @c{HOUSE-IN-URBAN-AREA} @c{HOUSE})
    Role Fillers and Restrictions
      Rent![1 ; 1] -> (1650.0)
                   => @c{FLOAT}
      Basic-rent![1 ; 1] -> (300.0)
                         => @c{FLOAT}
       Available-services[0 ; 9] => @c{SERVICE}
      Date-of-construction! => Tests: ((INTEGERP) (ATOM) (NUMBERP))
                               Interval: [1960 ; 1986]
      Number-of-bedrooms![1 ; 1] -> (2)
                                 => @c{INTEGER}
      Location![1 ; 1] -> (@i{SQUIRREL-HILL})
                       => @c{GEOGRAPHIC-LOCATION}
      Status![1 ; 1] -> (FOR-RENT)
@i{2B1}
```

Let's suppose that Squirrel Hill degrades to *average-level* area. To represent this fact in the knowledge base, it is necessary to delete the original value of the *area-type* role of Squirrel Hill and to assign the new value[4]. The result of performing the change is that @2B1 gets reclassified as *house-in-urban-average-level-area* and its rent is recomputed at 1050$, as shown below

```
>(change-area-definition @squirrel-hill 'average-level)

>(cl-ind-parents @squirrel-hill)
(@c{AREA-OF-PITTSBURGH} @c{URBAN-AVERAGE-LEVEL-AREA})

> (cl-print-ind @2b1)
@i{2B1} ->
  Derived Information:
    Primitive ancestors: (@c{INANIMATE-THING} @c{CLASSIC-THING})
    Parents: (@c{HOUSE-FOR-RENT} @c{TWO-BEDROOM}
              @c{HOUSE-IN-URBAN-AVERAGE-AREA} @c{AVERAGE-AGE-HOUSE})
    Ancestors: (@c{THING} @c{CLASSIC-THING} @c{INANIMATE-THING}
                @c{HOUSE-IN-URBAN-AREA} @c{HOUSE})
    Role Fillers and Restrictions
      Rent![1 ; 1] -> (1050.0)
```

[4]These operations are performed by the *change-area-definition* function, defined in the file "kb.lisp"

```
> (cl-print-ind @my-house)


@i{MY-HOUSE} ->
  Derived Information:
    Primitive ancestors: (@c{INANIMATE-THING} @c{CLASSIC-THING})
    Parents: (@c{HOUSE-FOR-RENT} @c{EFFICENCY}
              @c{HOUSE-IN-URBAN-AVERAGE-AREA} @c{HOUSE-IN-UNIVERSITY-AREA}
              @c{AVERAGE-AGE-HOUSE})
    Ancestors: (@c{THING} @c{CLASSIC-THING} @c{INANIMATE-THING}
                @c{HOUSE-IN-URBAN-AREA} @c{APARTAMENT} @c{HOUSE})
    Role Fillers and Restrictions
      Available-services[0 ; 9] => @c{SERVICE}
      Date-of-construction![1 ; 1] -> (1980)
                                   => @c{INTEGER}
      Number-of-bedrooms![1 ; 1] -> (0)
                                 => @c{INTEGER}
      Location![1 ; 1] -> (@i{SHADYSIDE})
                       => @c{GEOGRAPHIC-LOCATION}
      Basic-rent![1 ; 1] -> (100.0)
                         => @c{FLOAT}
      Rent![1 ; 1] -> (350.0)
                   => @c{FLOAT}
      Status![1 ; 1] -> (FOR-RENT)
@i{MY-HOUSE}
```

Analogous rules compute the actual rent for every combination of the *location* and the *age* features.

Re-evaluation of a house's cost

Another functionality provided by the KB is the re-evaluate the cost of a house in case the values of the evaluation parameters change. For example, it can happen that an urban area changes from poor to average-level if improvements are made or, viceversa, residential areas can be degraded to average level if poorer areas expand toward them, making the neighborhood less safe and fancy.

Location is one of the dimentions along with our houses are classified and it is used as an evaluation parameter. As the classification of an area changes, all the houses in that area are reclasssified and the cost of the house gets modified consequently (unless it had been stated by the house's owner, as described in the previous section).

Given the following definition of Squirrel Hill as a residential area

```
> (cl-print-ind @squirrel-hill)
@i{SQUIRREL-HILL} ->
  Derived Information:
    Primitive ancestors: (@c{GEOGRAPHIC-LOCATION} @c{INANIMATE-THING}
                          @c{CLASSIC-THING})
    Parents: (@c{AREA-OF-PITTSBURGH} @c{URBAN-RESIDENTIAL-AREA})
    Ancestors: (@c{THING} @c{CLASSIC-THING} @c{INANIMATE-THING}
                @c{GEOGRAPHIC-LOCATION} @c{URBAN-AREA})
    One-of Restr: (@i{OAKLAND} @i{WILKINSBURG} @i{HILL-DISTRICT}
                   @i{SOUTH-SIDE} @i{SHADYSIDE}
                   @i{SQUIRREL-HILL} @i{NORTH-HILL} @i{DOWNTOWN})
    Role Fillers and Restrictions
      Area-type[1 ; 6] -> (@i{RESIDENTIAL})
                       => @c{AREA-PROPERTY}
      Part-of-city![1 ; 1] -> (@i{PITTSBURGH})
                           => @c{CITY}
@i{SQUIRREL-HILL}
```

Determining the cost of a house

Let's consider the process of defining the renting price of a house (the process to define a selling price is analogus). Two situations are taking into consideration.

- The house's landlord wants to gain a certain profit and states the rent when he makes its house available to the agency. In this case, the stated amount is immediately assigned to the *rent* attribute of the individual inserted in the KB to represent the house to be rented.
- The landlord decides to keep the rent on the market's standards and defers to the agency its evaluation.

In the latter case, the rent is computed considering the house's features. As soon as a house enters the KB, a rule that defines its basic rent fires. The basic rent is the lowest rent that an house with a certain number of bedrooms can have[2]. For example, the following rule states that the basic rent for an efficency is 100$.

```
(cl-add-rule 'efficency-rent @efficency
'(fills basic-rent 100.0)
               :filter '(and house-for-rent (TEST-C cl-basic-rent-unfilled?)))
```

Analogous rules define the basic price for houses with a different number of bedrooms.[3]

Once the basic cost has been assigned, another rule increments it depending on the house's age and position. For example, the following rules increment the basic rent of 3.5 if a house is new and it is located in an average-level area

```
(cl-add-filler-rule 'increase-price-if-new-and-average-level
               @house-in-urban-average-level-area
               @price 'modify-basic-price :args '(3.5)
                :filter '(and new-house house-on-sale
                (TEST-C cl-basic-price-filled-and-price-unfilled?)))
```

Therefore, the definition of the following efficency for rent:

```
          (cl-create-ind 'my-house '(and efficency house-for-rent
                   (fills date-of-construction 1980)
                                 (fills location shadyside)))
```

brings to the creation of the following individual

[2]It is assumed to be the cost of an old house located in a miserable area
[3]All the rules mentioned in this section are in the file "house-rules.cl"

A Classic Knowledge Base for a Real Estate Agency

C

The Real Estate Agency Knowledge Base

The knowledge base developed manages data for an immaginary Real Estate agency. It constists of two main hierarchies: the hierarchy that organizes available houses and the taxonomy of agency's clients.

The Houses' Taxonomy

The root of the Houses' hierarchy is the concept of *house*. For the agency's purposes, a house is define by its location, the number of bedrooms, the date of construction, the available services (laundry, garage, parking lot, supermarket, schools...) Further classification is performed along dimentions defined by these properties:

- number of bedrooms. Along this dimension houses are classified as *efficency*, *one-bedroom*, *two-bedroom*, *three-bedroom*, and *more-than-three-bedroom*.
- Date-of-construction. This property defines the three categories *old-house*, *average-age-house*, *new-house*.
- Location. Fillers for this role belong to the class *geographic-location*[1]. In this work I am focusing on houses in the Pittsburgh area, that are defined as *house-in-urban-area*. A further classification is induced by the particular Pittsburgh area (Squirrel-Hill, Oakland, South Side...), where a house is located.

  Areas within a city can be classified as *residential*, *average-level*, *poor*, *miserable*, *university-area* and *bussiness-area* and a corresponding class is defined for houses located in each of these categories.
- Houses can be single houses, buildings or apartments. Since apartments are part of other houses, rules as been defined that perform inheritance along the *part-of* relation. Therefore, an apartment that is part of a building built in 1987 and located in Shadyside will automatically acquire these properties once inserted in the KB.
- Houses can be for rent or on sale. Houses on sale have a basic selling price and an actual selling price, while houses in the second category have a basic rent and an actual rent. The distinction between basic cost (price or rent) and actual cost is due to the fact that there are two ways to determine the actual cost of an house, as will be explained in the following section.

[1] The definition of this concepts and of its subconcepts can be found in the file "generic-kb.cl"