ADVENTURE GAMES IMPLEMENTATION UNDER THE PROLOG PROGRAMMING LANGUAGE

(Master Thesis)

VASILIS ZAFEIROPoulos
vzs06001@student.mdh.se

supervised by
BJÖRN LISPER

MÄLARDALEN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE AND ELECTRONICS

9/10/2008
ABSTRACT

This thesis describes the brand new concept of the Prolog Adventure Implementation technique, being a method for the implementation of adventure computer games with the use of Logic Programming (Prolog). An adventure games' environment is discrete, that is to say the player's character undertakes discrete actions such as "pick up an object", "open a door" as well as it updates its knowledge database, discovering new locations and new collectable objects, through the exploration of this environment. Therefore, considering this task, logic programming seems to be by far more efficient than conventional programming under imperative languages. The thesis, along with the implementation, demonstrates those special powerful characteristics that Prolog Adventure Implementation infuses to the game and renders it an innovating way of adventure game programming.
“I see a diorama
of the children of the world
living in peace and freedom...
No, wait! It can't be that!
It's just too dark
to make out what's in there!”

Guybrush Threepwood
(looking through a keyhole)
THE CURSE OF MONKEY ISLAND
# CONTENTS

1. Background and Motivation ................................................................. 7

2. Historical References and Related Work ........................................... 7

3. A Brief Outline of an Adventure Game ................................................. 8

4. Theoretical Framework ....................................................................... 9

5. A Short Tour of Prolog ....................................................................... 10
   5.1. Predicates ..................................................................................... 10
   5.2. Rules .......................................................................................... 13
   5.3. Write and Read .......................................................................... 15
   5.4. Unification and Release .............................................................. 15
   5.5. Learning and Forgetting ............................................................. 16
   5.6. Flow-Control ............................................................................ 17
   5.7. Negation and Failure .................................................................. 18
   5.8. Recursion .................................................................................. 20
   5.9. Modules ..................................................................................... 20
   5.10. Miscellaneous ........................................................................... 21

6. SWI-Prolog ......................................................................................... 21
   6.1. Why SWI-Prolog? ...................................................................... 21
   6.2. SWI-Prolog’s OOP Extension ..................................................... 22

7. The Prolog Adventure Implementation .................................................. 25
   7.1. The Realistic Object-Orientation ................................................. 26
   7.2. The States Diagram .................................................................... 28
   7.3. The Files Specialization ............................................................... 31
   7.4. The Ego Character ...................................................................... 33
1. BACKGROUND AND MOTIVATION

Adventure games have been a special kind of computer games in the computer games history. Even though they are not as popular as action, strategy or R.P.G. games, they still have a fanatic audience of all ages. Since they involve puzzles that the player must solve and consequently, require intelligence rather than reflexes or strategic thinking, they are popular among adults as well.

In general, in an adventure game the player controls a character which has to explore a place, collect objects, combine those objects together and solve puzzles in order to proceed to the next level. The first adventure games ever in computer games history were text-based, that is to say, they had no graphics and the player interacted with the game through an interface of text input and output. Later, adventure games incorporated graphics; in the short run text-based games were enriched by some descriptive pictures; in the long run the contemporary spectacular 3D adventure games were produced, like Syberia, Monkey Island and Runway. [1]

The handling of graphics of the adventure games does not really need to be done from scratch. Instead, there are plenty of adventure games engines which produce even realistic 3D graphics in a very easy way. As for the logical control of the game, this is handled most of the time by a conventional object-oriented script-programming language. [1]

However, there are two major drawbacks concerning the use of the script language. Firstly, it requires a quite high level of programming skills, making this way the programming irreproachable for most amateur programmers. Secondly, even this conventional script language doesn’t really handle the logical control of the game in a perfect way. The multiple nested if-then-else loops that need to be used under the script language aggravate the code but still don’t tackle all special cases. One can easily realize playing an adventure game -even one of most recently released- some logical “errors”, e.g. a new action being triggered before the necessary predecessive ones having been completed.

Contrariwise, logic programming, which apparently suits better the declarative nature of an adventure game, seems to resolve the aforementioned conflicts in an efficient way. Such is the purpose of this dissertation: to suggest an innovative pattern for the construction of an adventure game using the Prolog logic programming language.

2. HISTORICAL REFERENCES AND RELATED WORK

Prolog programming language has actually been linked to the construction of adventure games in the past. Yet, the adventure games that have been made under Prolog were
“flat” (i.e. not object-oriented) and build exclusively for teaching and illustrating purposes, or, in other words, they were intended to be used as Prolog programming manuals. [4][5][6] No attempt has been made to create a commercial graphics-based adventure game under Prolog so far.

There is no doubt that Prolog, conceptually, suits the adventure games’ environment in a better manner than a script programming language. However, no adventure games engine has been built on Prolog, as major drawbacks would rise, the most important of them being the management of the graphics. Prolog language is known to handle the graphics in a comparatively very slow way, let alone the fact that there hasn’t been any stable platform connecting Prolog with DirectX or OpenGL graphics engines yet; instead, there have been developed some experimental bindings of several different versions of Prolog with OpenGL, but not serving the purpose of handling interactive 3D graphics like those being used in modern adventure games.

Nevertheless, a variation of Logic Programming, namely Dynamic Logic Programming (DynLop), has been proposed to be used in adventure games as well as strategy and RPG ones. What Dynamic Logic Programming would be useful for in such a kind of game is the evaluation of the game’s world state, that is to say the examination of the current values referring to the variant entities constituting the game. [7][8]

Another variation of Logic Programming, that of Inductive Logic Program (ILP), has been proposed to be used in adventure game yet for a totally different purpose: the learning of an AI-agent in it, or, in other words, ILP is suggested as a machine learning technique for an agent “exposed” in an adventure game-like environment. [9][10]

### 3. A Brief Outline of an Adventure Game

An adventure game in brief revolves around an ego character, that is to say the character that the user controls. The ego character is moved by the user inside several discrete places such as rooms, houses and yards, which may be separated from one another by a door, a gate or a fence. It also encounters several discrete inanimate objects, such as tables, chairs, refrigerators, computers, trees and stones, upon which it performs several actions, such as looking, picking up, switching on and off. Among the variant objects, there are some of them that are supposed to be collected so it picks them up and places them in its inventory, that is to say its personal repository. It may as well combine two objects together, such as a lighter with a torch or a cassette with a radio.

Apart from the objects, the ego character may encounter other, animate entities like it, being people, trolls, parrots, ghosts etc. to which it may speak, give specific objects that they had asked and get some crucial hints or useful tools in return. Ego’s goal may be to discover a hidden treasure, to find a way out of a desert island, to release a friend from prison and so on.
It should be noted that in the vast majority of adventure games, the player can’t get their ego character killed or trapped in a dead-end state. While there may be two or more different finales in the story, the game can’t finish before ego has reached one of them.

4. THEORETICAL FRAMEWORK

The special property of logic programming that makes it more suitable for adventure games than the conventional imperative one is it being declarative. This means that the main emphasis is given on the facts (data, information and relations) of a problem and the rules concerning those facts, while the imperative programming languages focus on the algorithms for the solution of it.

As mentioned, an adventure game consists of several different entities, that is to say objects and characters, separate from the player’s one, the ego character. Those objects and characters have some properties. For example, the book is originally located in the bookshelf; the content of the cup is coffee etc. For declaring those properties, one can define some facts.

On each entity some actions can also be performed. For example, all entities can be looked at by the main character and a comment will be produced; the main character can begin a conversation with other person characters; the computer can be turned on and off; the cup can be used as a vessel for making a coffee; one can put only water in the boiler etc. All those actions comply with some rules considering which action is acceptable and which is not at a specific time of the game process. The red line however between the actions and the corresponding rules is rather hard – or even unnecessary – to specify. For instance, the rules about which elements and in which order can be used in the cup so as to result to a cup of coffee, ultimately describe the legal actions to be taken. We can therefore, as a matter of convention, “merge” the rules and the actions and refer to them as being the same concept.

All in all, the declaration of several facts for the properties and rules for the actions seem to be sufficient for describing the whole “logic” of the adventure game. Nevertheless, this dissertation is not restricted to the coding of mere facts and rules in Prolog. On the contrary, it introduces a novel construction method for adventure games, namely the “Prolog Adventure Implementation” technique, which claims better results than the ordinary ones in terms of clarity, complexity and error-proneness, as well as it encourages the construction of more realistic adventure games in general. But before getting into this, a brief reference to the Prolog programming language ought to be made.
5. A SHORT TOUR OF PROLOG

Prolog’s most powerful attribute is that it is a language which learns. That is to say, the user provides Prolog with data (information), and they can actually test what it had learned by asking proper questions in the listener. On its behalf, the listener is a special prompt for Prolog having the form of a question mark (‘?’).

Despite having few predefined constructs, Prolog has unlimited richness and capabilities. However, the demonstration of all of them is beyond the scope of this dissertation. Instead, only the ones necessary for the understanding of the latter will be presented.

5.1. Predicates

As mentioned before, Prolog’s difference to other programming languages lies in its declarative nature, which means that it manipulates facts rather than algorithms. In Prolog, one can declare statements like:

fruit(orange).
vehicle(car).
date_of_birth(28,december,1980).
twins(bill,john).
match(arsenal,tottenham,1,1).
it_rains.

Note that all statements as well as questions must end with a full-stop (“.”).

Those statements in Prolog actually represent some facts, which in turn are information or relationships. orange, car, bill, john, 1980 and the rest of the arguments inside the brackets are called ‘terms’ whereas the arguments outside the brackets “connecting” them, such as fruit, vehicle, date_of_birth, twins are called predicates. Terms can be classified as atoms (orange, december, bill, john) and numbers (28, 1980). it_rains is a predicate containing no terms. [2][3]

Assume that we have inserted the predicates above in Prolog’s database. If we now ask Prolog’s listener:

?- vehicle(car).

it will produce the answer:

true.

meaning that indeed, there is a fruit(orange) predicate in our database.

Similarly:
?- twins(bill,john).

produces:
true.

On the other hand, if we ask Prolog:

?- vehicle(motorcycle).

it will reply:

fail.

meaning that there is no such a fact in its facts’ database.

Note that Prolog is a case sensitive language; all terms and predicates start with a lower case. The terms can, however, take any name as long as they are enclosed in single quotes; for example 'Michael', 'old document', '2008' etc. are terms and have exactly the same properties as the terms starting with a lower case. Let us ignore the single quotes for a while and focus on the unquoted ones. If an atom starts with a capital, then it is not an atom, rather a variable. That is to say it doesn’t represent a specific fact; instead it refers to an infinitive number of facts. [2] The variables are, therefore, used as arguments in predicates being provided to the prolog’s listener. For example, the query:

?- vehicle(X).

asks for the existing vehicles and produces the output:

X=car.

while the query:

?- twins(Person,john).

asks for the twin brother of John and it gives:

Person=bill.

Moreover, the query:

?- date_of_birth(Day,Month,Year).

produces the answer:

Day=28,     
Month=december,  
Year=1980.
Considering the last example, in case we were not interested in the birthday but only in the month and the year of birth, we could query instead:

?- date_of_birth(_,Month,Year)

and the result would be:

Month=december,
Year=1980.

We have therefore encountered a new type of variable, the anonymous one ('_'), which is used when there is no need to particularize a term.

We can also have queries with the use of the same variable more than once. For example the query:

?- match(Team1,Team2,X,X).

asks which teams played against each other and had a draw, or, in Prolog's words, scored the same amount of goals, X. The answer that Prolog gives is:

Team1=arsenal
Team2=tottenham
X=1
X=1

Suppose now that our database contains the following predicates:

fruit(orange).
fruit(apple).
fruit(banana).

If we perform the query:

?- fruit(X).

Prolog's reply will be:

X=orange;
X=apple;
X=banana.

This means that Prolog didn't content itself with one single solution; instead, it exhausted all solutions justifying its reputation as a non-deterministic programming language. [1]

We mentioned in the beginning of the paragraph that a term can be either an atom or a number. For checking whether a term is either one of them, Prolog provides the
programmer with two predefined predicates, \textit{atom(\textit{term})} and \textit{number(\textit{term})}, which behaviour is obvious:

?- atom(car).
   true.

?- number(orange).
   fail.

?- number(12).
   true.

?- atom(5).
   fail.

5.2. Rules

Apart from the predicates, prolog incorporates the \textit{rules} as well. A rule is actually a combination of relations that allows us, with the help of variables, to express more general knowledge than the one we could extract from the predicates. [2]

The following example of a rule is illuminating:

\texttt{father(X,Y):-parent(X,Y),male(X).}

The rule’s \textit{head} is \texttt{father(X,Y)} while its \textit{body} is the \texttt{parent(X,Y),male(X)} part. It also defines that: \textit{X is the father of \textit{Y if X is the parent of \textit{Y and X is male.}} We may also refer to this rule as \texttt{father/2}, since it takes 2 arguments, and, in general, attach a ‘/\textit{n}’ complement to all rules, \textit{n (n\geq0)} being the number of the arguments the rule takes in total. Apart from rules, this naming convention can as well be applied to predicates.

Furthermore, we used the “and” conjunction in the previous expression because of the comma (‘,’) between \texttt{parent(X,Y)} and \texttt{male(X)} predicates. The comma therefore is the logical “and” of Prolog.

If, in our database, we had the predicates:

\texttt{parent(chris,john).}
\texttt{male(chris).}

and we asked the listener

?- father(X,john).

we would get the result:
\texttt{X=chris.}
It should be noted that a rule like the previous one, that is to say its body has different parts separated by a comma, when queried, it is being checked gradually, or, in other words, when a part fails, Prolog doesn’t bother to continue checking the remaining parts to the right.

For example, the query:

?-father(bill,john).

first checks the parent(bill,john) predicate, and as the latter fails, Prolog doesn’t proceed with checking the male(bill) one.

It should also be mentioned that we can have as many rules with the same head. For example, along with the previous father/2 rule we can have:

father(X,Y):-husband(X,Z),mother(Z,Y).

Apart from the logical “and”, Prolog has a logical “or” as well. Suppose that the database has the following predicates:

brother(richard,steven).
sister(mary,steven).

The rule below:

sibling(X,Y):-brother(X,Y);sister(X,Y).

can be translated as: \textit{X is the sibling of Y if X is a brother of Y or X is a sister of Y.}

If we now ask Prolog:

?- sibling(mary,steven).

we will get a positive answer:

true.

The logical “or” in the example above has the meaning that, while brother/2 rule fails to be verified, mother/2 doesn’t, thus sibling/2 comes true. In fact, the sibling/2 rule is equivalent to the following two rules:

sibling(X,Y):-brother(X,Y).
sibling(X,Y):-sister(X,Y).

Last but not least, we can have rules with no-valued predicates as their heads, for example:

check_if_john_and_bill_are_brothers:-sibling(john,bill);sibling(bill,john).
5.3. Write and Read

Among the few “commands” of Prolog are the write and read ones, which don’t differ to their counterparts in Pascal. The former prints an argument on the screen while the latter reads an input from the keyboard.

The following examples of write are illustrative:

?- Fruit=orange,write(Fruit).
orange
Fruit=orange.

?- write('Hello!').
Hello!
true.

Considering the first answer, one shouldn’t confuse the ‘orange’ output with the ‘Fruit=orange’ one. The former is the output from the write command while the latter is the assignment that took place.

Below follows an example for read:

?- read(Name).
|:

The ‘|:’ prompt actually waits for the user’s input. Whatever the input is, it must be always followed by a full-stop. In case we enter the argument john, we have:

?- read(Name).
|: john.
Name=john.
true.

Note also that there is a nl command used often after write, causing the cursor to change line. For example, we have:

?- write('Hi!'),nl,write('How are you?').
Hi.
How are you?
true.

5.4. Unification and Release

We saw before that ?- Fruit=orange,write(Fruit). produces the printing of the orange term on the screen. What will happen though if we try to insert the facts separately?

?- Fruit=orange.
Fruit=orange.
?- write(Fruit).
true.

It is rather obvious that the orange term becomes Fruit variable’s value only during the first query. In the second query, a memory address corresponding to that variable is shown instead, since the latter has no value. We can therefore claim that the Fruit variable is unified with the orange term in the first query, and after that it gets released.

The unification can not only take place between a variable and a term but also between two variables. The following queries therefore give:

?- X=Y,Y=water.
X=water,
Y=water.

?- X=Y.
X=Y.

The first query unifies X with Y, Y with water and, subsequently, X with water. Assuming that variables are bottles and terms are fluids, the unification of a variable (bottle) with a (term) fluid results to the former’s “filling” with the latter, whereas the unification of two variables (bottles) is equivalent to getting a “connection pipe” between them which lets the term (fluid) of the one “flow” into the other. In the second query though, the X and Y variables are “empty”, thus their unification resembles a pipeline connecting two empty bottles. [2]

If we now try to unify two terms, or even two variables that have already been unified with terms, Prolog, as a matter-of-course, won’t let this take place:

?- water=oil.
fail.
?- X=water,Y=oil,X=Y.
fail.

Nevertheless, if those two variables had been unified with the same term instead, the new unification would definitely have succeeded:

?- X=water,Y=water,X=Y.
X=water,
Y=water.

5.5. Learning and Forgetting

Although we have queried Prolog on several facts that exist in its database, we haven’t examined yet how those facts are inserted in it. This is actually done with the use of either of two basic commands, assert or consult.

assert/1 inserts a simple predicate in the database.
Suppose that Prolog still contains the *fruit/1* predicates that we met before in its database:

fruit(orange).
fruit(apple).
fruit(banana).

For example:

?- assert(fruit(cherry)).
true.

results to Prolog’s database being enriched with a new fact. If we now ask Prolog whether cherry is a fruit, we will get a positive reply:

?- fruit(cherry).
true.

The opposite command to *assert* is the *retract* one; it simply deletes a predicate from the database. For example:

?- retract(fruit(orange)).
true.
erases the *fruit(orange)* predicate and causes the following query to fail:

?- fruit(orange).
fail.

Another way to insert predicates and rules into the database is by the use of *consult*. The thing that one needs to do is to write all the predicates and rules they want in a Prolog file and then load it with the *consult* command. For example:

?- consult(fruitfile).
true.

### 5.6. Flow-Control

Although Prolog is deprived of several constructs that imperative programming languages are not, there are ways of imitating their behavior. Such a missing construct is the *if-then-else* one, which is imitated by the (*->, ;*) set of operators.

The Prolog rule below:

tall_or_short(Height):-Height>170
    -> write('Tall'),nl
    ; write('Short'),nl.
is equivalent to the following Pascal procedure:

```pascal
procedure tall_or_short(Height: integer);
begin
  if Height>170 then
    writeln('Tall')
  else
    writeln('Short')
endif;
end;
```

Note that while in Pascal `else` is optional, Prolog's `else` (`;`) is obligatory. Thus, in Prolog, when we need to form an `if-then-like` control, without the `else` option, we have to use the `true` predefined predicate, which is always true and prevents the control from failing. The following example casts light on that:

```prolog
check(Name,Age):-Age>=18
    -> register_in_electoral_system(Name)
    ; true.
```

5.7. Negation and Failure

Similarly to the imperative languages, Prolog has its own negation, `not`. Its syntax demands that the `not` operator embraces an argument in brackets while its semantics implies that the logical value of this argument is being “inverted” from true to false and vice versa. The following `not` examples are illustrative:

```prolog
unequal(X,Y):-not(X=Y).
tall(Person):-not(short(Person)).
```

Having explained the `not` operator, it’s time to proceed with the `fail` predefined predicate.

Suppose that we still have the `fruit/1` predicates in our database:

```prolog
fruit(orange).
fruit(apple).
fruit(banana).
```

Suppose also that we add the following `write_fruits/0` rule:

```prolog
write_fruits:-fruit(Fruit),write(Fruit),nl.
```

One may expect that when we call this rule in the listener:

```prolog
?- write_fruits.
```

Prolog will print all the fruits that we have declared. However, it will only print the first one:
orange.
true.

The reason is no other than the *write* command; its existence in the *write_fruits/0* rule instructs that as soon as the rule is satisfied once, Prolog doesn't need to search for other X solutions. In order to get all right answers printed, we can alter the *write_fruits* rule like this:

```
write_fruits:-fruit(Fruit),write(Fruit),nl,fail.
```

The *fail* predefined predicate forces Prolog, as soon as the latter encounters it, to repeatedly *backtrack* and search for all solutions. Now the *write_fruits/1* produces the desired result:

```
?- write_fruits.
orange
apple
banana
fail.
```

However, the prize we pay is that the rule always fails. This may seem unimportant, as we got the result we wanted, but if this rule was involved in other rules, its failure might cause the failure of the others as well. For example, when we may call the following *basic_rule/0* rule:

```
basic_rule:-write_fruits,
            write_foods,
            write_drinks.
```

this will fail and stop after having failed in *write_fruits/0*, and will never proceed to *write_foods/0* and *write_drinks/0*.

Nevertheless, there are ways of overcoming this problem, among them the use of the *not* operator. What needs to be done is to put *write_fruits/0* in a *not* bracket:

```
?- write_fruits.
orange
apple
banana
true.
```

Similarly, the *basic_rule/0* rule should contain *not(write_fruits)* instead of mere *write_fruits*. 
5.8. Recursion

One of the most powerful properties of Prolog is the recursion. Similarly to the most imperative programming languages, recursion in Prolog means in fact that a rule repeatedly calls itself.

Let us take the following example:

\[
\text{ancestor}(X,Y) : - \text{not}(X=Y),
\quad \rightarrow \text{parent}(X,Z), \text{ancestor}(Z,Y)
\; ; \; \text{true}.
\]

This recursive rule implies that \( X \) is an ancestor of \( Y \) if \( X \) is the father of \( Z \) and \( Z \) is an ancestor of \( Y \). The \text{not}(X=Y) condition is necessary in order for the recursion to proceed when the family tree hasn’t been totally exhausted yet.

5.9. Modules

Among the several constructs of Prolog, this dissertation does extensive use of modules. In brief, a module is a construct that has its own predicates and rules, which can be referred to only with its name being specified. While in traditional Prolog systems the predicate space is flat, or in other words, all predicates and rules that are in Prolog’s database are visible and directly accessible, the ones of a module are discriminated from the ones of another module as they use their modules’ names as specifications. Moreover, it is most convenient to keep a module in a separate file, and the dissertation complies with this regulation. Thus, wherever in the thesis we refer to a specific module, we imply that this module is saved in a separate file which bears the same name with it.

In order to illustrate this concept, we shall form a module file with the “family” predicates and rules encounter before:

\[
\text{:-module(family,[]).}
\text{twins(bill,john).}
\text{parent(chris,john).}
\text{brother(richard,steven).}
\text{sister(mary,steven).}
\text{male(chris).}
\text{father(X,Y) : - parent(X,Y), male(X).}
\text{father(X,Y) : - husband(X,Z), mother(Z,Y).}
\text{sibling(X,Y) : - brother(X,Y), sister(X,Y).}
\text{ancestor(X,Y) : - not(X=Y),}
\quad \rightarrow \text{parent}(X,Z), \text{ancestor}(Z,Y)
\; ; \; \text{true}.
\]

20
The first line is the definition of the *family* module’s name while the rest of the facts are the predicates and rules associated with it.

If one wants to call, for example, the *father*(bill,john) rule, they have to refer to this as: *family:father*(bill,john).

Although the module construct is very useful and its use dissolves many conflicts between the different predicates, it doesn’t achieve the desired object-orientation needed for our game. A module definitely doesn’t imitate sufficiently the *class* construct of C++ or the *object* one of Pascal. Consequently, in order to achieve the purpose of Object-Orientation, we need to use an Object-Oriented Programming extension.

### 5.10. Miscellaneous

A predefined predicate that is used often in the dissertation is the *atom_concat/3* one. Its syntax is:

```
atom_concat(<prefix>,<suffix>,<Name>).
```

while its function is rather simple; it concatenates the *<prefix>* with the *<suffix>* atoms and unifies the result with the *<Name>* variable.

For example, for the following rule, we have:

```
?- atom_concat('adventure ','_game ',Kind_of_Game).
Kind_of_Game=adventure_game.
```

### 6. SWI-PROLOG

The Prolog version that is used in the thesis is the latest one of SWI-Prolog (5.6.52). SWI-Prolog is a free software, been developed by Amsterdam University from 1990 to 2008, and despite being free, it is very powerful and has been widely used for commercial and educational applications. [11]

#### 6.1. Why SWI-Prolog?

The reason for the selection of SWI-Prolog for our project is basically two-fold.

On the one hand, SWI-Prolog has a convenient and easy-to-learn programming environment; the latter may miss many of the accessories that other Prolog versions’ GUI environments have, but considering our project, which is rather straightforward, a simple one would suffice.
On the other hand, SWI-Prolog has a powerful Object-Oriented Programming extension. Since Object-Oriented Programming is common sense for large-scale applications, several commercial or not versions of Prolog, SWI-Prolog being among them, have been given such a scope by incorporating OOP constructs. The latter’s OOP extension that this thesis is based on simply “upgrades” some modules into classes, giving this way an efficient pseudo-OOP scope to its applications.

6.2. SWI-Prolog’s OOP Extension

SWI-Prolog’s Object-Oriented predominant extension has been developed by Mauro di Nuzzo in 2007, and it is downloadable for free from SWI-Prolog’s official website. It concerns only one file (oop.pl) which, when loaded with the consult command, the programmer is allowed to manipulate special kind of modules in an OOP-like way.

For the purpose of illustrating this OOP technique, we shall demonstrate the following example.

Suppose we have a bookshop and we want to keep record of our books. Each book has a specific code with which we can refer to it. Moreover, it has the attributes of title, author, price and ISBN code. In order therefore to create a class for the books, we define a book_class file containing the homonymic following module:

```prolog
:-module(book_class,[[).]

title(Title):-atom(Title).
author(Author):-atom(Author).
price(Price):-number(Price).

set_title(Title):-this(This),reassert(This:title(Title)).
set_author(Author):-this(This),reassert(This:author(Author)).
set_price(Price):-this(This),reassert(This:price(Price)).
set_isbn(ISBN):-this(This),reassert(This:isbn(ISBN)).

write('Enter title of book: '),read(T),set_title(T),
write('Enter author of book: '),read(A),set_author(A),
write('Enter price of book: '),read(P),set_price(P).

show_info:-this(This),write('Code: '),write(This),nl,
This:title(T),write('Title: '),write(T),nl,
This:author(A),write('Author: '),write(A),nl,
This:price(P),write('Price: '),write(P),nl,
```

Along with loading this module into the database, we need to consult the oop.pl file:

?- consult(oop),consult(book_class).
% oop compiled 0.02 sec, 16,048 byte
The purpose of a class is not obvious unless we instantiate it. Before we proceed with the instantiation though, we need to cast some light on the class’s predicates and rules.

The first four predicates that are being declared are the four attributes of a book: title, author, price and isbn. Note that the predicates have been in fact defined as rules which purpose is to restrict the acceptable values for them; the first two are strictly defined as atoms whereas the last ones are defined as numbers.

The following four rules are actually assignment rules for the afore-mentioned attributes of a book; one can actually change any of those attributes using its respective assignment rule. Then comes the constructor/0 optional rule which, as its counterpart in imperative programming languages, is executed when a new instance of the class is being created. The constructor in our module simply asks the user to type in the attributes of the newly-created instance. Finally comes a show_info/1 rule which shows the current values of a book’s attributes. It should be mentioned again that those rules concern the instances of the class, and not the module prototype of it.

Considering book_class’s rules, some explanation about the this/1 and reassert/1 predicates, which were created when oop_file was consulted, must be given. The this(This) predicate does exactly what its counterpart in C++ does; it assigns the This variable with the name of the class’s instance. As for reassert/1, it has the powerful skill of assigning a predicate with a unique argument; in fact, it retracts all the previous arguments from Prolog’s database concerning that predicate and asserts the new predicate defined. This way, the programmer is capable of using some predicates exactly like conventional programming languages’ variables, for storing attribute values that are going to be used later on.

It is now time to instantiate book_class. The instantiation is done by using the instance_of/2 rule, deriving from oop_file as well, which first argument refers to the instance to be created from the second argument being a class module. Since there is a constructor rule in the latter, when the instance_of/1 rule is called in the listener, the constructor/1 one will also be invoked:

?- instance_of(clf5684, book_class).
Enter title of book: 'The Trial'.
Enter author of book: 'Franz Kafka'.
Enter price of book: 30.
true.

In case we want to call the show_info/0 rule of our instance, we have to refer to the instance name as well, exactly as we do when dealing with an ordinary module:

?- clf5684: show_info.
Code: clf5684
Title: The Trial
Author: Franz Kafka
In order to check how the different instances are related to the different classes, we use the `current_instance/2` rule, which is another rule defined by `oop.file`. The `current_instance/2` rule can be used in all four possible ways, considering the arguments it takes:

?- current_instance(clf5684,book_class).
true.

?- current_instance(clf5684,Class).
Class = book_class.

?- current_instance(Instance,book_class).
Instance = clf5684.

?- current_instance(Instance,Class).
Instance = clf5684,
Class = book_class.

In case, finally, we want to make a discount and reduce the price of Kafka’s book, we only need to invoke its `set_price/1` assignment rule with the new price as its argument:

?- clf5684:set_price(20).
true.

The `reassert/1` rule in `set_price/1` accomplished its purpose; it retracted the old `price(30)` predicate from `clf5684` instance and asserted the `price(20)` one. The call of `clf5684`'s `show_info/0` rule confirms that:

?- clf5684:show_info.
Code: clf5684
Title: The Trial
Author: Franz Kafka
Price: 20
ISBN: 9789600646008
true.

Thus, `price/1`, as well as the other attribute predicates of the instance, have each time unique values and can be used as the latter’s internal variables, contributing to an OOP prospect, similar to imperative programming languages’ one.

Yet, there is a drawback concerning this OOP extension of Prolog; the “overloading” of rules or predicates is not allowed, or, in other words, we can only have a unique rule with a specific head. In case we had more than one rules with identical heads, only the first one in the row would be recognized. In the particular cases therefore that we might need to have two rules with the same head, we construct one rule instead using Prolog’s if-then-else, that is to say the (-> , ;) set of operators.
7. The Prolog Adventure Implementation

The Prolog Adventure Implementation is a special technique of constructing adventure games using the Prolog programming language as a basis. This dissertation addresses specifically the construction of a text-based adventure game under Prolog using this Prolog Adventure Implementation technique, which, as mentioned in Chapter 4, claims best results in terms of clarity, complexity and error-proneness, as well as it leads to a more realistic game outcome, while it doesn’t neglect to give the prospect of being useful for the construction of graphics-based adventure games in the long run.

The Prolog Adventure Implementation complies with the main philosophy of the majority of adventure games considering the legal actions on each entity, that is to say an object or character participating in the game.

In most adventure games, four actions are allowed to be performed on an entity:

- **LOOK**, i.e. look at an entity and make a comment
- **TALK**, i.e. talk to an entity and initiate a dialog process (not valid for objects)
- **USE**, i.e. use an entity alone (not valid for characters)
- **USE WITH**, i.e. use an entity together with another entity

Those four actions – as well as some auxiliary ones developed exclusively for the text-based version of our game – are therefore those which will be provided by the user to the Prolog listener. In a graphics-based adventure game, however, those four actions would be triggered by a click of a particular mouse button on an entity’s surface.

Prolog Adventure Implementation also combines the following three basic attributes:

- The **Realistic Object-Orientation**
- The **States Diagram**
- The **Files Specialization**

There is also a fourth special attribute concerning specifically the person characters’ dialogues, called the **Dialogue Set Allocation**.

Before going any forward with the demonstration of those attributes as well as the overall technique, we need to make brief mention of the scenario of the game upon which this demonstration is going to take place. Our game revolves in brief around a student called Peter (our ego character) who, after managing to combat his terrible headache, he starts seeking a book to study for the next day’s exam, or get just a copy of it, and for that purpose he bothers his mates at the student residence. The scenario is demonstrated in detail in Appendix A.
7.1. The Realistic Object-Orientation

Complying with the predominant game-making philosophy of object-orientation, the Prolog Adventure Implementation suggests the instantiation of the several different objects and characters under specific classes.

As in most oop programming cases, there is an inverse proportionality between abstraction and details. In order to achieve great detail, a programmer is tempted to create as many classes as possible, each one of them having just one instance.

Considering our implementation, the creation of separate modules for each discrete object would result for example to the creation of two separate modules for the object instances of ‘box of coffee’ and ‘jar of sugar’, which both of them serve, actually, the same purpose, that of storage of something. But in order to distinguish between those two objects, one needs to define a predicate under the name of content, which can store the argument of ‘coffee’ in the first case and the one of ‘sugar’ in the second. The inclusion of such predicates as the content’s one is in fact a powerful characteristic of the Realistic Object-Orientation.

Apart from the definition of a predicate such as content, the File Specialization attribute contributes as well to the overcoming of the indirect proportionality between abstraction and detail. However, we are going to deal with it no earlier than two paragraphs ahead.

Along with the predicate of content, several other predicates are introduced, such as function (the purpose that an object serves), place (the place that an object is located), takeability (the fact whether we can pick up an object or not), usability (the fact whether we can use an object with another one or not) etc.

Apart from the predicates, the rules for the 4 actions allowed on a specific object, LOOK, TALK, USE and USE_WITH, are also defined in its module. Other rules, corresponding to auxiliary actions on an object, other than the 4 basic ones, and that the player has not right to invoke them directly, may exist in it as well. Such rules are the assignment ones set_content/1, set_place/1, set_function/1 etc., which define the new arguments for their respective predicates.

Before getting into any further detail, we need to make a convention for the naming of the prolog modules. In the Prolog Adventure Implementation several additional modules are used, which don’t serve the purpose of classes, but are used as help files for some of the instantiated objects. Thus, we shall add a ‘_class’ suffix to the name of any module which works as a class, and a ‘_file’ one to the name of any help-module file. As for the current class we are dealing with, which instantiates all the objects of the game which just serve the purpose of storage of something, we shall call it vessel_class.

It is now time to present the vessel_class module, which illuminates all of the above-mentioned concepts:
% vessel_class.pl

:-module(vessel_class,[]).

content(Content):=atom(Content).
place(Place):=atom(Place).
function(Function):=atom(Function).
takeability(Takeability):=atom(Takeability).
usability(Usability):=atom(Usability).

set_content(Content):=this(This),reassert(This:content(Content)).
set_place(Place):=this(This),reassert(This:place(Place)).
set_function(Function):=this(This),reassert(This:function(Function)).
set_takeability(Takeability):=this(This),reassert(This:takeability(Takeability)).
set_usability(Usability):=this(This),reassert(This:usability(Usability)).

constructor:=-this(This),This:set_function(storage),
             This:set_takeability(cant_take),This:set_usability(cant_use).

look:=-this(This),write('A '),write(This),nl,
       This:content(Content),
       write('It contains '),write(Content),write('.').

use:=-this(This),This:takeability(Takeability),
       (Takeability=cant_take
        -> write('I cant take it.'),nl
        ; not(This:place(inventory))
        -> This:set_place(inventory),
        write('I took the '),write(This),write('.'),nl
        ; write('I cant do that.'))).

use_with(_):=fail.

One may notice in the code above that the use_with/1 rule is set to always fail. This is actually what we aim at considering the vessel_class instances, since there is no intention to use any other object on them. Nevertheless, we do expect to use some objects with them, or, in other words, use those vessel_class objects on other ones; for instance, we use the box of coffee on the cup in order to get raw coffee in the latter as its content. The decision in which module the use_with/1 will be implemented depends on which one of the two objects changes after their interaction. In our example, the cup is the one which changes, as its content changes from nothing to raw coffee. We have, that way, come to the conclusion that a use_with/1 rule has to be implemented in the cup instance’s prototype module, cup_class.

Some light now needs to be casted on the takeability predicate. At first, it should be mentioned that the only values which it can take as arguments are the can_take and cant_take ones. The reason that we initialize it to the cant_take argument is that most of the objects which serve a storage purpose don’t need to be picked up during the course
of the game. If an object however, needs to be picked up, we can override this definition by externally setting the `takeability` predicate’s argument to `can_take`.

Considering the `use/0` rule, it should first be mentioned that as the overloading of rules— and predicates— doesn’t work when having instantiated a new class, we use instead prolog’s if-then-else, the ( -> , ; ) construct, in order to handle the different cases. The `use/0` rule in brief, causes the picking up of an object instance of `vessel_class` as long as this is “takeable” and not already placed in the inventory; when in the inventory, the object cannot be used alone, and a suitable message is prompted to the user. The `look/0` and `talk/0` rules’ functions are quite obvious and don’t actually need to be explained.

As for the similar to the `takeability` predicate, the `usability` one, things work quite different. One the one hand, the argument values for the `usability` predicate can either be `can_use` or `cant_use`. On the other hand, unlike the `takeability` predicate which argument is checked in the `use/0` function, the argument of `usability` predicate is not checked at all in the `vessel_class` module, except for the constructor’s rule, which initializes it to `cant_use`. Although this seems strange at first sight, it makes perfect sense if one takes into consideration that the fact if an object is going to be used with another one depends totally on the overall state of the game that we are currently at; thus, whenever the object is going to be usable, this has to be set externally. In which part of the game’s code the `usability` and `takeability` arguments are to be set is a matter that will be discussed later on. For the moment, it should be made clear that the `usability` predicate concerns both `use_with/1` and `use/0` rules.

It was mentioned before that a `use_with/1` rule needs to be implemented in the `cup_class` module. But how actually is this implementation going to take place? We all know that the procedure of making a coffee involves several different actions performed in a strict order. In order to construct the `use_with/1` rule of the `cup_class` module, we first need to introduce the second major concept of the Prolog Adventure Implementation, that is to say the States Diagram.

### 7.2. The States Diagram

As mentioned, the procedure of preparing a cup of coffee will be used in order to present the States Diagram technique. We should bear in mind that the `cup` is the main object that the whole procedure is going to be based on.

The player needs to perform the following tasks in the following order:

1. add sugar to the cup
2. add a spoon of coffee to the cup
3. add some water to the cup
4. stir the watered mixture
5. add hot water to the cup
6. add milk to the cup
It must be noted also that steps 1 and 2 can interchange their order, i.e. step 1 can become step 2 and step 2 can become step 1.

The diagram below depicts neatly the different states and the transitions between them. Each state represents the current content of the cup and each transition the element added to or the action performed on this content.

![Diagram of coffee-making process](image)

Considering a set of different states, instead of programming separately each transition between them, one can specify a predicate which connects two successive states and their transition all together. Such a predicate in its general form can be the following:

\[
\text{order}(\text{initial\_state}, \text{transition\_action}, \text{final\_state}).
\]

Accordingly, in our case of the “making of coffee” procedure, the respective predicate \textit{content\_order/3} would be:

\[
\text{content\_order(\text{initial\_content}, \text{x}, \text{final\_content})}.
\]

In the predicate above, \textit{\text{initial\_content}} stands for the content of the cup before the transition \textit{\text{x}} and \textit{\text{final\_content}} for the content after \textit{\text{x}}, while \textit{\text{x}} is either the \textit{element} to be added to or the \textit{action} to be performed on the former in order to get the latter.

In all, the different instances of the \textit{order} predicate would be defined by the set of the following facts:

\[
\begin{align*}
\text{content\_order(nothing, raw\_sugar, raw\_sugar).} \\
\text{content\_order(nothing, raw\_coffee, raw\_coffee).} \\
\text{content\_order(raw\_sugar, raw\_coffee, raw\_mixture).} \\
\text{content\_order(raw\_coffee, raw\_sugar, raw\_mixture).} \\
\text{content\_order(raw\_mixture, water, watered\_mixture).} \\
\text{content\_order(watered\_mixture, stir, stirred\_mixture).} \\
\text{content\_order(stirred\_mixture, hot\_water, coffee\_without\_milk).} \\
\text{content\_order(coffee\_without\_milk, milk, coffee).}
\end{align*}
\]
Each one of the above transitions, with the exception of the 6th one, describes an addition of an element (raw_sugar, raw_coffee, water, hot_water, milk, etc.) whereas the 6th one concerns an action to be performed (stir).

However, as we know, an instance of a class module recognizes only one out of the predicates of the same name, and particularly the first one in the row. So, we need to define a separate module file (not a class module file), the cup_file.pl, in which we shall store all of the above facts and will relate it to the class module file cup_class.pl. For this, we need to define a file predicate in the cup_class module, as well as the corresponding assignment rule and let the constructor/0 rule carry out the assignation:

Thus, we have:

```
file(File):-atom(File).
set_file(File):-this(This),reassert(This:file(File)).
constructor:-File=cup_file,
    this(This),This:set_file(File).
```

Like the vessel_class module, the cup_class one has also a single-valued predicate called content, which stores the current content of the cup. For example, if the cup’s current content is the ‘raw mixture’, the predicate content(raw_mixture) is declared in our cup_class module’s database of facts. The definition of the content predicate as well as its assignment rule are shown below:

```
content(Content):-atom(Content).
set_content(Content):-this(This),reassert(This:content(Content)).
```

What we need now is to find a way to define any state change of the cup’s content according to the Element being added to or the Action being performed on it. The following set of rules in the cup_class module serves perfectly this purpose:

```
use_with(Object):-(Object:content(X) ; Object:function(X)),
    this(This),This:change_content(X),
    This:show_content.
change_content(X):-this(This),This:file(File),
    This:content(Initial_Content),
    File:content_order(Initial_Content,X,Final_Content),
    This:set_content(Final_Content).
```

As one can see, the use_with/1 rule picks out the content or function X of Object and invokes the change_content/1 rule which uses X as its argument. The checking control on the current content of the cup, the reading of the allowed actions (content_order predicate) on this current content and the potential replace of this old content by the corresponding new one take all place in the latter. Definitely we could have implemented the content replacement set of actions inside the use_with/1 and not use the
change_content/1 at all, but for the sake of clarity it is wiser to implement two different but interrelated rules for this purpose.

The show_content/0 rule that is invoked inside the use_with/1 rule’s body shouldn’t also bother the reader. It is just a simple rule which serves the purpose of printing the content of the cup, and its separate implementation is justified by the fact that it is also invoked by the look/0 action. Below come both the show_content/0 and look/0 rules:

show_content:-this(This),This:content(Content),
             write('The '),write(This),write(' contains: '),write(Content),nl.

look:-this(This),
     write('A cup.'),nl,
     This:show_content.

It should be taken into consideration that in the cup_class example, the module cup_file has been set as the default associate file for all instances of it.

This makes perfect sense because the predicates saved in the cup_file which define the procedure of making a coffee, could be used by several more instances of cup_class, as the coffee-making procedure is a trivial one and all cup instances serve the same purpose. However, in our game we don’t need to instantiate cup_class more than one time, in comparison to glass_class, of which we need to create two instances, or for door_class that we need to create many more.

It is time now to introduce the third powerful attribute of Prolog Adventure Implementation, being the Files Specialization.

7.3. The Files Specialization

We have already mentioned three cases in which a single help-module file is assigned to the corresponding class module, the ones of cup_class, glass_class and door_class, and apparently there are more. Nevertheless, there are also cases that every single instance of them needs to be associated with a unique help-module. Such is the case of person_class, i.e. the class module for other characters than the ego one in the game. As each person_class instance represents a different game character, and each character apparently initiates unique dialogues and messages, a unique file needs to be assigned to each one of them.

The following diagrams illuminate these modules-instances associations. The module files are represented by rectangles and the instances by circles.
As seen above, the only \textit{cup\_class} instance we have is associated to the unique \textit{cup\_file} help-module as well as both \textit{glass\_class} instances point at \textit{glass\_file}.

The diagram for the \textit{person\_class} instances is, however, totally different, as each instance points at a separate help-module file:

The special attributes of some classes such as the \textit{person\_class} and their related help-module files will be analyzed in later chapters.
Considering our game, a convention concerning the desired level of abstraction of the class modules has been followed; *a class module should be as abstract and general as possible*, even defined in such a way that it could be used in its entire form many times in the game or in other games as well. We actually don’t put that way the detailed description of the game entities in any danger, as the latter’s special traits which define the desired level of detail are specified in their help-module files. The reason for following this convention is our aim to produce realistic and non-deterministic adventure games, that is to say even closer to the perception of the real world, where several different objects can be used and several different actions to be taken for a particular purpose. This distinctive way of a class-module definition is illuminated throughout the demonstration of the Prolog Adventure Implementation technique.

What is missing for now is the connection of the entity instances with each other resulting to the flow of the game. This connection is accomplished by using a special class and a special help-module for our ego character, while the following diagram depicts the class’s, instance’s and help-module’s association with each other:

![Diagram](image)

### 7.4. The Ego Character

Prolog Adventure Implementation is based upon an egocentric point of view. That is to say the ego character is not just a character controlled by the user and walking around in the game environment collecting objects and talking to persons; it is rather the mind of the game, as nothing can happen unless it gets ego’s permission which is “issued” in its corresponding modules, *ego_class* and *ego_class’s* help-module file. The way that ego controls the whole course of the game through its modules shall now be presented.

#### 7.4.1. The Ego Class

To begin with *ego_class*, it first needs to be mentioned that, like all class modules, it has its own attribute predicates as well, those of *file*, *place* and *state*. 
place(Place):-atom(Place).

file(File):-atom(File).

state(State):-atom(State).

Those predicates save as their arguments the associated file, the place that the *ego_class* instance is currently located at and the state of this instance respectively. For these assignments to take place, the corresponding to those predicates assignment rules are used:

set_place(Place):-this(This),reassert(This:place(Place)).

set_file(File):-this(This),reassert(This:file(File)).

set_state(State):-this(This),reassert(This:state(State)),
                 This:file(File),File:initialize(State).

One may notice that the *set_state/1* rule is quite different to the trivial *set_place/1* and *set_file/1* assignment rules. Apart from assigning a new value to the *state/1* predicate, it also invokes the *initialize/1* rule from the associated file. This rule as well as the rest of predicates and rules in the *File* help-module will be examined in detail later on.

Another rule in *ego_class* that needs to be demonstrated is the *change_state/1* one:

change_state(Action):-this(This),This:file(File),
                      This:state(State1),
                      File:order(State1,Action,State2),
                      This:set_state(State2).

A careful reader may notice that there is a perfect similarity between the *change_state/1* rule and the *change_content/1* one in the *cup_class* module that was encountered before:

change_content(X):-this(This),This:file(File),
                   This:content(Initial_Content),
                   File:content_order(Initial_Content,X,Final_Content),
                   This:set_content(Final_Content).

The only difference in the *change_state/1* rule is that we now deal with ‘states’ – apparently considering the ego character –, and not with ‘contents’. In other respects, we have a potential transition from *State1* to *State2* according to the *Action* argument, as long as those three arguments agree with any of the *order/3* predicates that exist in the *File* help-module.

It is therefore obvious that a new States Diagram is encountered, and this time it concerns the ego character’s state, apart from the already encountered one which refers to the content of the cup instance. This ego character’s States Diagram is the backbone of the game; it represent the different game stages and the transitions from one to another
and, along with the *initialize/1* rule, it initializes and reasserts the predicate arguments of the variant objects in the game, whenever a new state is reached. The ego character’s States Diagram and its related rules will be thoroughly demonstrated in the next chapter.

Considering the ego’s rules-actions that the user can invoke, it is wise to start with the three auxiliary ones, *check_current_place/0*, *show_object_of_current_place/0* and *show_objects_of_inventory/0*, which inform the user which place the ego character is in, which objects surrounds the ego character at this place and which objects he has collected so far in his inventory respectively:

check_current_place:-this(This),This:place(Place),
    write('current place: '),write(Place),nl.

show_objects_of_current_place:-this(This),This:place(Place),
    write(Place),write(' contains:'),nl,
    not(show_objects_of(Place)).

show_objects_of_inventory:-write('inventory contains: '),nl,
    not(show_objects_of(inventory)).

The usefulness of the *not* operator remains unknown until the *show_objects_of/1* rule is demonstrated:

show_objects_of(Place):-current_instance(Object,Class),
    not(Class=ego_class),
    Object:place(Place),
    write(Object),nl,
    fail.

As one sees, this rule demonstrates all the entities (except for the ego one) which *place* attribute has the same current value as the ego’s one, that is to say all the entities lying at the same place that ego is. The *fail* predicate at the end of *show_objects_of/1* rule simply forces Prolog to do as many recursions as required so that all the desired entities have been traced and displayed. However, *fail* causes always *show_objects_of/1* rule to fail, even though the latter succeeds with printing the desired results. The *not* operator therefore comes to “invert” the failure of *show_objects_of/1* rule and make it produce a ‘true’ result instead.

We mentioned before that a very crucial purpose that the ego class serves is that it “controls” the different object and character classes in the game. It is therefore time to explain that this means that *ego_class*, which in our game is instantiated to our ego character *peter*, and *peter_file*, the help-module file related to the instance, are the modules which “stand above” the rest of the modules of the game and invoke the latter’s rules.

As discussed in the beginning, we have decided to comply with the convention that most of the contemporary adventure games follow, that is to say that the only permitted actions on the game’s objects and characters would be the *look, talk, use* and *use_with* ones. Those rules-actions are defined in *ego_class* in such a way that they correspond to the rules *look/0, talk/0, use/0* and *use_with/1* of an object.
Specifically, we have the following correspondences:

<table>
<thead>
<tr>
<th>Ego</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>look(Object)</td>
<td>Object:look</td>
</tr>
<tr>
<td>talk(Object)</td>
<td>Object:talk</td>
</tr>
<tr>
<td>use(Object)</td>
<td>Object:use</td>
</tr>
<tr>
<td>use_with(Object1, Object2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Object1:use_with(Object2)</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>Object2:use_with(Object1)</td>
</tr>
</tbody>
</table>

To give an example, in the case of the cup object, the peter:look(cup) action corresponds to the cup:look one, the peter:talk(cup) to the cup:talk, the peter:use(cup) to the cup:use and the peter:use_with(cup, box_of_coffee) to the cup:use_with(box_of_coffee).

Hence, the permitted actions that the user can call are the ego’s ones. One may also notice that the arity of an ego’s rule is by one greater than the respective object’s rule.

Having explained this crucial concept, we are allowed to proceed with demonstrating the above-mentioned ego_class’s rules. Below follow the look/1 and talk/1 ones:

look(Object):-this(This),
    current_instance(Object,Class),
    not(Class=egoclass),
    Object:place(Place),
    This:valid_position_for(Place),
    Object:look.

     talk(Object):-this(This),
    current_instance(Object,Class),
    not(Class=ego_class),
    Object:place(Place),
    This:valid_position_for(Place),
    Object:talk.

One may notice that, unlike the embedded look/0 and talk/0 rules in the different objects or person classes which take no argument, the respective rules in ego_class take the Object variable. Object can be anything, but in both rules we first check if it is an instance of any class, if yes, we check whether this class is not the ego_class as there is no point the ego character performing those action on itself, and then we call the valid_position_for/1 rule in order to check if the position of Object is a valid one.

The valid_position_for/1 rule actually checks if the position of our ego character is the same with the object’s one or if just the latter’s position is the inventory, and it is demonstrated below:

valid_position_for(Place):-this(This),This:place(Ego_Place),
    (Place=Ego_Place;Place=inventory).
If all checks come to be true, the look/0 or talk/0 rule of the Object instance is invoked.

However, if our adventure game was equipped with graphics, those multiple checks wouldn’t need to take place; the user would be able to invoke the Object instance’s rules by just clicking a mouse button upon the object’s representation on the screen. That is to say those checks are meaningful only for a text-based adventure game, like the one that this report deals with, and they exist just for the purpose of making the game more user-friendly. The additions and the modifications that need to be done on the existing text-based game in order to create a graphics-based one are discussed thoroughly in a later chapter.

Similar to ego_class’s look/1 and talk/1 rules is the use/1 rule:

use(Object):-this(This),
    current_instance(Object,Class),
    not(Class=ego_class),
    Object:place(Place),
    This:valid_position_for(Place),
    Object:usability(can_use),
    Object:use.

As one can easily see, the same checks for the validity of Object’s class and position that take place in look/0 and talk/0 rules, take place in this rule as well. Nevertheless, the check: Object:usability(can_use) also appears. This usability predicate is actually no other than the one discussed earlier, and serves the purpose of checking whether this object can be used at this stage of the game. As mentioned before, the usability predicate’s argument can be either can_use or cant_use, and it is set externally, that is to say not in the object’s class module. However, it is still not time to discuss where those settings take place, as we need to proceed with the illustration of several other rules in ego_class.

The use_with/2 rule, i.e. the rule which uses two objects together, is shown below:

use_with(Object1,Object2):-this(This),
    current_instance(Object1,Class1),current_instance(Object2,Class2),
    not(Class1=ego_class),not(Class2=ego_class),
    Object1:place(Place1),Object2:place(Place2),
    This:valid_positions_for(Place1,Place2),
    Object1:usability(can_use),Object2:usability(can_use),
    This:file(File),
    (File:priority(Class1,Class2) -> Object1:use_with(Object2)
    ; File:priority(Class2,Class1) -> Object2:use_with(Object1)
    ; fail).

As one notices, the use_with/2 rule contains the same checks for the objects’ classes and positions with the use/1 one. Nevertheless, it checks two incoming objects (Object1 and Object2), instead of one (Object) that use/1 checks. Like the checks in use/1, those multiple checks of use_with/2 don’t need to be incorporated in a graphics-based adventure game, as they are only useful for a text-based one.
One can also see a valid_positions_for/2 rule being invoked. This valid_position_for/2 rule differs to valid_position_for/1 at the fact that, instead of checking the validity of the position of a single object it checks the validity of the relative positions of two objects, Object1 and Object2:

\[
\text{valid\_positions\_for}(\text{Place1}, \text{Place2}) : - \text{this}(\text{This}), \text{This}: \text{place}(\text{Ego\_Place}),
\begin{align*}
&\text{(Place1=} \text{Ego\_Place}, \text{Place2=} \text{inventory}) \\
&\quad ; \text{(Place1=} \text{inventory}, \text{Place2=} \text{Ego\_Place}) \\
&\quad ; \text{(Place1=} \text{inventory}, \text{Place2=} \text{inventory})
\end{align*}
\]

This rule actually checks whether one of the objects lies in the inventory and the other one in the place that our ego character currently is, or if both of them are in the inventory. Again, this checking rule would be meaningless in a graphics-based adventure game.

Then comes the priority/2 predicate invoked from File. For its thorough explanation, the reader needs to wait until the next section.

As we know, File is the help-module that our ego_class instance is associated to. This association takes places in the constructor rule in ego_class, which is shown below:

\[
\text{constructor} : - \text{this}(\text{This}),
\begin{align*}
\text{atom\_concat}(\text{This},'\_file',\text{File}), \\
\text{This}: \text{set\_file}(\text{File}), \text{consult}(\text{File}), \\
\text{File}: \text{load\_modules}, \\
\text{File}: \text{starting\_place}(\text{Place}), \text{This}: \text{set\_place}(\text{Place}), \\
\text{File}: \text{starting\_state}(\text{State}), \text{This}: \text{set\_state}(\text{State}).
\end{align*}
\]

The name of the help-module is produced by concatenating the name of the ego_class instance, i.e. our ego character, and, the '_file' prefix. In our game, where the ego_class instance is peter, File's name becomes peter_file. This peter_file module is then loaded by the consult(File) invocation.

In the constructor, one can also see the following actions taking place: the invocation of the load_modules/0 rule from File as well as the reading of the arguments’ values of the starting_place/1 and starting_state/1 predicates from File and the assignment of the place/1 and state/1 predicates’ arguments with those values.

As the File help-module’s predicates and rules that we have encountered so far and have not been explained are becoming more and more, it is rather time to start shedding some light on them.

### 7.4.2. Ego Class’s Help-Module

In our game, the help-module for ego_class instance peter has been the peter_file one. For the sake of illustration, in this section we will examine the rules and predicates of this special example of peter_file, which, nevertheless, follows the general pattern of the Prolog Adventure Implementation.
7.4.2.1. The Classes’ Priority

To begin with, we shall clarify the priority/2 predicates that were met before in ego_class’s use_with/2 rule. As we have already seen, priority takes as arguments the classes that the two input objects in the use_with/2 rule belong to and accordingly, it invokes Object1’s use_with/1 rule with Object2 as argument, Object2’s use_with/1 rule with Object1 as argument or none of them:

use_with(Object1, Object2) :- this(This),
    .
    .
    (File: priority(Class1, Class2)
     -> Object1:use_with(Object2)
     ; File: priority(Class2, Class1)
     -> Object2:use_with(Object1)
     ; fail).

In other words, if Class1 precedes Class2, the use_with/1 rule from Object1 is invoked, if Class2 precedes Class1 this rule is invoked from Object2, and if none of the two classes are prioritized over the other, no action is taken at all.

Thus, one may assert that there is a partial order of the different classes considering their priorities over each other. To illuminate this concept, we need to take a look at the priority/2 predicates in peter_file:

priority(paddle_class, battery_class).
priority(clock_class, battery_class).
priority(cup_class, vessel_class).
priority(cup_class, paddle_class).
priority(glass_class, sponge_class).
priority(cup_class, paddle_class).
priority(glass_class, sponge_class).
priority(person_class, glass_class).
priority(person_class, camera_class).
priority(person_class, book_class).
priority(camera_class, book_class).
priority(printing_computer_class, camera_class).
priority(sponge_class, vessel_class).
priority(basin_class, cup_class).
priority(basin_class, glass_class).

Based on the priority/2 predicates above, we have the following diagram of the classes’ partial order:
Considering the diagram above, an arrow connecting two classes represents that the class which the arrow points at has priority over the class from which this particular arrow begins. For instance, the fact that an arrow starts from battery_class and ends at clock_class means that clock_class has priority over the battery_class; thus, when the user attempts to use together an instance of clock_class, e.g. clock, and one of battery_class, e.g. new_battery, by calling ego_class’s use_with/2 rule with clock and new_battery as input arguments, no matter what the order between those arguments is – use_with(clock,new_battery) or use_with(new_battery,clock_table) – the clock:use_with(new_battery) rule is always invoked, that is to say the use_with/1 rule in clock_class’s clock instance. We have also shown this way that the order in which we use two objects together doesn’t play any role at all, and that the successive priority controls inside the parentheses at the bottom of ego_class’s use_with/2 rule results to the invocation of the appropriate rule. However, when two classes are not directly connected to each other by an arrow, therefore none of them has priority over the other, no rule is invoked and ego_class’s use_with/1 rule always fails.

7.4.2.2. The Ego’s States Diagram

Thereafter, peter_file’s load_module/0 rule that we met before simply loads all the necessary _class and _file modules for the game.

load_modules:-consult(time_file),
            consult(door_class),consult(doors_file),
            consult(basin_class),consult(basin_file),
            consult(cup_class),consult(cup_file),
            consult(glass_class),consult(glass_file),
            consult(battery_class),
            consult(paddle_class),consult(paddle_file),
            consult(clock_class),
            consult(sponge_class),consult(sponge_file),
            consult(vessel_class),
            consult(person_class),
            consult(robert_file),consult(helen_file),consult(catherine_file),
            consult(book_class),
            consult(camera_class),consult(camera_file),
consult(printing_computer_class),
consult(papper_class),

Yet, there is a module that needs to be loaded separately, and this is no other than the ego_class one, as ego_class is the module which loads the peter_file module and the latter loads the rest of the modules, therefore there is no any loading of ego_class. For this purpose, an “umbrella” game.pl file which, among other tasks, loads the ego_class module, is defined and demonstrated in a following chapter.

Next in the peter_file, there are the starting_place/1 and starting_state/1 predicates:

starting_place(bedroom).
starting_state(woke_up).

As we saw before, the arguments of those predicates are read and saved as arguments at peter’s place/1 and state/1 predicates. What we’ve actually done is that we saved the initial argument values for place/1 and state/1 predicates in peter_file and, when the peter instance of ego_class is being created, ego_class’s constructor/0 rule causes the assignation of them with those values.

woke_up is apparently the name of the initial state for peter, meaning that he’s just waken up and needs to perform a set of actions in order to proceed to the next state. In which order those states are arranged is shown below by the set of peter_file’s order/3 predicates:

order(woke_up,drinks_coffee,drank_coffee).
order(woke_up,drinks_aspirin_dilution,drank_aspirin_dilution).
order(drank_coffee,drinks_aspirin_dilution,feels_good).
order(drank_aspirin_dilution,drinks_coffee,feels_good).
order(feels_good,doesn’t_get_book_from_robert,didn’t_get_book_from_robert).
order(didn’t_get_book_from_robert,talks_to_helen,talked_to_helen).
order(talked_to_helen,gets_instructions_from_catherine,knows_the_plan).
order(knows_the_plan,drugs_helen,got_camera).

These order/3 predicates bears perfect resemblance to the content_order/3 ones encountered before in cup_file:

content_order(nothing,raw_sugar,raw_sugar).
content_order(nothing,raw_coffee,raw_coffee).
content_order(raw_sugar,raw_coffee,raw_mixture).
content_order(raw_coffee,raw_sugar,raw_mixture).
content_order(raw_mixture,water,watered_mixture).
content_order(watered_mixture,hot_water,coffee_without_milk).
content_order(coffee_without_milk,milk,coffee).
In exactly the same manner, the first and the third argument in any of \textit{peter_file's order/3} predicates represent an initial and a final state respectively while the second argument represents the transition action from the former to the latter.

However, the last assertion is not 100% correct. The second argument in a \textit{peter_file's order/3} predicate does not represent the transition action from the initial state (the first argument) to the final one (the third argument). It rather represents the final action among a set of actions that need to be taken, in order to advance from the initial to the final state. For simplicity reasons though, we shall be omitting this clarification from now on and whenever we may refer to any transition action between two ego character’s states, we will imply that this transition action is in fact the last in a row action of a set of actions.

Moreover, as the different contents of the cup and the transitions from one to another were represented by a states diagram, so can \textit{peter_file's order/3} predicates be:

Considering the cup, in order that a transition between the different contents of the cup could take place, the second object that the cup is being used with needs to have a particular content or function, as described in the \textit{use_with/1} rule of \textit{cup_class}:

\begin{verbatim}
use_with(Object):-\{(Object:content(X) ; Object:function(X)),
     this(This),This:change_content(X),
     This:show_content.
\end{verbatim}

As for \textit{cup_class's change_content/1}, it has as well been shown and explained before:

\begin{verbatim}
change_content(X):-this(This),This:file(File),
    This:content(Initial_Content),
    File:content_order(Initial_Content,X,Final_Content),
    This:set_content(Final_Content).
\end{verbatim}
It has also been demonstrated that ego_class’s \textit{change\_state/1} rule resembles perfectly cup_class’s \textit{change\_content/1}:

\begin{verbatim}
change\_state(Action):-this(This),This:file(File),
    This:state(State1),
    File:order(State1,Action,State2),
    This:set\_state(State2).
\end{verbatim}

Consequently, the \textit{Action} variable in \textit{peter} instance’s \textit{change\_state/1} rule can be unified with one of the second in row arguments of \textit{peter\_file’s order/3} predicates, that is to say \textit{drinks\_coffee}, \textit{drinks\_aspirin\_dilution}, \textit{doesnt\_get\_book\_from\_Robert}, \textit{talks\_to\_helen}, etc.

We mentioned before that a transition from one ego state to another refers only to the last in the row action among a group of actions. This specificity, actually, can sometimes be very useful; the execution of a transition’s required actions can be carried out while another transition’s actions are executed. For instance, considering the ego’s States Diagram above, Peter may put some of the ingredients for the coffee into the cup and at the same time do the necessary actions in order to prepare the aspirin dilution.

In order now to have a transition from one \textit{peter}’s state to another, the \textit{peter:change\_state\(<\text{Action}\>\)} rule, where \textit{<Action>} can be any of the above-mentioned values, needs to be invoked from somewhere. So far we haven’t met any invocation of such a rule, in any of the modules that we have examined. It is time therefore to proceed with those rules’ definitions.

One may suspect that the \textit{drinks\_coffee} transition action is caused when we call the \textit{peter:use\(<\text{cup}\>\)} rule while the content of the cup instance is currently \textit{coffee}. Thus, we will define a proper \textit{use/0} in \textit{cup\_class} module, which, if the content of its instance is \textit{coffee}, calls the \textit{drink/1} rule from \textit{cup\_file} with \textit{coffee} as the argument of it:

\begin{verbatim}
use:-this(This),This:takeability(can\_take),
    (not(This:place(inventory)))
    -> write('I got the cup.'),nl,
    This:set\_place(inventory)
    ; (This:content(Content),This:file(File),
        (File:drinkable(Content)
         -> This:set\_content(remnants),File:drink(Content)
         ; write('No point doing that.'),nl))).
\end{verbatim}

This rule instructs the picking up of the cup if it hasn’t been picked up (i.e. its place being any other than the \textit{inventory}), and in case the cup has already been picked up, the rule checks whether the content of the cup is drinkable and if yes, it calls the \textit{drink/1} rule from \textit{cup\_file} with this particular content as its argument, setting as well the new content of the cup to be \textit{remnants} as the old content of the cup will have been drunk.

What is left now is to define the \textit{drinkable/1} predicate and \textit{drink/1} rule in the \textit{cup\_file}:
drinkable(coffee).

drink(coffee):-time_file:show('What a nice coffee!'),
    time_file:delay,
    peter:change_state(drinks_coffee).

The `time_file:show/1` and `time_file:delay/0` rules are special output predicates developed so that the game becomes user-friendlier, and are explained in detail in Appendix C.

If `peter`'s current state is `woke_up`, having the `peter:change_state(drinks_coffee)` rule being called, a transition is being made to the `drank_coffee` state, according to `peter_file`'s States Diagram. If however, `peter`'s current state is `drank_aspirin_dilution`, a transition to the `feels_good` state takes place instead.

We saw before that the `set_state/1` assignment rule in `ego_class` is quite different to the trivial assignment rules in `ego_class` as well as other class modules. In particular, `ego_class`'s `set_state/1` rule has an extra line which calls the `initialize/1` rule in the File help-module (in our case, `peter_file`):

```
set_state(State):-this(This),reassert(This:state(State)),
    This:file(File),File:initialize(State).
```

This means that whenever there is a state change for `peter`, the new state is being initialized. What this `initialize/1` rule of `peter_file` mainly does is creating the new object instances to be used from that point on and (re)setting the argument values for several of them.

The `initialize/1` rule however, as it exists in a help-module, can, unlike the rules existing in class-modules, can easily be “overloaded”. That is to say, instead of defining one single `initialize/1` rule and distinguishing between the different cases with Prolog’s if-then-else set of operators (->, ;), the less complex encoding of the different cases into separate `initialize/1` rules can be adopted. The citation of some of `initialize/1` rules illuminates that concept:

```
initialize(woke_up):-instance_of(bedroom_door,door_class),
    bedroom_door:set_usability(can_use),
    bedroom_door:set_place(bedroom),
    instance_of(kitchen_door,door_class),
    kitchen_door:set_usability(can_use),
    kitchen_door:set_place(corridor),
    instance_of(helensroom_door,door_class),
    helensroom_door:set_usability(cant_use),
    helensroom_door:set_place(corridor),
    helensroom_door:set_person(helen),
    instance_of(catherinesroom_door,door_class),
    catherinesroom_door:set_usability(cant_use),
    catherinesroom_door:set_place(corridor),
    instance_of(robertsroom_door,door_class),
    robertsroom_door:set_usability(cant_use),
```

44
robertsroom_door:set_place(corridor),
robertsroom_door:set_person(robert),
instance_of(drawer_grip,door_class),
drawer_grip:set_usability(can_use),
drawer_grip:set_place(bedroom),
instance_of(refrigerator_door,door_class),
refrigerator_door:set_usability(can_use),
refrigerator_door:set_place(kitchen),
instance_of(freezer_door,door_class),
freezer_door:set_usability(can_use),
freezer_door:set_place(kitchen),
instance_of(studentsresidence_door,door_class),
studentsresidence_door:set_usability(cant_use),
studentsresidence_door:set_place(corridor),
instance_of(basin,basin_class),
basin:set_place(kitchen),
instance_of(cup,cup_class),
cup:set_content(remnants),
cup:set_place(kitchen),
cup:set_takeability(can_take),
cup:set_usability(can_use),
instance_of(first_glass,glass_class),
first_glass:set_content(remnants),
first_glass:set_place(kitchen),
first_glass:set_takeability(can_take),
first_glass:set_usability(can_use),
instance_of(second_glass,glass_class),
second_glass:set_content(remnants),
second_glass:set_place(kitchen),
second_glass:set_takeability(can_take),
second_glass:set_usability(can_use),
instance_of(new_battery,battery_class),
new_battery:set_energy(full),
instance_of(old_battery,battery_class),
old_battery:set_energy(flat),
instance_of(paddle,paddle_class),
paddle:set_content(old_battery),
paddle:set_place(drawer),
paddle:set_takeability(can_take),
paddle:set_usability(can_use),
instance_of(clock,clock_class),
clock:set_content(new_battery),
clock:set_usability(cant_use),
clock:set_place(bedroom),
instance_of(sponge,sponge_class),
sponge:set_place(kitchen),
sponge:set_takeability(can_take),
sponge:set_usability(can_use),
instance_of(jar_of_sugar,vessel_class),
jar_of_sugar:set_content(raw_sugar),
jar_of_sugar:set_place(kitchen),
jar_of_sugar:set_usability(can_use),
instance_of(box_of_coffee,vessel_class),
  box_of_coffee:set_content(raw_coffee),
  box_of_coffee:set_place(kitchen),
  box_of_coffee:set_usability(can_use),
instance_of(tap,vessel_class),
  tap:set_content(water),
  tap:set_place(kitchen),
  tap:set_usability(can_use),
instance_of(bottle_of_water,vessel_class),
  bottle_of_water:set_content(water),
  bottle_of_water:set_place(kitchen),
  bottle_of_water:set_usability(can_use),
instance_of(water_boiler,vessel_class),
  water_boiler:set_content(hot_water),
  water_boiler:set_place(kitchen),
  water_boiler:set_usability(can_use),
instance_of(packet_of_disolvable_aspirins,vessel_class),
  packet_of_disolvable_aspirins:set_content(disolvable_aspirins),
  packet_of_disolvable_aspirins:set_place(drawer),
  packet_of_disolvable_aspirins:set_takeability(can_take),
  packet_of_disolvable_aspirins:set_usability(can_use),
instance_of(packet_of_tranquillizers,vessel_class),
  packet_of_tranquillizers:set_content(tranquillizers),
  packet_of_tranquillizers:set_place(drawer),
  packet_of_tranquillizers:set_takeability(can_take),
  packet_of_tranquillizers:set_usability(can_use),
instance_of(carton_of_milk,vessel_class),
  carton_of_milk:set_content(milk),
  carton_of_milk:set_place(refrigerator),
  carton_of_milk:set_usability(can_use),
instance_of(carton_of_orange_juice,vessel_class),
  carton_of_orange_juice:set_content(orange_juice),
  carton_of_orange_juice:set_place(refrigerator),
instance_of(carton_of_apple_juice,vessel_class),
  carton_of_apple_juice:set_content(apple_juice),
  carton_of_apple_juice:set_place(refrigerator),
instance_of(ice_cubes_case,vessel_class),
  ice_cubes_case:set_content(ice_cubes),
  ice_cubes_case:set_place(freezer),
instance_of(bottle_of_detergent,vessel_class),
  bottle_of_detergent:set_content(detergent),
  bottle_of_detergent:set_place(kitchen),
  bottle_of_detergent:set_usability(can_use),

nl,nl,
time_file:show('I have a terrible headache!'),
time_file:show('I shouldnt have drunk that much last night at the party!'),
time_file:show('This is my last day before the Economics exam and I need to study!'),
time_file:show('But I first need to drink a coffee and an aspirin in order to feel good!').
initialize(drank_coffee):-cup:set_usability(cant_use),
carton_of_milk:set_usability(cant_use),
water_boiler:set_usability(cant_use),
jar_of_sugar:set_usability(cant_use),
box_of_coffee:set_usability(cant_use),
paddle:set_usability(cant_use),
time_file:show('Now that I have drunk the coffee, I need an aspirin to feel 100% allright!').

initialize(drank_aspirin_dilution):-packet_of_disolvable_aspirins:set_usability(cant_use),
time_file:show('The aspirin was totally necessary, but my spinning head demands a coffee as well!').

initialize(feels_good):-cup:set_usability(cant_use),
first_glass:set_usability(cant_use),
second_glass:set_usability(cant_use),
carton_of_milk:set_usability(cant_use),
water_boiler:set_usability(cant_use),
jar_of_sugar:set_usability(cant_use),
box_of_coffee:set_usability(cant_use),
paddle:set_usability(cant_use),
packet_of_disolvable_aspirins:set_usability(cant_use),
time_file:delay,
time_file:show('Mission accomplished! I really feel good now!'),
time_file:show('So, I should start studying...'),
time_file:show('Lets see... I need a book on Economics that I dont have...'),
time_file:show('However, my grumpy studetmate Robert, has it!'),
time_file:show('I must go and ask him to lend it to me!'),
robertsroom_door:set_usability(can_use),
instance_of(robert,person_class),
robert:set_printname('Robert'),
instance_of(economics_book,book_class),
economics_book:set_place(robert).

As one can see, the initialize/1 rules, apart from instantiating several entities and (re)setting their predicate attributes, output some specific messages on the screen, which allow the player to realize at which state the game is as well as they provide some hints considering what they need to do later on in the game.

Now that we have illuminated all those code parts of ego_class and peter_file that had been unexplained, it is time to discuss how this ego state-change mechanism works.

7.4.2.3. The Ego’s State-Change Mechanism

Suppose that our ego character peter had originally been at the woke_up state and we have performed the appropriate set of actions which results to the cup’s content being coffee. If we now call the peter:use(cup) rule, the following actions will be successively triggered:
(1) \textit{peter:use(cup)} calls the \textit{cup:use} rule and the control passes to the \textit{cup} instance, 
(2) \textit{cup:use} calls the \textit{cup\_file:drink(coffee)} rule and the control passes to the \textit{cup\_file} module, 
(3) \textit{cup\_file:drink(coffee)} calls the \textit{peter:change\_state(drinks\_coffee)} rule and the control passes to the \textit{peter} instance, 
(4) \textit{peter:change\_state(drinks\_coffee)} calls the \textit{peter:setting\_state(drunk\_coffee)} rule which calls the \textit{peter\_file:initialize(drunk\_coffee)} one and the control now passes to the \textit{peter\_file} module, 
(5) \textit{peter\_file:initialize(drunk\_coffee)} rule calls several assignment rules such as \textit{box\_of\_coffee:set\_usability(cant\_use)}, \textit{jar\_of\_sugar:set\_usability(cant\_use)}, \textit{paddle:set\_usability(cant\_use)}, \textit{cup:set\_usability(cant\_use)}, etc., and finally, the transition and the actions triggered by it are all complete.

The following diagram depicts those five different sub-stages of the transition caused by the invocation of \textit{peter:use(cup)} rule:

![Diagram](image)

One therefore notices that an attribute of a third-party entity changes only when we have a transition from one ego state to another. By using the term third-party for the entity, we mean an entity not being directly invoked at this point. For example, considering the
peter:use(cup) rule above which triggers a specific sequence of rules – among them the peter_file:initialize(drunk_coffee) one –, the box_of_coffee entity is actually a third-party one in respect to both peter:use(cup) and peter_file:initialize(drunk_doffee) rules as it appears nowhere in them, but its usability/1 attribute value does change.

In fact, it is Prolog Adventure Implementation’s main philosophy, for the sake of clarity and simplicity, not to allow others than the initialize/1 rules in ego_class’s help-module to alter the attributes of third-party entities, that is to say the change of any of the attributes of a third-party entity is in general not permitted unless there has been a change in the ego state first. The word “in general” was used, as there are actually cases that it is much more convenient to override this convention and let an entity have its attributes changed without the ego state having changed.

For the purpose of illustrating this exceptional case, the example of paddle and clock is used. Peter, in his procedure of making a coffee, fails to get the paddle to work because the battery inside it has run out of energy; thus, trying to find a new battery for the paddle, he opens the clock, removes the existing battery from it and puts it in the paddle in place of the saved one. In order for our game to be more realistic, the battery is set to be unable to be removed until the player checks the function argument of the paddle and discovers that it doesn’t work. By the way, this configuration is in fact something missing from most conventional adventure games, resulting to the player being able to collect all objects into the inventory, even if they haven’t got any tip why they should do so. Considering the paddle and clock example, this anomaly is overcome by originally setting the usability argument of clock to cant_use and resetting it to can_use after the paddle has been checked. The resetting is carried out in the paddle’s help-module, paddle_file, and particularly by the initialize(checked) rule of it. The initialize/1 rules of paddle_class are shown below:

initialize(unchecked).
initialize(checked):-clock:set_usability(can_use).

While the first initialize/1 rule, initialize(unchecked), is simply a predicate, or, in other words, a rule that does nothing, the second one, initialize(checked), resets clock’s usability attribute to can_use.

Yet, we still need to define a rule in paddle_class that would call paddle_file’s initialize/1 rule. Taking the ego_class’s pattern into consideration, we shall define a state/1 attribute and an identical set_state/1 assignment rule for paddle_class as well:

state(State):-atom(State).
set_state(State):-this(This),reassert(This:state(State)),
  This:file(File),File:initialize(State).

However, unlike ego_class, there is no such rule as change_state/1 in paddle_class, as there is no States Diagram considering the two states, unchecked and checked, of the paddle – a definition of a States Diagram for just two states would only result to the
increase of the game’s complexity. Instead, the set_state(checked), which results to the invocation of initialize(checked) rule in paddle_file rule, is called by paddle’s use/0 rule:

use:-this(This),This:takeability(can_take),
   (not(This:place(inventory))
   -> This:set_place(inventory),
       write('I picked up the paddle.'),nl,
       This:check_function,
       This:set_state(checked)
   ; (This:content(nothing)
       -> write('I cant use it alone.'),nl
       ; This:content(Battery),
       This:remove_content(Battery),
       write('I removed the battery from the paddle.'),nl,
       write('Now it has no battery inside.'),nl)).

Let us suppose that the paddle has not been picked up (i.e. its place was not the inventory). The peter:use(paddle) rule then, invoking the paddle:use one, results to the paddle’s picking up, its function-checking by check_function/0, and finally the set_state(checked) rule.

The newly-met check_function/0 rule simply prints a suitable message according to the current value of paddle’s function attribute:

check_function:-this(This),This:function(Function),
   (Function=stir -> write('It is working.'),nl ; write('It is not working.'),nl).

In contrast with the previous five-staged set of actions cause by peter:use(cup) invocation, the actions triggered by peter:use(paddle) – in case the paddle has not been picked up – can be divided into 3 stages:

(1) peter:use(paddle) calls the paddle:use rule and the control passes to the paddle instance,
(2) paddle:use calls the set_state(checked) rule which in turn calls the paddle_file:initialize(checked) and the control passes to the paddle_file module,
(3) paddle_file:initialize(coffee) calls the clock:set_usability(can_use) rule and the 3-stages procedure comes to an end.

As for the corresponding diagram, that would be:
This overriding of the afore-mentioned philosophy of Prolog Adventure Implementation and the alteration of clock’s usability attribute without any change in the ego state, has been absolutely necessary. The reason is that there is only one attribute to be altered and a specification of a new ego state just for the paddle’s function-checking would result to a more complicated code as well as to redundant definitions of predicates and invocations of rules, let alone the fact that, due to the “rhomboid shape” at the beginning of the ego’s States Diagram, we would have to create two new ego states, one being between woke_up and drank_coffee states and the other between drank_aspirin_dilution and feels_good ones.

The paddle paradigm nevertheless, is also useful for illustrating another powerful attribute of Prolog Adventure Implementation, that of an object being the content of another object raising that way even higher the level of abstraction.

7.5. An Object inside another Object

In paddle_class’s use/0 rule, there is a remove_content/1 rule which has the variable Battery as its argument. In fact, the paddle has originally got as its content the object instance old_battery of battery_class, as well as clock has the new_battery instance inside it. Those configurations are decided by peter_file’s initialize(woke_up) rule, which, as demonstrated before, among many other instantiations and assignments contains:
Apparently, we do have an object instance of a specific class inside another object instance of a different class. Prolog’s meta-programming ability and high level of abstraction allow the handling of an object as a content of another.

As a paddle can accept only a battery as its content, the use_with/1 rule in paddle_class module is defined as follows:

```
use_with(Battery) :- current_instance(Battery, battery_class),
    this(This), This:content(Content),
    (Content = nothing
    -> This:set_content(Battery),
        write('I put the battery in the paddle.'), nl,
        This:check_function
    ; write('I cant do that. There is already a battery inside.'), nl).
```

This rule simply allows an external object to be set as the paddle’s content only if the latter is a battery and there is no other battery inside it.

Considering the remove_battery/1 rule that we mentioned before, this is:

```
remove_content(Battery) :- Battery:set_place(inventory),
    this(This), This:set_content(nothing).
```

As for the paddle’s set_content/1 rule that we have come across two times so far, this is:
The set_content/1 rule apparently doesn’t just assign the value of the Content variable to the content attribute of paddle, but it also sets the function attribute of it according to Content, and, if Content is a battery, it sets the battery’s place attribute to be paddle.

What is left now is the illustration of the set_function/1 rule of paddle_class:

The mechanism of set_function/1 rule is quite easy to understand; if the content of the paddle is an unexhausted battery (i.e. the energy attribute of the battery instance is full), the paddle’s function attribute becomes stir; otherwise it takes the argument out_of_order as its value. The stir function is actually the desired one, as it is this which causes the transition from the watered_mixture content of the cup object to the stirred_mixure one.

Apart from the “object inside another object” specificity, which definitely increases the game’s realism, there are other traits that contribute to a more realistic outcome. Such are the water’s and the glasses’ generality that have been “incorporated” into the game causing the latter to be deprived of some of its determinism in favour of realism.

7.6. Non-Determinism

As one may have noticed, two vessel_class objects instances containing water as their content have been defined, bottle_of_water and tap. This means that both instances can be used with cup, when the latter’s content is raw_mixture, in order to produce watered_mixture in it. This apparently simple concept implies that, instead of a single object with a particular property – in our paradigm its content being water – an infinitive amount of objects can also be used as long as they are equipped with this property. In practice, we can have adventure games with an extended object world, more close to the real world than the conventional ones.

Nevertheless, the glasses’ case is the one which infuses even greater non-determinism to our game. In the game’s object world, there are only two glasses, although Peter needs to prepare three drinks in total; one aspirin dilution for him, one orange juice for Catherine
and one apple juice for Helen. Since he is given the chance to clean and reuse the glasses as many times as he wants, it doesn’t matter which glass is going to be the vessel for the preparation of a specific drink. What actually matters is the content of a glass, that is to say the drink itself. For details considering the emptying and cleaning of the drinking vessels, the reader is referred to Appendix B.

7.7. The Doors

The space that an adventure game takes effects is usually divided into separate rooms. In our game, all of the rooms are connected to each other with doors, while each room has at least one door. Even the university campus area can be considered to be a room with two doors; the first being the main door of the student residence and the second door the university’s main entrance. The drawer can also be regarded as a room, its “door” being the drawer grip. Apparently, our ego character can’t move inside the drawer like he can for example inside the bedroom or the kitchen. However, while he has opened the drawer, he focuses exclusively on its content, and doesn’t turn his look away from it until he uses the drawer’s grip in order to close it; thus, we can make a convention and say that the drawer is a room like all the other ones. The same applies of course for the refrigerator and the freezer, which are also treated in the game as rooms.

The following diagram displays the overall interconnection of the different rooms that the game space is divided into; while the ellipses represent the rooms the doors are implied by the lines between them:
A door is definitely an entity like the rest ones in the game; it can be looked at, talked to (being inanimate though it is not going to reply), used alone or potentially used with together another another object. Every entity has some particular attributes, while all the entities share a specific one: the place, and the door entities can’t be an exception. Consequently, as a door is supposed to connect two rooms, the question that rises is what the door’s place is. Is it the first or the second room? Is it simultaneously both rooms, or is it none of them? In order to efficiently answer this question, the following paradigm is used.

Suppose that our ego character, Peter, is in his bedroom. One of the entities that he can see there if he looks around is his bedroom’s door. In other words, in case the `peter:show_contents_of_current_place` rule is invoked, among the entities that will be listed must be the `bedroom_door` one. We know however that the `show_contents_of_current_place/0` rule shows only the entities which place attribute’s
value coincides with the ego character’s one; in our case, this value must be bedroom. Hence, bedroom_door’s place must also be bedroom.

On the other hand, when Peter is at his floor’s corridor, he can as well see his bedroom’s door; that is to say bedroom_door’s place is now corridor. We have therefore come to the conclusion that place attribute’s value of bedroom_door changes according to what peter’s place is. Yet, if Peter goes to the kitchen, there his bedroom’s door will not be in his vicinity, the latter’s place will of course not be kitchen. Hence, one can derive a general rule: a door connecting two rooms, ‘room1’ and ‘room2’, get its place attribute value changed to ‘room1’ if the ego character comes into ‘room1’, and to ‘room2’ if it comes into ‘room2’.

Now that the concepts have been clarified, it is time to demonstrate the door_class module. To begin with, door_class has the following attribute predicate:

place(Place)::atom(Place).
usability(Usability)::atom(Usability).
file(File)::atom(File).
person(Person)::atom(Person).

while the respective (trivial) assignment rules are:

set_place(Place)::this(This),reassert(This:place(Place)).
set_usability(Usability)::this(This),reassert(This:usability(Usability)).
set_file(File)::this(This),reassert(This:file(File)).
set_person(Person)::this(This),reassert(This:person(Person)).

We have already explained the importance of the place attribute.

Next comes the usability attribute is also very important; when set to cant_use, the player doesn’t have access to the related room. During the course of our game, and specifically when there is transition from one ego state to another, several doors get their usability status changed, resulting this way to the desired gradual extension of the game’s world.

Then we have the file attribute which points at the auxiliary help-module file and finally, the person attribute, which takes as its argument the game character which is the “owner” of the room behind that door (e.g. robertsroom_door’s person attribute is Robert), or, incase there is nobody behind, person becomes nobody (e.g. kitchen’s person attribute is nobody).

Below follows door_class’s constructor/0 rule which initializes the file attribute to doors_file for all door_class’s instances, and the person one to nobody, as most of the doors in the game don’t hide any person behind them, (for the exceptional cases of characters’ rooms, the person attribute’s value is reset externally):

creator::File=doors_file,
    this(This),
    This:set_file(File),
    This:set_person(nobody).
Before continuing with demonstrating door_class's use/0 rule which apparently is the rule causing the opening of a door, we need to make a deviation and illustrate doors_file, the associate help-module file for all door_class instances:

connected(bedroom,bedroom_door,corridor).
connected(kitchen,kitchen_door,corridor).
connected(helensroom,helensroom_door,corridor).
connected(robertsroom,robertsroom_door,corridor).
connected(catherinesroom,catherinesroom_door,corridor).
connected(corridor,studentsresidence_door,campus_area).
connected(campus_area,university_door,university_corridor).
connected(university_corridor,lab_door,lab).
connected(refrigerator,refrigerator_door,kitchen).
connected(freezer,freezer_door,kitchen).
connected(drawer,drawer_grip,bedroom).

The general form of the connected/3 predicates set is:

connected(<Room1>,<Door>,<Room2>).

where <Door> is the door connecting <Room1> and <Room2> together.

The connected(bedroom,bedroom_door,corridor) predicate for example, declares that the bedroom room is connected to the corridor one via the bedroom_door door_class instance.

It's time now to present the use/0 rule of door_class, which does nothing more that instructing the opening of a door:

use:-this(This),This:file(File),
    This:place(Current_Room),This:person(Person),
    (current_instance(Person,person_class)
        -> Person:place(PersonsPlace),
        (not(Current_Room=PersonsPlace)
            -> time_file:show('KNOCK KNOCK'),write(Person),time_file:show(': Come in!')
            ; true)
        ; true),
    File:connected(Current_Room,This,New_Room),This:set_place(New_Room),
    current_instance(Ego,ego_class),Ego:set_place(New_Room),
    write('I am now at: '),write(New_Room),nl.

Let us start with the special check being performed inside this rule:

    (current_instance(Person,person_class)
        -> Person:place(PersonsPlace),
        (not(Current_Room=PersonsPlace)
            -> time_file:show('KNOCK KNOCK'),write(Person),time_file:show(': Come in!')
            ; true)
        ; true),
This particular “block of commands” checks if the **person** attribute of a door is indeed a character, i.e. an instance of **person_class**, or, in other words, if there is a character standing behind that door; if yes, it then checks if our ego character is currently inside the character’s room, and if it is not, ego knocks the door and the other character tells it to come in. In case that ego is not already in the other character’s room and it attempts to use the latter’s door, that knocking-and-replying process doesn’t take place, as, apparently, ego is exiting its mate’s room.

For example, if Peter is standing at the corridor and decides to use the door of Robert’s room, the former will first knock and the latter will tell him to come in. On the other hand, if Peter is inside Robert’s room and he wants to get out of it, he is naturally not going to knock its door.

The rest of **use/0** rule:

```
File:connected(Current_Room,This,New_Room),This:set_place(New_Room),
current_instance(Ego,ego_class),Ego:set_place(New_Room),
write('I am now at: '),write(New_Room),nl.
```

simply finds the room, namely **New_Room**, which is connected to the original room, **Current_Room**, through this specific door, **This**, and sets the ego character in **New_Room** by actually changing the latter’s **place** attribute into that. **New_Room** becomes also the new place of the door, so it can be in ego’s new vicinity.

The following example is an illuminating one considering this last block of code: Peter, who has stood in the kitchen and proceeds now with using the kitchen’s door and getting out of it, gets his **place** changed into **corridor**, while the kitchen’s door gets its own **place** changed into **corridor** too, so it can be in Peter’s new visibility scope.

What happens though when Peter wants to use the kitchen’s door in order to get out of the kitchen? One may have noticed that there is no **connected/3** predicate assuring the transition from the kitchen to the corridor. In other words, the existing **connected(kitchen,kitchen_door,corridor)** predicate is not symmetric; that is to say it doesn’t guarantee us that there is a connection in the opposite way, from the corridor to the kitchen. In order for this desired connection to be achieved, one can add a **connected(corridor,kitchen_door,kitchen)** predicate in the **doors_file** help-module file. However, the fact that we have eleven different **connected/3** predicates in **doors_file** doesn’t really suggest the construction of another 11 (in total 22) ones. Instead, a much better solution would be the addition of a rule of the same name and arity as **connected/3** predicates which would imply the latter’s symmetry. Such a rule is the following one:

```
connected(Room1,Door,Room2):-connected(Room2,Door,Room1).
```

Note, however, that the general pattern described above is only applicable for trivial games spaces with rooms connected to each other with doors in a way that no closed circle is formed between any of the rooms, in other words similar to the space of our game. In cases that we have a space in which there are rooms or places with no doors, or rooms with circular door-interconnection between them, a different approach must be
adopted. Such an approach can be for example, the specification of two door instances instead of only one referring to the door connecting two rooms, with each door’s being in one of the rooms and pointing at the other.

We mentioned before that when *peter* knocks the door of another character’s room, this character tells him to come in. Apparently, it is a discussion that is going to start between them. How this discussion is carried out as well as other traits of a person character is thoroughly demonstrated below.

### 7.8. The Person Characters

How different is a person character from an object? This is actually the pilot question that leads us to the introduction of a person character’s particular traits.

We have seen in chapter 8 that a *person_class*’s instance has a unique help-module file associated to itself. This is a matter-of-course as every single person in the game produces a unique conversation with the ego character. While presenting the *person_class* module and its instances’ help-modules, at the same time we demonstrate the special mechanism that Prolog Adventure Implementation uses for the dialogues, that is to say the *Dialogue Set Allocation* referred to in the beginning.

#### 7.8.1. The Dialogue Set Allocation

In order to demonstrate how the several different phrases in the whole *conversation* are grouped and allocated into different dialogue sets, we use the example of our ego character *peter* talking with *robert*’s person character.

At first, the conversation must be divided into sets in such a way that each one of them is “semantically cohesive” and “stand-alone”. This actually means that, although each set is either strongly or weakly related to other ones, this division secures that it “makes sense” when demonstrated alone. Using the language of science, one may define the dialogue set as the least logical unit that a conversation can be divided into.

When a conversation takes place, the player, each time after a dialogue set has been demonstrated, has to select one between some of the sets that the conversation is divided to and that “follows” the recently-demonstrated one. This automatically implies an *order* specifying how the different sets are interconnected to each other. The player’s selection of the next dialogue set is made by choosing among the first phrases of the succeeding sets and will be discussed later on.

In our game, Peter tries to get Robert’s book but the latter kindly refuses, until Peter provides himself with a camera so he can eventually borrow the book to take some pictures of it. The conversation taking place between Peter and Robert, which is demonstrated below, is stored in the *robert_file* help-module:
dialogue(0,'Peter','Hello there!').
dialogue(0,'Robert','Hello, nice to see you! What’s up?').
dialogue(0,'Peter','The usual staff...').
dialogue(0,'Peter','I was wondering...').

dialogue(1,'Peter','What about your studies?').
dialogue(1,'Robert','They are going terrible... as usual!').
dialogue(1,'Peter','Mine as well...').

dialogue(2,'Peter','You have borrowed a book on Economics from the library, haven’t you?').
dialogue(2,'Robert','Yes, sure I have!').
dialogue(2,'Peter','May I borrow it for a couple of hours, make some photocopies and then bring it back to you?').
dialogue(2,'Robert','I am sorry pal! I can’t do that!').
dialogue(2,'Robert','I haven’t studied for the whole semester, and I need to study now...').
dialogue(2,'Peter','Oh, I see...').

dialogue(3,'Peter','Hello again!').
dialogue(3,'Robert','What brings you back?').
dialogue(3,'Peter','Well...').

dialogue(4,'Peter','Please Robert, give me your book for a while! I need to make photocopies of it!').
dialogue(4,'Robert','No way! I need to study it!').
dialogue(4,'Peter','Don’t you want to have a break for lunch or something?').
dialogue(4,'Robert','This book is my only chance to pass this god damn course, and I am not going to waste it!').
dialogue(4,'Peter','Ok, Ok! Take it easy dude!').

dialogue(5,'Peter','Well, I found a solution to my problem!').
dialogue(5,'Peter','Really? Let me hear it then!').
dialogue(5,'Peter','I will just borrow your book for two minutes, take some pictures of it with my camera, and then print them at the computer lab!').
dialogue(5,'Robert','Great idea! You won’t be late at all this way!).
dialogue(5,'Robert','Here you are then, get the book!').
dialogue(5,'Peter','Thanks a lot man! I will bring back in ten minutes!).
dialogue(5,'Robert','Hey! You said two, not ten!').
dialogue(5,'Peter','Make it seven!').
dialogue(5,'Peter','Five!').
dialogue(5,'Peter','Deal!').

dialogue(6,'Peter','I am now going to take pictures of your book...').
dialogue(6,'Albert','Come on, hurry up! I need the book back to study!').
dialogue(6,'Peter','Ok, ok! I will be as fast as I can...').

dialogue(7,'Peter','I guess I will be going now...').
dialogue(7,'Robert','Ok, see you man!').
dialogue(7,'Peter','Bye bye!').

The sets that the conversation has been divided into are apparent and distinguished from each other by the numerical first argument of each dialogue/3 predicate. The other two
arguments that the dialogue/3 predicates take are the name of the person speaking and the phrase this person says.

The question that rises immediately is what the order between the different sets is. One may, at first, notice that set ‘0’ and set ‘3’ have something in common; they are both sets that a dialogue can start with. The fact that those two sets can’t be used together in the same dialogue implies that Peter and Robert may have more than one discussions. This is also obvious from the fact that the above conversation contains contradictory sets, like ‘2’ which deals with Robert’s refusal to lend his book to Peter and ‘5’ which shows that Robert eventually decides to give it away.

Thus, we have encountered the necessity of rigidly defining of two concepts, those of conversation and discussion. A conversation shall consist of many different dialogues, each one of them being performed at a different time and definitely under different circumstances, while a dialogue of this kind shall be called a discussion. Consequently, the conversation embraces all dialogue sets whereas a discussion contains only some of them. In our example, sets ‘0’, ‘1’, ‘2’, ‘7’ and ‘3’, ‘5’, ‘7’ form two different discussions between Peter and Robert, whereas both of discussions, as well as several other ones, constitute the two mates’ conversation in total.

As, apparently, set ‘7’ is always the final set of a discussion between Peter and Robert, one derives that the latter starts with either set ‘0’ or ‘3’ and ends with ‘7’. One also notices that set ‘0’ is meant to start a discussion only the very first time that Peter talks to Robert; from the second time over, every discussion starts with ‘3’. This is actually a general rule that the Dialogue Set Allocation method complies with; a discussion starts with a specific starting set – except for the first time that the conversation which it falls under takes place, whereon another specific starting set is used – and finishes with a specific finishing set as well.

As briefly mentioned before, a discussion is non-deterministic, that is to say the player has to choose which one of the several different dialogue sets following the current set they want to continue with; sometimes, however, the player has only one option, thereby the dialogue being degenerated into a deterministic one. The order in which each dialogue set follows another is also stored in person_class’s help-module file with the help of the dialogue_order/2 predicates. In Robert’s case, robert_file shall contain:

dialogue_order(0,1).
dialogue_order(0,2).
dialogue_order(1,2).
dialogue_order(1,7).
dialogue_order(2,7).
dialogue_order(3,2).
dialogue_order(3,4).
dialogue_order(3,5).
dialogue_order(3,6).
dialogue_order(4,7).
dialogue_order(5,7).
dialogue_order(6,7).
dialogue_order(7,7).

Diagrammatically, this sets’ order is depicted as follows:

From the diagram above, one ensures themselves that sets ‘0’ and ‘3’ are the sets that a discussion can start with whereas set ‘7’ is always the finishing set. Those facts are defined in the robert_file module by the following predicates:

\[
\begin{align*}
\text{start\_set}(0). \\
\text{last\_set}(7). \\
\text{new\_start\_set}(3).
\end{align*}
\]

We mentioned before that, after a set has been demonstrated, the player, in order to proceed to the next one, chooses between the first phrases of the succeeding sets. Although the first sets’ phrases could be derived directly from dialogue/3 predicates, due to Prolog’s OOP restrictions, we are forced to define a new set of predicates each one of them storing the first phrase of each dialogue set. Those predicates, namely first_phrase/2, will be:
first_phrase(1,'What about your studies?').

first_phrase(2,'You have borrowed a book on Economics from the library, havent you?').

first_phrase(4,'Please Robert, give me your book for a while! I need to make photocopies of it!').

first_phrase(5,'Well, I found a solution to my problem!').

first_phrase(6,'I m now going to take pictures of your book...').

first_phrase(7,'I guess I will be going now...').

One notices that each predicate misses the name of the character speaking out their phrase. However, as a dialogue set always starts with a phrase by the ego character, the display of the latter’s name would be trivial and, consequently, redundant.

One also sees that there are no first_phrase/2 predicates for sets ‘0’ and ‘3’, that is to say the two starting sets. This is actually justifiable, as the player would never have those sets among their available options; ‘0’ and ‘3’ can only initiate a discussion, thereby not succeeding any set.

In order now to clarify the mechanism that a player chooses a set in a discussion, the discussion consisting of sets ‘0’,'1','2','7' will be illustrated.

Since the discussion starts with set ‘0’, as soon as it starts the following dialogue will appear on the screen:

Peter: Hello there!
Robert: Hello, nice to see you! Whats up?
Peter: The usual staff...
Peter: I was wondering...

Immediately after that, the player will be prompted with the first phrases of the succeeding sets ‘1’ and ‘2’ thereby having to chose one of them in the reading prompt below them:

1. What about your studies?
2. You have borrowed a book on Economics from the library, havent you?

The selection is done by inserting the corresponding set number to a reading prompt. If the input by the user is: 1., the homonymic set will be demonstrated:

Peter: What about your studies?
Robert: They are going terrible... as usual!
Peter: Mine as well...

As set ‘1’ is succeeded by sets ‘2’ and ‘7’, the options below will follow:
2. You have borrowed a book on Economics from the library, haven't you?
7. I guess I will be going now...

In case the user inserts: /: 2. in the new prompt, set ‘2’ demonstrating Robert’s refusal to give the book will be shown:

Peter: You have borrowed a book on Economics from the library, haven't you?
Robert: Yes, sure I have!
Peter: May I borrow it for a couple of hours, make some photocopies and then bring it back to you?
Robert: I am sorry pal! I can’t do that!
Robert: I haven't studied for the whole semester, and I need to study now...
Peter: Oh, I see...

As seen in the sets’ order diagram, instead of “deviating” through set ‘1’, the user could have actually reached this point by selecting set ‘2’ immediately after set ‘0’.

In any case, no matter which “path” the user has selected to reach set ‘2’, the latter is only succeeded by set ‘7’, thus, after its demonstration, one single option is available:

7. I guess I will be going now...

This dialogue part can therefore be considered to be deterministic. The unique acceptable answer, that is to say /: 7., presents set ‘7’ and finishes the discussion:

Peter: I guess I will be going now...
Robert: Ok, see you man!
Peter: Bye bye!

Similarly to the direct connection between set ‘0’ and set ‘2’, there is a set ‘1’-set ‘7’ path as well, as one can deduct from the sets’ order diagram.

For the purpose of further clarifying this concept, the whole 0→1→2→7 dialogue of the afore-illustrated discussion is demonstrated below:

Peter: Hello there!
Robert: Hello, nice to see you! What's up?
Peter: The usual stuff...
Peter: I was wondering...
1. What about your studies?
2. You have borrowed a book on Economics from the library, haven't you?

Peter: What about your studies?
Robert: They are going terrible... as usual!
Peter: Mine as well...
2. You have borrowed a book on Economics from the library, haven't you?
7. I guess I will be going now...

64
Peter: You have borrowed a book on Economics from the library, havent you?
Robert: Yes, sure I have!
Peter: May I borrow it for a couple of hours, make some photocopies and then bring it back to you?
Robert: I am sorry pal! I cant do that!
Robert: I havent studied for the whole semester, and I need to study now...
Peter: Oh, I see...
7. I guess I will be going now...
| : 7.
Peter: I guess I will be going now...'  
Robert: Ok, see you man!  
Peter: Bye bye!
true.

The Prolog code for the dialogue mechanism shown above will be discussed later on. It should be noted, however, that the way of selecting which set shall follow each time, i.e. inserting the corresponding dialogue set’s number, is only useful for the text-based version of our game while in a potential graphics mode, a mouse-click on the suitable starting phrase would accomplish such a task.

We mentioned before that two discussions differ to each other on the fact that they take place at a different time and under different conditions. Discussions ‘0’-‘1’-‘2’-‘7’ and ‘3’-‘5’-‘7’, for example, deal with Robert’s refusal to give the book to Peter and his final making-up of his mind respectively, imply that there is a change considering Robert’s state. The latter declares explicitly that robert, along with all person_class instances, is characterized by the fact that it has different states, exactly like ego_class’s instance, peter. In fact, a person character can be considered to be an entity between an ego character and an object.

7.8.2. The Person Character’s States Diagram

The existence of different states for a person character entails the definition of the respective States Diagram. In order however to draw robert’s States Diagram, we first need to define its different states. This is achieved by defining the state_order/3 predicates in robert_file help-module:

```
state_order(beginning,talks_2,refused_to_give_book).
state_order(refused_to_give_book,peter_got_camera,can_lend_book).
state_order(can_lend_book,talks_5,gave_book).
```

As in the similar states’ predicates encountered before, the first and the third arguments of a state_order/3 predicate represent the instance’s current initial and resulting final state when the transition action represented by the second argument is applied.

The corresponding States Diagram shall therefore be:
One may notice that the first and the third transition actions are related to the sets ‘2’ and ‘5’: when Roberts speaks out set ‘2’ and set ‘5’ there is a transition from beginning to refused_to_give_book and from can_lend_book to gave_book respectively. Those “transitional” properties of sets ‘2’ and ‘5’ need for the moment to be declared in robert_file module:

```
set_which_changes_state(2).
set_which_changes_state(5).
```

We mentioned before that a discussion is distinct from the other discussions not only chronically but also accordingly to the state that our person character instance is currently at. However, we haven’t enumerated the existing discussions so far neither have we assign the different sets to a particular state; apparently, it was the missing concept of state preventing us from doing this. Now that the latter has been properly defined, we can specify the association between set and states using the state_for_set/2 predicates in robert_file:

```
state_for_set(0,beginning).
state_for_set(1,beginning).
state_for_set(2,beginning).
state_for_set(3,beginning).
state_for_set(3,refused_to_give_book).
state_for_set(3,can_lend_book).
state_for_set(3,gave_book).
state_for_set(3,waiting_book_back).
state_for_set(4,refused_to_give_book).
state_for_set(5,can_lend_book).
state_for_set(6,gave_book).
state_for_set(7,beginning).
state_for_set(7,refused_to_give_book).
state_for_set(7,can_lend_book).
state_for_set(7,gave_book).
state_for_set(7,waiting_book_back).
```
Taking both the order between the different sets and the sets’ grouping into discussions into consideration, we get the following illuminating diagram, where each coloured circle represents the discussion that corresponds to a specific state:

It’s now time to demonstrate how this overall dialogue mechanism is implemented by the **talk/0** rule — with the help of numerous auxiliary ones — in the *person_class* module.

### 7.8.3. The Person Character Class

At first, we need to examine the attribute predicates that *person_class* contains. Those predicates are demonstrated below:

- `start_set(Start_Set):-integer(Start_Set).` % to be set in the associated file
- `last_set(Last_Set):-integer(Last_Set).` % to be set in the associated file
- `saved_set(Saved_Set):-integer(Saved_Set).`

- `state(State):-atom(State).`
talk_first_time(Talk_First_Time):-atom(Talk_First_Time).

place(Place):-atom(Place).
file(File):-atom(File).
usability(Usability):-atom(Usability).

while their corresponding assignment rules are:

set_start_set(Start_Set):-this(This),reassert(This:start_set(Start_Set)).
set_last_set(Last_Set):-this(This),reassert(This:last_set(Last_Set)).
save_set(Saved_Set):-this(This),reassert(This:saved_set(Saved_Set)).

set_state(State):-this(This),reassert(This:state(State)),
This:file(File),File:initialize(State).

set_talk_first_time(Talk_First_Time):-this(This),
reassert(This:talk_first_time(Talk_First_Time)).

set_place(Place):-this(This),reassert(This:place(Place)).
set_file(File):-this(This),reassert(This:file(File)).
set_usability(Usability):-this(This),reassert(This:usability(Usability)).

Apparently, start_set/1 and last_set/1 attribute predicates store the starting and finishing sets defined in person_class’s help-module files respectively. saved_set/1 predicate’s function though is totally different; its purpose is to save a set capable of changing the instance’s state, like sets ‘2’ and ‘5’ in our paradigm, and it will be explained thoroughly later on.

Next comes the state/1 predicate which looks exactly like the homonymic predicate of ego_class. Its corresponding assignment rule also calls the initialize/1 one from the help-module which will be illustrated during the demonstration of person_class’s talk/0 rule.

Then we have talk_first_time/1 predicate which can take only two argument values, yes and no. This boolean-like predicate’s argument must originally be set to yes, and, as soon as our ego character talks for the first time with this person, the predicate’s argument becomes no. As for its usefulness, this is rather obvious; when talk_first_time/1’s argument is yes, the discussion starts with the original starting set defined in person_class’s help-module, while when it becomes no, it starts with the new starting set also defined in the same file. In robert’s case, for example, when talk_first_time’s argument is yes the starting set is ‘0’ and will change to ‘3’ when the former changes to no.

Finally, we have the trivial attribute predicates of place/1, file/1 and usability/1. While place/1 and file/1 representing the place and the associate help-module file of person_class’s instance make perfect sense, usability/1’s purpose may seem a little obscure. In fact, it concerns the use_with/1 rule of the class-module; using an object with a person character amounts to giving this object to the person.
Before proceeding with talk/0, we should first demonstrate the auxiliary rules that talk/0 invokes. But even before them, the constructor/0 rule shall be illustrated:

constructor:-this(This),
    atom_concat(This,'_file',File),
    This:set_file(File),
    File:last_set(Last_Set),This:set_last_set(Last_Set),
    File:start_set(Start_Set),This:set_start_set(Start_Set),
    This:set_talk_first_time(yes),This:save_set(0),
    atom_concat(This,'sroom',Place),This:set_place(Place),
    State=beginning,
    This:set_state(State),
    This:set_usability(cant_use).

In brief, the constructor implies that when an instance is created, among the tasks that take place, its help-module is assigned to be the file with the same prefix and the '_file' suffix, first and last sets are being read from the help-module, place is set to be corresponding room, state to be beginning and saved_set to be 0. The latter assignation is deliberately improper; set '0' is definitely not a set that can change robert's state, or any instance's state, as it is the very first dialogue set of the conversation. Such an assignation is done in order that saved_set/1 attribute will have whichever argument value and render Prolog's compiler capable of carrying out the several checks on it while the selection of this value (0) secures that no change of state is accidentally going to take place.

Where en passant is this saved_set/1 predicate going to be useful? The following person_class's auxiliary rule gives us a hint:

check_if_set_changes_state(Current_Set):-this(This),This:file(File),
    (File:set_which_changes_state(Current_Set)
    -> This:save_set(Current_Set)
    ; true).

This rule simply checks if the incoming set can change a state of person_class's instance, by checking the set_which_changes_state/1 predicate in the help-module file, and if it can, it is being stored as the instance's saved_set. In robert's paradigm, robert_file's set_which_changes_state/1 predicates that we have encountered before imply that only set '2' or '5' could be saved as saved set.

The saved set's necessity is to make person_class instance advance to a next state, like set '2' and '5' can let Robert do so, as shown by the first and the third transition of his States Diagram. This purpose is fulfilled by the following person_class's auxiliary rule:

advance_to_next_state(Saved_Set):-this(This),
    atom_concat('talks_',Saved_Set,Action),
    This:change_state(Action).

Afterwards, we have the following two rules, which function is rather apparent:
check_if_first_talk:-this(This),
   (This:talk_first_time(yes)
   -> This:reset_start_set,
       set_talk_first_time(no)
   ; true).

reset_start_set:-this(This),This:file(File),
   File:new_start_set(New_Start_Set),
   This:set_start_set(New_Start_Set).

check_if_first_talk/0 rule checks whether it is the first time that the ego character speaks to our instance, and if yes, it calls the reset_start_set/0 which substitutes the start_set argument value; in robert’s paradigm, the latter changes from 0 to 3.

Then come the rules responsible for demonstrating the dialogue sets and the following sets’ first phrases on the screen:

proceed(Current_Set):-this(This),This:file(File),
   File:dialogue(Current_Set,Name,Text),
   write(Name),write(': '),time_file:show(Text),
   fail.

show_options(Current_Set):-this(This),This:file(File),This:state(State),
   File:dialogue_order(Current_Set,Next_Set),
   File:state_for_set(Next_Set,State),
   File:first_phrase(Next_Set,Text),
   write(Next_Set),write('. '),write(Text),nl,
   fail.

proceed/1 rule gets as its unique input argument the number label of the current dialogue set and displays the latter. show_options/1 finds the following sets to the current one that it also accepts as its input argument, checks if the current state of our instance allows each of those sets, and prints sets’ initial phrases. Both rules have the fail predefined predicate in the end, since they need to do as many recursions as required in order to cover all their phrases.

Nevertheless, due to the fail predicate, those two rules always fail. Thus, when they are invoked, they need to be accompanied by a not operator so that they avoid failure. The rule that carries out the invocation is the discuss/1 one and is shown below:

discuss(Current_Set):-this(This),
   This:last_set(Last_Set),
   not(This:proceed(Current_Set)),
   This:check_if_set_changes_state(Current_Set),
   (not(Current_Set=Last_Set)
   -> not(This:show_options(Current_Set)),
       read(Next_Set),
       This:discuss(Next_Set)
   ; true).
In brief, the \textit{discuss/1} rule, which takes the current dialogue set as its argument, starts with displaying the whole set, and if this set is not the last one, it performs the following actions: it shows the available options, that is to say the first phrases of the succeeding states, it then asks the user to insert the set label of their choice, and finally, it recursively calls itself using the chosen set as argument this time. It also performs a check whether each current set is a set capable of performing a state change on the instance by calling the \textit{check\_if\_set\_changes\_state(Current\_Set)} rule; the latter, as we have seen before, if true, it results to the saving of \textit{Current\_Set} into the \textit{saved set} attribute. How this \textit{saved set} is going to be used will be illustrated soon.

It is obvious that this \textit{discuss/1} rule demonstrates the whole dialogue starting from the set that it is taken as argument until the finishing set is reached. Hence, one may infer that the call of the \textit{discuss/1} rule, having as argument the starting set, would result to the accomplishment of a specific discussion. We have therefore arrived at \textit{talk/0} action-rule, which, among other tasks of it, invokes the \textit{discuss/1} rule with the starting set as its argument. \textit{talk/0} will specifically be:

\begin{verbatim}
talk:-this(This),This:start_set(Start_Set),
   discuss(Start_Set),
   This:check\_if\_first\_talk,
   This:saved\_set(Saved\_Set),
   nl,
   (This:advance\_to\_next\_state(Saved\_Set);true).
\end{verbatim}

As shown, \textit{talk/0} basically generates a full discussion by calling the \textit{discuss/1} rule with the starting set as its argument; then it checks if this was the first time that the person character talked to the ego one by calling the \textit{check\_if\_first\_talk/0} rule, which, as we have already examined, changes, if required, the starting set; and finally, it attempts to carry out a transition to the next state according to the saved set in the \textit{saved\_set/1} attribute, by calling the \textit{advance\_to\_next\_state/1} rule with the latter as its argument.

The reason that we have used the following block in the \textit{talk/0} rule:

\begin{verbatim}
(This:advance\_to\_next\_state(Saved\_Set);true)
\end{verbatim}

is that the \textit{advance\_to\_next\_state/1} rule may sometimes fail, since not all saved sets are capable of causing a state transition; thus, in order that \textit{talk/0} won’t fail together with \textit{advance\_to\_next\_state/1}, a \texttt{true} predefined predicate is used for “rectifying” the latter when it fails.

It was mentioned before that \textit{person\_class’s use\_with/1} rule is used in order for ego to give an object to a person. \textit{use\_with/1} rule, which resembles its counterpart in \textit{cup\_class}, which we have examined before, is shown below:

\begin{verbatim}
use\_with(Object):-this(This),
   Object:tag(X),
   atom\_concat('gets\_',X,Action),
   This:change\_state(Action),
   Object:set\_place(This).
\end{verbatim}
In all, there are three special traits characterizing this rule. Firstly, it uses the *tag* attribute of the object being used with; furthermore, it potentially causes a transition from one person’s state to another, lastly, it sets object’s place to be the person instance in question.

Considering the *tag* attribute, we haven’t encountered such a predicate in any of the class-modules that we have examined so far. In fact, *glass_class*, *book_class* and *camera_class* are the only class-modules having those attributes, as their instances are the only ones which are going to be used with (i.e. given to) person characters. The *tag* attribute of an object, in brief, stores a friendly name of it. For instance, the *tag* of an instance of *book_class* is the same as the instance’s name, whereas the tag of a *glass_class* instance is defined according to its content; this way *economics_book*’s *tag* is also *economics_book*, while *first_glass*’s one may be *glass_with_orange_juice*, *glass_with_water* etc.

### 7.8.4. The Person Character’s Initializations

As one may have noticed in *robert*’s States Diagram, the last transition action is the *gets_economics_book* one. Apparently, this action is produced by the afore-mentioned *atom_concat/3* predefined rule in the *use_with/1* one. Hence, in general, for a person character to advance *itself* to a new state of it, two means are possible; selecting a specific dialogue set in a discussion or using the person character instance with a particular object. We used the word “*itself*” because those two kinds of transitions are invoked when calling this person instance’s rules, either the *talk/0* or the *use/1* one. In fact, there is a third way of causing a transition to happen; that of *ego_class*’s help-module externally calling a transition action of *person_class*’s instance’s States Diagram. In Robert’s paradigm, the transition actions of *peter_got_camera* and *peter_took_pictures_of_book* fall into this category, and they lie respectively inside the following *peter_file*’s *initialize/1* rules:

```
initialize(got_camera):-first_glass:set_usability(cant_use),
            second_glass:set_usability(cant_use),
            carton_of_orange_juice:set_usability(cant_use),
            carton_of_apple_juice:set_usability(cant_use),
            ice_cubes_case:set_usability(cant_use),
            robert:change_state(peter_got_camera).
```

```
```

It is now time to plumb the person character’s change state mechanism. We have already seen that there is an *initialize/1* rule’s invocation in the *person_class*’s help-module file which is called by the *set_state/1* assignment rule and that, after a potential state change, it initializes the new state:

```
set_state(State):-this(This),reassert(This:state(State)),
            This:file(File),File:initialize(State).
```
In fact, there are several different initialize/1 rules in the help-module, each one of them initializing a different state. In robert_file’s case, those rules are:

initialize(beginning).

initialize(refused_to_give_book):-peter:change_state(doesnt_get_book_from_robert).

initialize(can_lend_book).


initialize(waiting_book_back):-robert:set_usability(can_use).

initialize(got_book_back):-time_file:show('Peter: I am finally done!'),
                    time_file:show('Peter: Here is your book! Thanks a lot!'),
                    time_file:show('Robert: You are welcome...'),
                    time_file:show('Peter: I hope I wasn’t very late...'),
                    time_file:show('Robert: No, not really! You arrived in time!'),
                    time_file:show('Robert: Now excuse me, but I need to get back to my studying...'),
                    time_file:show('Peter: Sure. I am going then...'),
                    time_file:show('Robert: Bye!'),
                    time_file:show('<SLAM>'),
                    peter:set_place(corridor),
                    robert:set_usability(cant_use),
                    peter:change_state(returns_book_to_robert).

Taking a careful look on them, one definitely notices three important things that need to be taken into consideration.

The first thing is that the initialize(got_book_back) rule demonstrates a dialogue that doesn’t exist among robert’s dialogue sets. Apparently, that dialogue couldn’t stand in a separate set, since it appears only once and independently from the other sets, being that way impossible to be embraced by starting sets ‘0’ or ‘3’ and finishing set ‘7’; thus, it is incorporated in an initialize/1 rule.

The second thing that needs to be taken care of is that some rules are in fact predicates and they do absolutely nothing. The reason, however, that they are used is that we want to prevent person_class’s set_state/1 rule, which invokes the initialize/1 rule from the help-module, from failing.

Finally, the third noteworthy thing is that some of the initialize/1 rules invoke a change of state for our ego character. Bearing in mind that ego_class’s help-module’s initalize/1 rules externally activate transition in a person’s States Diagram, one can roughly infer that there is a kind of interaction between ego’s States Diagram and the person’s one, in the sense that they cause each other to potentially change state. Considering robert’s case, the interaction between its States Diagram and peter’s one is depicted in the following illustrative diagram:
Having demonstrated and explained the person characters’ entities, Prolog Adventure Implementation’s mechanism is now completely exposed. Yet, there is something else missing; an “umbrella” file standing above the rest of the game files, both class-modules and help-modules, and causing the game to run, or, in other words, the “exe-like” file which starts the game.

7.9. The “Umbrella” Game File

As discussed in a previous chapter, all modules used in our game have been loaded into Prolog’s database by peter_file’s load_modules/0 rule, except for that of ego_class. Furthermore, ego_class module, apart from being loaded, needs to be instantiated to our ego character peter, and it is peter instance which, through its associate help-module, peter_file, loads the rest of the modules into the game. Thus, an “umbrella” file is necessary, a rule of which would load the ego_class module and instantiate it to peter. Before that, however, the rule must load the necessary oop.pl Object-Oriented Programming extension file for SWI-Prolog, without which no _class file is going to work.

Such a rule would therefore be:

```
begin:-consult(oop),consult(ego_class),
      Ego=peter,instance_of(Ego,ego_class).
```

In all, in order to load the game into SWI-Prolog, one should first need to consult the game.pl file:

```
?- consult(game).
% game compiled 0.00 sec, 10,092 bytes
true.
```

while in order to start playing, the afore-mentioned begin/0 rule has to be called in the listener:
Apart from the begin rule, however, other rules, serving the purpose of making our text-based game user-friendlier, should be defined in the same file. In particular, those rules to be defined shall invoke ego_class’s basic rules that the user is supposed to call within the game:

```
look_at(Object):-current_instance(Ego,ego_class),
            Ego:look(Object).
talk_to(Object):-current_instance(Ego,ego_class),
            Ego:talk(Object).
use(Object):-current_instance(Ego,ego_class),
            Ego:use(Object).
use(Object1,Object2):-current_instance(Ego,ego_class),
            Ego:use_with(Object1,Object2).
look_around: current_instance(Ego,ego_class),
            Ego:show_objects_of_current_place.
look_at_inventory: current_instance(Ego,ego_class),
            Ego:show_objects_of_inventory.
where_i_am: current_instance(Ego,ego_class),
            Ego:check_current_place.
```

Finally, an additional rule for demonstrating the other rules’ use and purpose also needs to be defined:

```
help:-write('Allowed actions:'),nl,
        write('look_at(X) % Look at entity X'),nl,
        write('talk_to(X) % Talk to entity X'),nl,
        write('use(X) % Use entity X'),nl,
        write('use(X,Y) % Use entity X with entity Y'),nl,
        write('where_i_am % Show the place which I am at'),nl,
        write('look_around % Show the entities lying at the current place'),nl,
        write('look_at_inventory % Show the objects lying in the inventory'),nl.
```

Despite the fact that our game has definitely become user-friendlier with those auxiliary rules, it is still a text-based game, that is to say boring and not really able to compete against the contemporary adventure games which incorporate 3D graphics, gallant music and realistic dialogues. Consequently, a way to upgrade the proposed gaming technique into graphics-based adventure games ought to be sought.

### 7.10. From Text to Graphics

Prolog doesn’t have a good reputation for dealing with graphics and audio effects. On the contrary, the most complicated the produced graphics and effects are, the slower the
execution of Prolog is. For this reason, the construction exclusively upon Prolog of a 3D game engine capable of producing state-of-the-art interactive graphics is strongly discouraged.

Yet, there is a potential solution to the graphical integration of our Prolog Adventure Implementation. The contemporary adventure games engines like Wintermute, SLUDGE, SCUMM etc, handle the graphics in an efficient way; not only do they produce realistic 3D graphics but they also hide all the OpenGL low-level details from the programmer, rendering the construction of the desired interactive graphics easy and highly effective. The same easiness is also valid regarding the handling of the audio effects. Nevertheless, those engines use an ordinary scripting language like Python for the coding of the game classes and events, which definitely doesn’t have Prolog’s powerful adventure-game-construction capabilities described in the previous chapters. The solution therefore seems to be the construction of a new adventure game from scratch which shall combine the scripting language for the programming of graphics and sounds and Prolog for the coding of the game’s “logic”.

As been decided, in Prolog Adventure Implementation the only actions that our ego character is allowed to take, apart from the redundant auxiliary ones, are to look at, talk to and use an entity as well as use two entities together. Considering therefore the proposed game engine, an appropriate “division of labour” between the scripting language and Prolog should assign the former with the movement of the ego character and the handling of the variant events whereas the latter should only handle the four permitted actions described above. In particular, whenever such a mouse event on an entity is triggered, for instance the event dictating the use of the cup, the respective rule will be called in Prolog’s listener, in our case: `?-use(cup),` while the `cup_class.pl` prolog file, on its respect, needs to have an attribute storing the name of the .bmp file of the cup’s image.

Moreover, after the performance of a permitted action on an entity, and subsequently the call of the respective Prolog rule in the listener, some events may also need to be triggered. For example, the use of a door resulting to the ego character’s getting out of a specific room and entering a new one, needs to trigger the script events of the depiction of the ego walking through the door and appearing in the new room. Hence, in the `door_class` module, an additional rule calling the respective event in the scripting language needs to be included.

In case such an adventure games engine is to be designed, the Prolog Adventure Implementation traits having been described throughout this dissertation can contribute to the construction of powerful graphics-based adventure games, combining a wide range of advantages.

7.11. Advantages of the Technique

The most obvious advantage of Prolog Implementation Technique is the perfect clarity infused by it into the game’s code. The states of the ego character or other entities are discrete, easy to encode and indentify. The variant rules are also easy to understand and
deal with, while concepts such as the content and the function of an entity add even more explicitness to whole outcome.

Moreover, high clarity comes together with low complexity. While under a conventional scripting language the programming of the transitions between the numerous states of an entity or the interchangeable contents of an object would demand innumerable nested if-then-else commands, following the Prolog Adventure Implementation technique the definition of some fully-comprehensible predicates and a simple transition rule are enough. The programmer can that way program any scenario they want for their game no matter how complicated this scenario is; all they need is to get the several different states diagrams to “work” effectively with each other.

Along with the increase of clarity and the decrease of complexity comes also the elimination of error-proneness. Although even the state-of-the-art commercial adventure games may get one state initiated before the preceding one has been completed, under Prolog Adventure Implementation such logical flaws are impossible to take place; the different states have been defined in a strict order and are clearly separated from each other, so the admissible sequence is by no means possible to be violated.

Nevertheless, the most important and challenging advantage stemming from this technique is the realism that its games are characterized by. The notions of the content, the function etc. of an object, transfer the focus from the object itself to those specific attribute of its. Any object can be therefore used for a particular purpose, as long as it has the specific attribute that is required for it; for instance, if the player wants to fill their glass with water, they can use any object containing water, that is to say a tap, a bottle, a bucket in the bottom of a well etc. This automatically extends the object world and increases exponentially the non-determinism of the game.

But in order for this realism and non-determinism to be fully achieved, even further development of our technique is necessary to take place.

### 7.12. Future Work

The first enhancement that needs to be made so that even greater realism is added to a game of this kind is the relaxing of the 4-actions convention. That is to say, instead of allowing only the “look”, “talk”, “use” and “use with” rules to be performed on a game entity, a broad variety of actions should be introduced, such as “pick up”, “turn on”, “open”, “push”, etc. In fact, those kinds of actions were present in older adventure games, but it has apparently become common sense that the more the allowed actions are in an adventure game, the less attractive that game is. Yet, increasing the number of the permitted actions in the Prolog Adventure Implementation technique along with extending the object world, could lead to a totally different kind of adventure games, imitating efficiently the real world and requiring even greater skill from the players. For this new kind of games to be effective though, additional changes concerning the degree of the player’s freedom need to be made, such as allowing the player to pick up and use any of the collectable objects around him regardless of the state the player is currently at, setting restrictions on the amount of items the player can carry in their inventory, etc.
Another enhancement that can be made in our technique is the addition of the concept of arity to several objects serving the purpose of storage of some kind of content. The jar of sugar, for example, instead of containing infinitive amount of sugar inside, should have an arity attribute representing the amount of sugar remaining, every time the ego character uses it. As sugar is uncountable, the arity attribute for the jar of sugar can be the weight of the remaining sugar in it, while in the case of a packet of aspirins, where its content is countable and discrete, the arity can represent the amount of pieces still inside.

Apart from the arity attribute, an effectiveness one, that is to say a number from 0 to 100 representing how effective is an object for a particular purpose, can be incorporated as well. Suppose, for example, that, like in our game, we want to stir a watered mixture of coffee and sugar, and the available objects having as their function's argument the stir value are the paddle and the spoon. A paddle is naturally more effective at stirring liquids than a spoon, so we may assign the former's effectiveness attribute to be 100, while the corresponding one of the latter may be 60. A similar effectiveness attribute can be related to a box of coffee according to the type of coffee contained in it; red-cup coffee may have effectiveness 100 while the one of the ordinary type may have 80. The variant effectivenesses of the objects finally selected to be used in the coffee-making procedure can be well summed up with the appropriate weights, resulting to an overall coffee effectiveness which, if over a specific limit, the drinking-coffee process may be considered to be effective and lead to the next stage, otherwise a new cup of coffee may need to be made.

Finally, considering the long run, there is another development that can be made, this being of highest importance. A user-friendly interface may be built upon the potential Prolog adventure games engine been suggested in the previous section, which shall raise the level of abstraction even higher; that is to say, a novice programmer shall be able to chose between a set of predefined objects for their game and encode any scenario they wish by just drawing the variant states diagrams and handling the attributes of those objects through a graphical interface. Although such a goal would seem distant and unlikely to be achieved under a conventional script language, Prolog's powerful declarative nature promises that nothing like this is impossible.
ACKNOWLEDGEMENT

I would like to thank Björn Lisper for supervising me on my thesis dissertation. Not only did he give me unlimited freedom of choice and showed confidence in my work but he also provided me with valuable guidance and crucial pieces of advice. His contribution has, indeed, been significant.

I would like as well to thank Lars Asplund for trusting me from the very first moment that I suggested this ‘Prolog Adventure Game’ subject for my thesis and for orientating me properly. His help has been without doubt more than important.
REFERENCES


APPENDIX A

The Game Scenario

The scenario revolves around a university student called Peter (our ego character), who has to take an important exam in less than 24 hours, but he misses a book that is necessary for him to study.

It’s Friday morning, and he wakes up in his room at the student residence, having a terrible headache. He suddenly realizes that he was partying yesterday all night long at the students union’s disco, having drunk innumerable quantities of alcohol. What he needs in order to get over is an aspirin and a cup of coffee. He therefore goes to the common kitchen room of the student residence and makes hot coffee to drink to feel better. For this purpose he uses the following items: a cup which lies in the kitchen and an electric paddle, which he finds in his drawer. The paddle however has an old battery inside, so he needs to replace it by a new one. In order to make the coffee, he first needs to clean the cup since it is dirty, so he uses the washing fluid and the sponge lying next to the wash basin. Along with the coffee he uses one of the two glasses lying in the kitchen in order to make an aspirin dilution to drink. After drinking his coffee and the aspirin dilution – two tasks that he can carry out in any order – he starts feeling better and gets a flash of memory that his student-mate, Robert, who lives at the same residence, has borrowed the last copy of an book on Economics from the library in order to study for the exam. Therefore, thinks Peter, it would be nice if he could borrow the book to make some photocopies of the most important chapters and study them later. He immediately goes to Robert’s room, knocks the door and asks the latter to lend him his book to make photocopies. Robert however, kindly refuses, as he would miss his book for too much time until Peter goes to and comes back from the closest bookstore to make his copies. Peter tries to persuade him but Robert doesn’t make up his mind.

Peter then has a brilliant brainwave: to take some pictures of the pages of the book with a digital camera, and go to the university lab to print them. He remember that the only person around who has a digital camera is Helen, a beautiful girl whose room lies next to his and with whom he is in deep love but she is always rejecting him. He then knocks the door of Helen to ask her to borrow him her own camera. Helen opens the door but she soon slams it on his face. Peter then knocks Catherine’s room door; she lives also in the same residence. He first asks her why Helen reacted like this. Catherine explains that yesterday night at the party Helen saw him kissing with another girl and since then she got very upset. Peter feels very sad, but he thinks that there is nothing he can do now, and that his goal is to get her camera. He then asks Catherine to help him. Catherine comes up with a plan to invite Helen to her room and watch a movie together, and somehow make her fall asleep and remove the camera from her while she is sleeping and give to Peter. Therefore, Peter needs to prepare a drink for Helen putting something in it that can make her fall asleep. The tranquillizers that he keeps in a special medicine box in his drawer are suitable for this purpose. He directly goes to the kitchen, takes the two...
glasses, cleans them, prepares two drinks in them and dissolves a pill in the one which Helen is going to use.

Everything goes according to the plan; Helen falls asleep immediately and Catherine gives Peter her digital camera. He then borrows Roberts's book for a few minutes in order to take pictures of it. So does him.

As soon as he finishes taking pictures of the book, he returns it to Robert. He then goes to the university, enters the lab, plugs in the camera to the computer and prints the pictures. Later on, he goes back to the residence, meets Catherine and returns Helen's camera to her. He is eventually done with everything so he goes to his room to study the print-outs of the book.

At the end, Peter is faced with either one of two different finales; which finale will be shown depends on whether Peter remembered to erase the photos from the camera or not.
APPENDIX B

A Realistic Kitchen

We saw at section 7.2 that the content_order/3 predicates for the making coffee procedure are:

- `content_order(nothing, raw_sugar, raw_sugar). % Addition`
- `content_order(nothing, raw_coffee, raw_coffee). % Addition`
- `content_order(raw_sugar, raw_coffee, raw_mixture). % Addition`
- `content_order(raw_coffee, raw_sugar, raw_mixture). % Addition`
- `content_order(raw_mixture, water, watered_mixture). % Addition`
- `content_order(watered_mixture, stir, stirred_mixture). % Action`
- `content_order(stirred_mixture, hot_water, coffee_without_milk). % Addition`
- `content_order(coffee_without_milk, milk, coffee). % Addition`

However, in the beginning of the game, the cup’s content was set to be remnants. What was therefore deliberately omitted when illustrating the coffee’s States Diagram is time to show up:

- `content_order(remnants, clean, foam). % Action`
- `content_order(foam, water, nothing). % Addition`

Those two lines actually refer to the transitions from the remnants content to the foam one and from the foam one to nothing. It should be mentioned that the clean function is achieved by the sponge, when detergent and then water has been added to it, according to its own States Diagram.

What now if we want to discharge the current content of the cup? Such an option would add even more realism and flexibility to our game.

Hence, for this purpose, a basin_class module is defined and a basin instance is created upon it. With this basin instance we are allowed to use the cup – as well the two glasses, according to the priority/2 predicates defined in peter_file – emptying this way the latter. In order to achieve that, we only need to call basin’s use_with/1 rule with cup as its argument, or equivalently, one of the peter:use(basin,cup) or peter:use(cup,basin) ones. basin_class’s use_with/1 rule is therefore demonstrated below:

```
use_with(Object): Object:content(Initial_Content),
    this(This),This:file(File),
    File:emptying(Initial_Content,Final_Content),
    Object:set_content(Final_Content).
```
At first, it should be mentioned that File, i.e. basin_class’s help-module, has been set by the constructor/0 rule to be basin_file, common for all basin instances:

```
constructor:-File=basin_file,
    this(This),This:set_file(File),
    This:set_usability(can_use).
```

The use_with/1 rule therefore, reads the Initial_Content content argument of Object, and if there is a Final_Content which is related with this Initial_Content through an emptying/2 predicate in basin_file, Object’s new content will be assigned with the Final_Content value.

This emptying procedure becomes clear if we take a look at the basin_file’s emptying/2 predicates, and specifically those which concern the variant cup contents:

```
emptying(raw_sugar,nothing).
emptying(raw_coffee,nothing).
emptying(raw_mixture,nothing).

emptying(watered_mixture,remnants).
emptying(stirred_mixture,remnants).
emptying(coffee_without_milk,remnants).
emptying(coffee,remnants).
```

One can see that if the cup contains a gummy content, such as watered_mixture, stirred_mixture etc, when being emptied into the basin, what is left in the cup is remnants. As we have also seen before, in order for those remnants to be cleaned, the sponge needs to be used. On the contrary, when the content is raw_sugar, raw_coffee, or raw_mixture, that is to say non-gummy substances, when the cup is being emptied, absolutely nothing remains inside it.
APPENDIX C

User-friendly Output Functions

In our game’s code, we have encountered several times the `time:show/1` output rule. This rule apparently refers to the `show/1` rule in the `time` module, which takes a text as its argument and results to the cursor’s remaining idle for 2 seconds after the text has been printed on the screen. The `show/1` rule is demonstrated below:

```
show(Phrase):-write(Phrase),nl,delay.
```

What one may have suspected is indeed true; a `delay/0` rule is the one causing the delay on the screen. The `delay/0` is therefore shown:

```
delay:-get_current_time(Original_Time),
       repeat(Original_Time).
```

As one sees, `delay/0` rule is analyzed into two other rule, the `get_current_time/1` and `repeat/1` ones, which are as follows:

```
get_current_time(Current_Time):-get_time(X),
       stamp_date_time(X,date(_,_,_,Hours,Minutes,Seconds,_,_,_),-1),
       Current_Time=3600*Hours+60*Minutes+Seconds.
```

```
repeat(Original_Time):-get_current_time(Current_Time),
       Current_Time<Original_Time+2
       ; repeat(Original_Time)
       ; true.
```

What the `get_current_time/1` rule does is to get the current time X with the predefined `get_time/1` rule of Prolog, analyze it with the predefined as well `stamp_date_time/1` rule into hours, minutes and seconds of the current Swedish time (the last “-1” argument in `stamp_date_time/1`’s brackets represents the Greenwich +1 hour time), and finally convert all of them into seconds and add them upp, saving the sum in the `Current_Time` variable.

`repeat/1` rule then takes this sum, which actually represent the original current Swedish time into seconds, reads the new current time in the same way and causes an endless loop by calling repeatedly itself, until the new current time to be read is by two seconds greater than the original one. The reason for taking into consideration the hours, the minutes and the seconds of the current time and converted all of them into seconds, instead, for example, of taking only the seconds, is to eliminate the possibility of having the time reset to 0 while being in a loop, for in this case it will take the look ages to terminate, and the user will inevitably to the Control+C termination option. On the other hand, taking the full hours, minutes and seconds time into account, a reset takes place.
only once a day, at 12.00 a.m., and it is even more unlikely for a time:show/1 rule to be invoked between 2 seconds before 12 am and precisely at 12 am.