FREEFALL REPORT

A Database for the electronic informative system
of a fictitious mobile phone network operator

Nicola Lamonaca
March 2013

TABLE OF CONTENTS

"Per chi viaggia in direzione ostinata e contraria
col suo marchio speciale di speciale disperazione
e tra il vomito dei respinti muove gli ultimi passi
per consegnare alla morte una goccia di splendore,
di umanità,
di verità."

[Fabrizio De André]

I. START YOUR ENGINES

I.I STRUCTURE OF THIS DOCUMENT

Section I introduces some basic information about **Freefall** - including how to use it - and about the present document.

Section II will cover the Analysis, during which we will translate our problem specification document from Italian to English and then analyse it in order to capture the main concepts involved. Later we will refine this analysis in order to make the specification uniform and better understand our domain.

Section III will introduce the Conceptual Design, where we will model our domain further, including those concepts not directly discovered in the previous stage though necessary to the working of the system and discuss how we obtained the final Object-Relational Mapping Diagram (ORMD) shown.

Section IV will discuss the Logical Design, during which we translate our ORMD into structures directly manipulable from the DBMS: tables, views, triggers, etc. Then we will explain each single structure in detail. We do not include the code of each structure in the present document to improve readability; instead, the title introducing each of them is a link pointing to its code snippet.

Section V shows the entire output of the Freefall script.

Section VI finally reports a list of external resources.

I.II WHAT IS FREEFALL

Freefall is the name of the database of the electronic informative system of a fictitious mobile phone network operator and it may be seen as its backend.

To better understand the context in which Freefall would operate, think of this scenario: in Operatorsland there's a building inside of which there are 1 thousand computer screens by which 1 thousand human operators perform each day billions of operations on the **Freefall** database, like activating new contracts, registering errors on bills, getting statistics on customers and so on. Moreover, in each of the several dozens of offices in the Freefall building there's an enormous screen on which the details of billions of calls and messages flow uninterruptedly.

Freefall takes its name upon Felix Baumgartner's leap into the void from 128k+ ft. done on Oct. 14, 2012 (see *http://goo.gl/X02m5*) and it's the project for the exam "*Designing Data Bases with Advanced Data Models*" taught by professor **Donato Malerba**.

Starting from the *project's web page* you can find more in-depth analysis about design decisions, discussions and other considerations and you can freely download the complete project, as long as it is not redistributed without the author's permission.

NOTICE: an internet connection is required in order to consult the present document, as many references are linked on external locations for clarity's sake.

I.III MINIMUM REQUIREMENTS

In order to run **Freefall**, the following list of items must be installed on your system (required items are **bold-typed**):

- **PostgreSQL 9.2** (or newer);
- **PostGIS 2.0.1-1** (or newer);
- **pgAdmin III** (or newer);
- QGIS 1.8.0-Lisboa (optional)

Freefall may work on less recent versions of these items, although it has been developed on the listed versions and its working is guaranteed on them only.

I.IV INSTALLATION GUIDE

Once you've installed the required items, you can proceed with the installation:

1. Download the **Freefall** backup file from *here* (~106 MB, internet connection required);
2. Open pgAdmin III;
3. If you never did this before, *add a new server*, otherwise ignore this step;
4. Create a new database naming it "freefall", leaving all the default options untouched;
5. Right-click on the newly created database and choose "Restore...";
6. Choose the **Freefall** backup file you downloaded and start restoring;
7. Wait until the procedure ends, then you're done.

II. ANALYSIS

Once we have breefly introduced Freefall, we can start telling how the whole story started.

II.I TRANSLATION OF THE SPECIFICATION DOCUMENT

The first step is that of translating the problem specification document from Italian to English (extra info *here*):

> *A brand-new mobile phone operator is making its entrance on the market. Part of the company's electronic informative system relates to the management of contracts with customers. For each stipulated contract, the company registers customer data (which include, among other things, a set of alternative telephone numbers) and assigns a unique telephone number to him or her. Any given customer can stipulate more contracts, and therefore have multiple telephone numbers.*
>
> *Each contract specifies the price plan chosen by the customer. Each price plan includes the price for voice calls, for video calls, for SMSs & MMSs and for international calls. Reduced price plans can be applied during calls made by certain telephone numbers (You&Me). Moreover, the customer contract provides the payment by a recharge voucher ("Pay&Go" tariff) or a monthly bill ("Pay monthly" tariff). If the customer tops up, then the balance must be known. If customers choose "Pay monthly" tariffs, then bank details must be known instead.*
>
> *For each operation (call, SMS, etc.) made by a customer, the operator registers the dialed number, the date and time of the operation and its cost. For [video]calls, the operator registers the duration in seconds, too; for [video]messages, it registers the quantity of data transmitted (in bytes). At the end of each month, the operator ships bills to monthly-paying users. The first page of the bill contains the date, the customer details, and the total amount*

for each category of operation (call, message, etc.); following pages contain the details of each operation; each row reports the operation type, the date and time, the called number and the duration or the quantity of transmitted data, depending on the operation type. The operator provides these kind of bills to rechargeable users who prompt for them, too. It may happen that a customer finds errors in the bill. In this case, he notifies the company of the error, which registers the issue details, the reason and the associated bill; after evaluating the situation, the company may register the amount of the refund, too.

In order to advertise new products/technologies, the company may define temporary offers to make those products/technologies available to customers on a reduced price basis. Each promotion can be associated to certain contracts.

For the current law regulations to be respected, the operator must associate each operation with the location in which the operation has been executed. (i.e. the city where the call comes from).

II.II REQUIREMENTS ANALYSIS

2.2.1 Requirements Grouping

We then group together requirements which refer to the same concept. During this step, we still leave terms that we suspect to be synonyms as separated. What we obtain is shown below (extra info *here*; in red are sentences not pertaining to any concept in particular):

A brand-new mobile phone operator is making its entrance on the market. Part of the company's electronic informative system relates to the management of contracts with customers.

_____CONTRACT_____

- For each stipulated contract, the company registers customer's data (which include, among other things, a set of alternative telephone numbers) and assigns a unique telephone number to him or her.

- Each contract specifies the price plan chosen by the customer.

- Moreover, the customer contract provides the payment by a recharge voucher ("Pay&Go" tariff) or a monthly bill ("Pay monthly" tariff).

_____CUSTOMER_____

- Any given customer can stipulate more contracts, and therefore own multiple telephone numbers.

- If the customer tops up, then the balance must be known.

- If customers choose "Pay monthly" tariffs, then bank details must be known instead.

_____PRICE PLAN_____

- Each price plan includes the price for voice calls, for video calls, for SMSs & MMSs and for international calls.

- Reduced price plans can be applied during calls made by certain telephone numbers (You&Me).

_____OPERATION_____

- For each operation (call, SMS, etc.) made by a customer, the operator registers the dialed number, the date and time of the operation and its cost.

- For [video]calls, the operator registers the duration in seconds, too;

- For [video]messages, it registers the quantity of data transmitted (in bytes).

- For the current law regulations to be respected, the operator must associate each operation with the location in which the operation has been executed. (i.e. the city where the call comes from).

_____BILL_____

- At the end of each month, the operator ships bills to monthly-paying users.

- The first page of the bill contains the date, the customer details, and the total amount for each category of operation (call, message, etc.); following pages contain the details of each operation;

- Each row reports the operation type, the date and time, the called number and the duration or the quantity of transmitted data, depending on the operation type.

- The operator provides these kind of bills to rechargeable users who prompt for them, too.

- It may happen that a customer finds errors in the bill. In this case, he notifies the company of the error, which registers the issue details, the reason and the associated bill;

- After evaluating the situation, the company may register the amount of the refund, too.

_____PROMOTION_____

- Each promotion can be associated to certain contracts.

_____SPECIAL OFFER_____

- In order to advertise new products/technologies, the company may define special offers to make those products/technologies available to customers on a reduced price basis.

2.2.2. Requirements Refinement

In this step we unify terms which have synonyms, we explicitly define who or what the pronouns refer to, in order to remove ambiguities (see `Price Plan` and `Tariff` in the step above, which were treated as equal in the current step: it's been chosen to substitute `Price Plan` in sentences formerly referring to `Tariff` because `Price Plan` was considered more expressive than `Tariff`), we enumerate requirements for traceability, we expand and contract sentences in order to gain more clarity, and so on.

A brand-new mobile phone operator is making its entrance in the market. Part of the company's electronic informative system relates to the management of contracts with customers.

_____CONTRACT_____

1. For each stipulated contract, the company registers customer's data (which also include a set of alternative telephone numbers owned by the customer) and assigns a new and unique telephone number to the customer.

2. Each contract specifies the price plan chosen by the customer.

3. The customer contract provides the payment by a recharge voucher ("Pay&Go" [*] tariff) or a monthly bill ("Pay monthly" [*] tariff).

NOTE: [*] as of Vodafone UK naming.

_____CUSTOMER_____

4. The customer can stipulate more contracts, and therefore may own multiple telephone numbers.

5. If the customer chooses a "Pay&Go" tariff, then the operator must know the current customer's balance.

6. If the customer chooses a "Pay monthly" tariff, then the operator must know the customer's bank details.

NOTE: [FUNCTIONAL CONSTRAINT: (5. OR 6.)]

_____PRICE PLAN_____

7. The price plan includes the price for voice calls, the price for video calls, the price for SMSes & MMSes and the price for international calls.

8. The operator can apply reduced price plans during calls made by and towards certain telephone numbers (You&Me).

_____OPERATION_____

9. For each operation being a [video]call, the operator registers the dialed number, the date and time of the operation, the cost and the duration (in seconds) of the operation.

10. For each operation being a [video]message, the operator registers the receiver's number, the date and time of the operation, the cost and the quantity of transmitted data (in bytes) of the operation.

11. The operator must associate each operation with the geographical location in which the operation has been executed. (i.e. the city where the call comes from).

_____BILL_____

12. At the end of each month, the operator ships bills to monthly-paying users. The first page of the bill contains the date, the customer details, and the total expense amount for each category of operation (call, message, etc.); following pages of the bill contain the details of each operation: each row reports the operation type, the date and time when the operation started, the receiver's phone number and the duration of the operation for (video)calls or the quantity of transmitted data for (video)messages.

13. The operator provides bills to rechargeable users who prompt for them, too.

14. If the customer finds errors in the bill, he notifies the company of the error. The company then registers the issue details, the reason and the associated bill; after evaluating the situation, the company may register the amount of the refund, too.

_____PROMOTION_____

15. Each promotion can be associated to certain contracts.

_____SPECIAL OFFER_____

16. In order to advertise new products/technologies, the company may define special offers to make those products/technologies available to customers on a reduced price basis.

2.2.3. Classes Identification

Once we have come to a clear, complete and non ambiguous list of requirements, during last step in Analysys we identify classes, which will be only *some* of our entities in the data model. Each of these classes will hold part of the domain knowledge. It is important to point out that this is only a first, high level classification and that the final, actual taxonomy will take into consideration Object-Relational Database modeling techniques, in order to distribute that knowledge in a more convenient way.

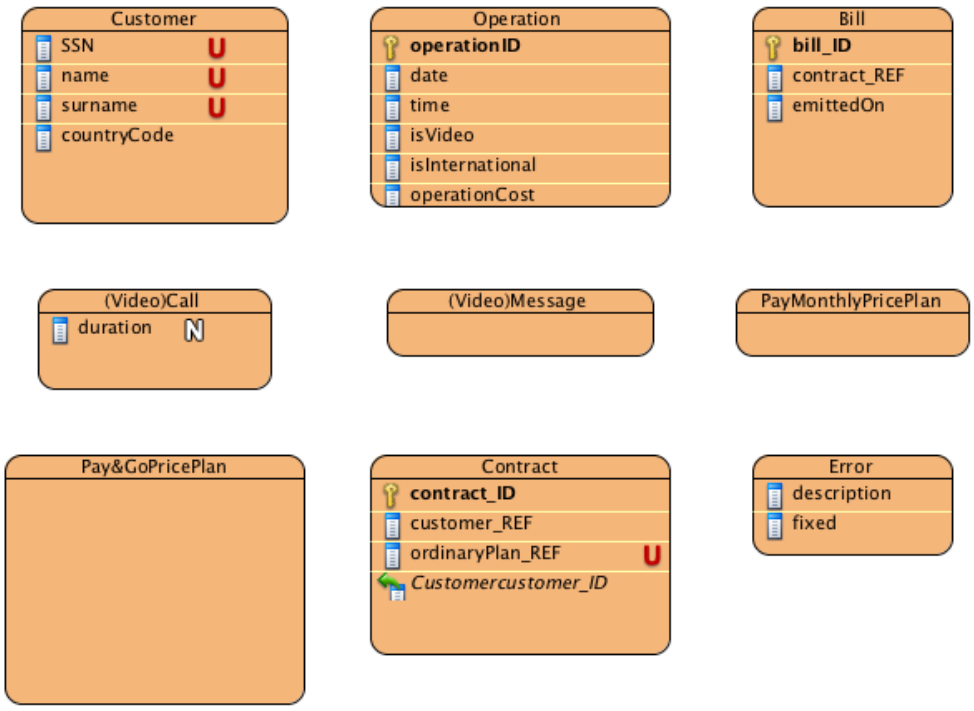
The final list of accepted classes is the following:

No.	Name	Description
1.	**Contract**	Represents a contract associated to a given customer
2.	**Customer**	Represents a customer
3.	**PricePlan**	Represents a price plan associated to a given customer
4.	**Operation**	Represents an operation
5.	**(Video)Call**	Represents a kind of operation
6.	**(Video)Message**	Represents a kind of operation
7.	**Bill**	Represents a bill associated to a contract
8.	**Error**	Represents an error in the bill
9.	**Promotion**	Represents a promotion associated to a contract
10.	**Special Offer**	Represents an offer of a new product

Upon these, we can develop the Conceptual Data Model.

III. CONCEPTUAL DESIGN

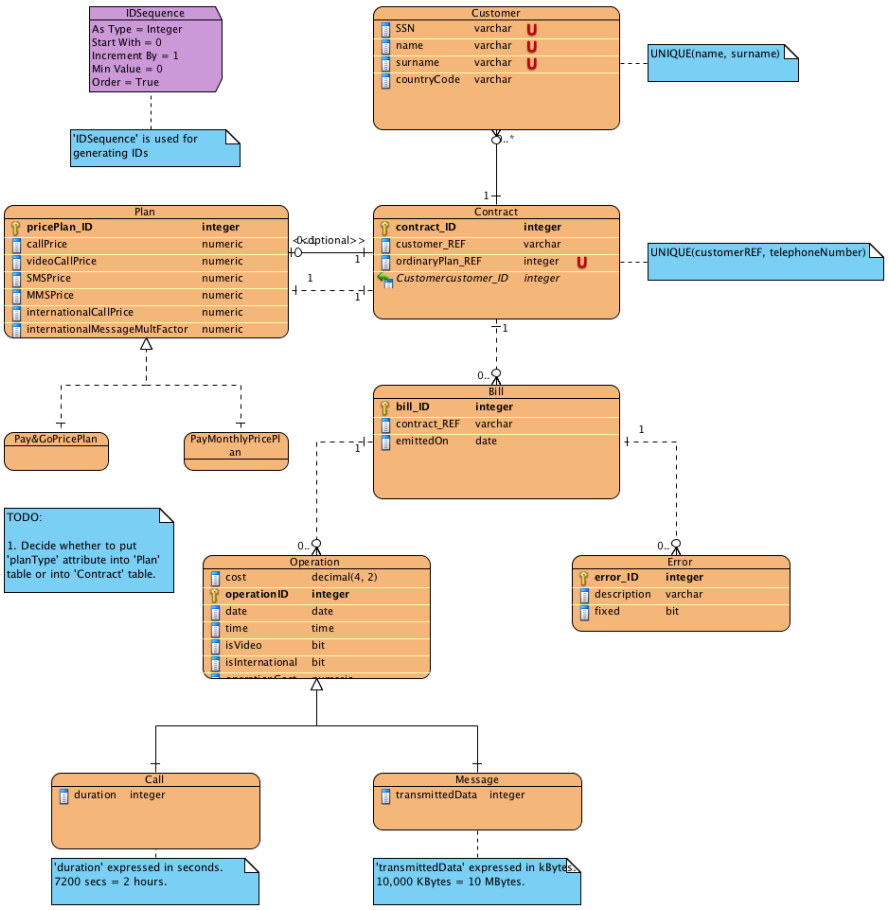
The initial step is that of creating as many entities as the accepted classes in the previous phase are. The first, rough, diagram is shown below.



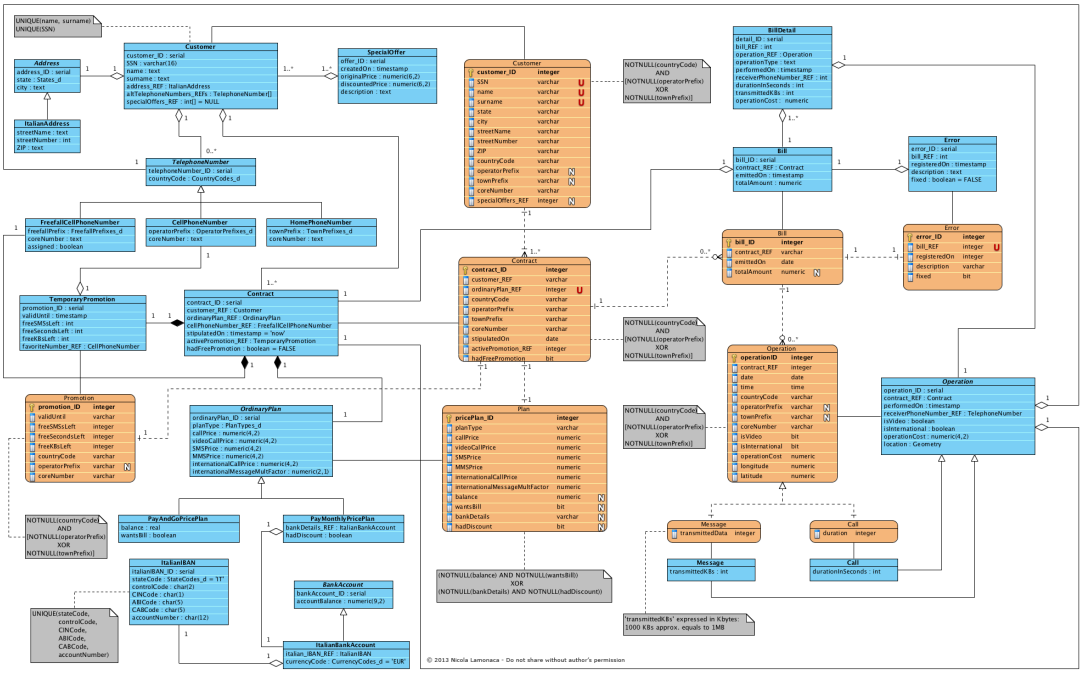
At the moment it is not important to connect entities: we only want to start wondering what the overall infrastructure will be. We begin by first adding the essential attributes we think are necessary for the whole apparatus to work. At this stage, it is highly likely that we will remove, add and modify attributes many, many times, until a minimal stable configuration is reached.

Next step is that of enriching the very first diagram, adding even more attributes and connecting entities by the proper relations with proper multiplicities. Then we start thinking about subclassing some entities and factorize some attributes where appropriate.

Comments and stereotypes are good friends to fix ideas, since we still are not sure about some decisions that must be taken: for example, we can write down doubts on comments and leave their solution for later moments, when the point will become clearer or we can express complex contraints which are not directly expressable with the diagramming tool.



If everything goes fine, we should come to a very stable diagram, which does not require modifications at all, or does require very minimal and easy manageable modifications. The final version of the **Freefall** ORMD is shown below.



A high-resolution version of the diagram can be found *here* (internet connection required).

If you're curious about how many modifications are required in order to obtain the final ORMD shown below, look at the file name shown in the link above: it says "25" at some point. It means that the diagram has been updated and uploaded *at least* (first versions were not uploaded, of course) 25 times **after the author considered it to be a stable diagram, which didn't require no more modifications!** :)

We can now discuss some of the decisions made during the development of the model.

III.I WHY THE ORMD?

It was chosen not to develop neither an Entity-Relationship Diagram nor a Class Diagram to design the system, but an Object-Relational Mapping Diagram (ORMD) instead. As we've seen, the final ORMD shown above is the refinement of a first, very rough ER Diagram consisting of only the Entities discovered in the Requirements Analysis. ERs are very useful tools for quickly and easily focus on initial ideas, when they're still few and blurry. But as you develop the system more and more, they soon become not much suitable, as they cannot represent composite values, custom types or relationships among entities, such as composition and aggregation. Not to mention inheritance, which is not supported by pure relational modelling at all. These limits are all overcome with Class Diagrams; so, instead of simply creating two separate diagrams (an ER Diagram and a Class Diagram), an Object-Relational Mapping Diagram was created instead, which was used to map Objects (expressed in UML Classes) to Relations (expressed in ER Entities), as the name implies.

The orange boxes are entities, as they've been identified in the Requirements Analysis phase. The blue boxes are classes. In total, we have 9 entities and 22 classes. The reason why we have more classes than entities (as I explained above) is an effect of the refinement: in fact, first as much classes as the entities were were created, and then started wondering how to extend the design working on classes. These classes as a whole constitute the core of **Freefall**, but the actual number of classes actually used in the system is higher due to implementation matters. Let's discuss some of these extensions in detail.

3.1.1 Modelling Addresses

The first kind of refinement is regarding how to model addresses. Right from the beginning, we tried to look forward and decided to give the possibility to potentially use any existing addresses in the world, thus promoting extensibility. So two classes were created: a parent class named `Address`, which contains information related to the address' state and city, and a child class named `ItalianAddress`, which also adds attributes to represent the street name, the street number and the ZIP code. The State will be of type `States_d`, where the trailing '`_d`' stands for *domain*. ZIP codes are not treated as a domain because when the development started (October 2012), italian districts reorganization was in place, so it had no much sense to create a domain as it was not a stable concept. Surely a customer can possess one or more addresses, but we're not interested in knowing them all, so we simply ask for one, the main one. This way, in the **Freefall** little world, each customer has one and only one address and that's the reason why the relationship's multiplicity between the `Customer` and `Address` classes is a 1-to-1 aggregation relationship - where the whole-side is the `Customer` class and the part-side is the `Address` class, since in no way if a `Customer` object is deleted (or even killed!) an `Address` object's lifetime ends. We have to decide which object must refer to the other. In that it is a 1-to-1 relationship, any of the two objects can do this, but the common sense suggests us to let the `Customer` class refer to the `Address` class. Another reason for doing so is that the `Customer` is a central entity in our domain, much more than what the `Address` is meant to be, so it is much more likely to ask for the address given the customer, rather than the contrary. Finally, The `Address` class is an abstract class, since in no way we can instantiate generic addresses if we don't know what their structure is. We use rules to prevent the `Address` class (respectively table) from being created (respectively populated) or manipulated:

```
CREATE RULE no_insert_in_address_table AS
   ON INSERT TO Address DO INSTEAD
      SELECT operation_not_allowed_FNC('INSERT', 'Address');
```

where the *operation_not_allowed_FNC(text, text)* function simply raises an exception each time someone tries to manipulate the table which name is passed as the second argument:

```
ERROR: Operation not allowed: INSERT INTO Address.
```

The two parameters are simply used to personalize the exception message. Other rules are created for UPDATEs and DELETEs, so each of the 5 abstract classes appearing in our ORMD (*Address*, *TelephoneNumber*, *Operation*, *OrdinaryPlan* and *BankAccount*) will have 3 rules, one for each event: INSERT, UPDATE and DELETE.

3.1.2 Modelling Telephone Numbers

The next kind of refinement pertains to telephone numbers. Even in this case a base abstract class was used, named *TelephoneNumber*, which holds information about country codes, treated as domains. *TelephoneNumber* has three subclasses: *CellPhoneNumber*, *HomePhoneNumber* and *FreefallCellPhone-Number*. The first one models mobile phone numbers (holding the operator prefix and the core number), whereas the second one models home phone numbers and so it holds town prefixes and core numbers, too. The third and last one stores Freefall numbers assigned to newcome customers. *CellPhoneNumber* contains those telephone numbers that either have been assigned to customers or are customer's alternative telephone numbers. The reason why they all store the core number is because not necessarily it has the same format for cell phone and home phone numbers. Country codes, operator prefixes and town prefixes are declared of a custom domain type. A customer is supposed to possess no, one or more than one alternative telephone numbers, so the relationship's multiplicity is declared to be a 1-to-many relationship. Moreover, it is an aggregation relationship, because just like in the *Address* case, if a customer gets

accidentally killed in a car crash his telephone numbers would remain; another reason is because it would be completely a nonsense stating that a customer "*is composed by*" telephone numbers. Since a customer can potentially own more than one telephone number, we use a container to hold them in the `Customer` table. Note: the best choice here is to store `TelephoneNumber`s and not `CellPhoneNumber`s, `HomePhoneNumber`s or `FreefallCellPhoneNumber`s, since we have no mean of knowing in advance what kind of phone numbers customers will already own. An identical reasoning applies to the `Operation` table, since we want to register operations towards both type of phone numbers, while in the `Contract` table we include `FreefallCellPhoneNumber`s only, of course.

Another table is derived from `CellPhoneNumber` table: `FreefallCellPhone-Number`. This table holds randomly-generated cellphone numbers which are assigned to a contract as soon as a new one is created.

3.1.3 Modelling Bank Accounts

Last kind of refinement is related to the modelling of bank accounts, which are of interest for licensed customers only (customers owning a `PayMonthly-PricePlan`). Again, an abstract base class, named `BankAccount`, was created, which only stores information about the balance, and a derived class, `ItalianBankAccount` which models an italian bank account using the italian version of the International Bank Account Number (**IBAN**) and storing information about the currency code (EUR for Italy). Just like addresses, the system was made suitable for future developments giving the possibility to use other kinds of bank accounts if required.

III.II. A NOTE ON INHERITANCE - PART I

We discuss here Inheritance from a modelling point of view, leaving the implementation-related discussion to the next section, where we will discuss the logical design of **Freefall**.

Since we are working on PostgreSQL, which is an Object-Relational DataBase Management System (ORDBMS), we can take advantage of its native support of **Inheritance** to build an extensible data model. Think of a scenario where we want to model italian addresses: we could define a relation on italian addresses and this would be fine as long as we operate with italian addresses only. But what if at one point for some reason we have to manage Anglo-Saxon addresses, too? We would of course define a new relation to hold the new structure of Anglo-Saxon addresses, but problem is that at this point we would have redundant attributes over the multiple classes, like the State or the city. Moreover, we would have to change our model in a destructive manner, since Foreign Keys formerly referring to italian addresses must somehow refer to Anglo-Saxon addresses too, and this is not easy to achieve at all.

Instead, we take a different approach: we factorize those we think may be recurrent attributes over multiple instances of different kind of addresses in a meta, non instantiable table which will act as a template to instantiate actual addresses. We will populate derived tables only, while the PostgreSQL's inheritance mechanism will take care of propagating data from lower-level to upper-level tables in the hierarchy. Since attributes of parent tables are visible in all of its children tables, instantiation of derived classes would consist on a mere INSERT into the child table where the attributes of both the parent and the child table are specified.

IV. LOGICAL DESIGN

IV.I A NOTE ON INHERITANCE - PART II

As of version 9.2, PostgreSQL's support for inheritance is not really powerful in that it lacks of some key-features: in fact, while all check constraints and not-null constraints on a parent table are automatically inherited by its children, other types of constraints (primary key, unique and foreign key constraints) are not. How to deal with this limitation? One possibility is that of blindly using third-party scripts which are supposed to guarantee the propagation of these constraints throughout the hierarchy. Another solution is that of first questioning if such an approach is actually the best one given a particular context. Let's examine ours:

1. **Primary Key constraints** are not a real problem. Since we are using implicit sequences on PKs defined on parent tables (i.e. `operation_ID serial PRIMARY KEY` in attributes definitions) and since we are making INSERTs only into derived tables, preventing those on parent abstract tables, we are sure we'll have different tuples in each table. Where appropriate, we'll have a field in derived tables acting as a primary key, while always storing the inherited ID of its parent table (which cannot be used to reference a child table from an external table). This mechanism is completely automatic as it is managed with the SERIAL type, which makes use of auto-incrementing SEQUENCEs behind the scenes.

2. **Unique constraints** are defined only on parent abstract classes, thus we don't need to propagate them throughout the hierarchy.

3. **Foreign Key constraints** are defined on parent abstract classes but in one case: the `Operation` table. In fact, here we have a column which stores the contract reference and for this reason a foreign key constraint is defined. In order to obviate the fact that DML's operations on its children tables will not check that the constraint is respected - thus leading to situations where someone could perform operations for free by simply

injecting somehow a reference to a non-existent contract in an Operation's INSERT - we have 2 triggers defined on both `Call` and `Message`, which check that a series of conditions are verified before an operation is actually performed, among which is the check that the contract actually exists.

Apart from these cases, **check** and **not-null** are the most part of the constraints defined on **Freefall** tables and, as mentioned above, these give no problems for what concerns inheritance, since they are automatically propagated over derived classes by PostgreSQL's inheritance mechanism.

It turns out that external scripts would have introduced in **Freefall** an unnecessary complexity which has been avoided with a little bit of domain analysis.

IV.II SCHEMATA DEFINITION

Schemata have been used to organize all the elements in a rigorous manner, depending on the purpose of each. We have a total of 4 schemata in **Freefall**; one is *public*, which contains those elements resulting from the restoring of the Lodi province backup the author was provided of, then we have *topology*, a default schema created by Postgis to manage spatial features. **Freefall** on its own creates two schemata: *support* and *freefall*, which are breefly described below.

4.2.1 SUPPORT

The schema *support* contains all those elements which are not directly part of the core of **Freefall**, but are however needed in order to implement part of its functionalities.

4.2.2 FREEFALL

The schema *freefall* is the main and more important one as it contains all those tables, functions and other elements which consitute the **Freefall** kernel.

IV.III DOMAINS DEFINITION

We use domains to constraint inputs on well-known data types, which are not native types known by the DBMS. Domains inherit their behavior (ie. the internal representation on the machine) from default data types, but they restrict the set of admitted values by listing only the allowed ones. Unlike ENUMs, they have the ability to check that complex constraints are verified on the acmittable values.

4.3.1 SUPPORT.COUNTRYCODES_D

The domain *support.CountryCodes_d* defines all the possible values for country codes used to model telephone numbers. A country code is a 1- to 4-digits string which must be placed before the desired number when calling towards a foreign country. Normally, for in-country calls it is optional, however within **Freefall** we require it anyway.

4.3.2 SUPPORT.OPERATORPREFIXES_D

The domain *support.OperatorPrefixes_d* lists all the admitted italian operators prefixes. An italian operator prefix is a 3-digits string.

4.3.3 FREEFALL.FREEFALLPREFIXES_D

The domain *support.FreefallPrefixes_d* is a subset of *support.OperatorPrefixes_d*, in that it lists the **Freefall**'s operator prefixes, used while generating a new cellphone number which will be later assigned to a new contract as a new customer arrives.

4.3.4 support.TownPrefixes_d

The domain *support.TownPrefixes_d* establishes all the admitted values for italian town prefixes, which are 3- or 4-digits strings and come before the core number.

4.3.5 support.States_d

The domain *support.States_d* specifies all the allowed values for country names. These are used to model addresses. Since at the time writing we're interested in italian addresses only, we only use the value "Italy".

4.3.6 support.PlanTypes_d

The domain *support.PlanTypes_d* lists only two values for the two kinds of plans a customer can subscribe. "*m*" stands for *PayMonthlyPricePlan* and indicates a plan where a customer is charged by bills emitted on a month basis, while "*g*" stands for *PayAndGoPricePlan* and indicates a plan in which a customer can perform operations as long as he has enough credit left.
If a customer has enough credit at the beginning of the operations but this turns to be not enough while the operation is still in place, the current operation is not aborted; instead, it is completed and the balance updated at the end of the operation. When the credit is not enough, the customer will be allowed to perform new operations only if he tops up again. When this happens, the actual credit will depend on the voucher total amount and how much he went down during his last operation.

4.3.7 support.CurrencyCodes_d

The domain *support.CurrencyCodes_d* defines the values for the currency codes of the world used to model IBAN codes in bank accounts. Since at the time writing we're interested only in italian bank accounts, we use the 'EUR' currency code only.

4.3.8 support.StateCodes_d

The domain support.StateCodes_d specifies the codes of all the countries in the world. Again, this is done to promote extensibility. It is used in the modelling of addresses. At this time we use only the 'IT' value.

IV.IV TYPES DEFINITION

Types are used to force input of row values to a custom format, as in the case of `support.TelephoneNumber_t` and `support.Address_t`, or to build a custom return type from functions, as in the case of `support.Resume_t` and `support.ResumeExtended_t`.

4.4.1 support.Resume_t

The type `support.Resume_t` is used to build a custom type which we use to return customers' basic information from the `get_resume_FNC(int)` function.

These information include:

- the contract's ID;
- the customer's ID;
- the customer's name;
- the customer's surname;
- the customer's SSN;
- the contract's country code, prefix and core number;
- the date when the contract has been stipulated;
- how many operations the customer has performed so far.

4.4.2 SUPPORT.RESUME_EXTENDED_T

The type *support.Resume_Extended_t* is used to build a custom type which we use to return customers' extended information from the function *get_resume(int, text)*.

These information include those listed in *support.Resume_t* and, in addition:

- if the customer already had a free promotion;
- the promotion expiration date;
- how many free SMSs, seconds and KBs he has left;
- the plan balance; (for *PayAndGoPricePlans*)
- if the customer asks for a bill;
- if the customer already had a discount;
- the account balance (for *PayMonthlyPricePlans*);
- the customer's account IBAN.

Unfortunately, PostgreSQL 9.2 does not support types inheritance, so we cannot let *support.ResumeExtended_t* inherit from *support.Resume_t*, nor we can declare a field of type *support.Resume_t* in the type *support.ResumeExtended_t*, instead we are forced to copy all *support.Resume_t*'s fields into *support.ResumeExteded_t* too.

4.4.3 SUPPORT.TELEPHONENUMBER_T

The type *support.TelephoneNumber_t* bilds a structure used to force the input of a telephone number when a new operation is performed.

It includes:

- the country code;
- the prefix;
- the core number.

4.4.4 SUPPORT.ADDRESS_T

The type *support.Address_t* defines a structure used to input customers' addresses upon registration.

It includes:

- the state name;
- the city name;
- the street name;
- the street number;
- the ZIP code;

IV.V TABLES DEFINITION

In this section we describe all the tables in the *freefall* and *support* schemata.

4.5.1 FREEFALL.TELEPHONENUMBER

The table *freefall.TelephoneNumber* is a parent abstract table as it cannot be directly manipulated. It only contains the country code.

4.5.2 FREEFALL.CELLPHONENUMBER

The table *freefall.CellPhoneNumber* is a sub-table of the table *freefall.TelephoneNumber* and contains all telephone numbers of current customers and the alternative numbers they indicated upon registration into the system.

4.5.3 FREEFALL.HOMEPHONENUMBER

The table *freefall.HomePhoneNumber* is a sub-table of the table *freefall.TelephoneNumber* and contains the alternative home phone numbers they indicated upon registration into the system.

4.5.4 FREEFALL.FREEFALLCELLPHONENUMBER

The table *freefall.FreefallCellPhoneNumber* is a sub-table of the table *freefall.TelephoneNumber* and contains randomly-generated cellphone numbers. When a new contract is activated, the first number available is assigned to the contract and marked as unavailable for subsequent contracts.

4.5.5 freefall.FreefallPrefixes

The table *freefall.FreefallPrefixes* contains prefixes of type *freefall.FreefallPrefixes_d*. The prefixes in this table are randomly picked and used to generate a random cellphone number to store in *freefall.FreefallCellPhoneNumber*.

4.5.6 support.LodiProvinceIstatCodes

The table *support.LodiProvinceIstatCodes* stores the ISTAT (Italy's National Statistics Institute) code which uniquely identifies italian cities, along with their names.

4.5.7 freefall.Address

The table *freefall.Address* is an abstract base table and cannot be directly manipulated. It only stores the state name and the city name.

4.5.8 freefall.ItalianAddress

The table *freefall.ItalianAddress* subclasses the table *freefall.Address* and it stores customers' address they indicated upon registration into the system.

4.5.9 freefall.BankAccount

The table *freefall.BankAccount* models a bank account. It is a base abstract table and cannot be directly manipulated. It only stores the account balance.

4.5.10 freefall.ItalianIBAN

The table *freefall.ItalianIBAN* stores italian IBAN codes made up by the state code, the control code, the CIN code, the ABI code, the CAB code and the account number.

4.5.11 freefall.ItalianBankAccount

The table *freefall.ItalianBankAccount* stores italian bank account for customers who subscribed a pay-monthly price plan.

4.5.12 freefall.Customer

The table *freefall.Customer* stores customers information. As soon as a new customer is registered, we insert generic telephone numbers into the right table (*freefall.CellPhoneNumber* or *freefall.HomePhoneNumber*) based on the prefix used. Here comes the advantage of defining custom domains for prefixes, by the way. Moreover, along with telephone numbers, we store the customer's address into the *freefall.ItalianAddress* table.

4.5.13 freefall.SpecialOffer

The table *freefall.SpecialOffer* stores some coll stuff customers can buy at a discounted price from the Freefall Store using a coupon code.

A CHECK constraint is defined in order to make sure at least a 5% discount is applied on the original price.

4.5.14 freefall.OrdinaryPlan

The table *freefall.OrdinaryPlan* stores the plans actually active on the contracts registered. This is a base abstract table and cannot be directly manipulated. The attribute *internationalMessageMultFactor* defines a factor by which the price of calls, SMSs and MMSs towards foreign countries is multiplied, in order to normalize the operation cost in a uniform way when data volume explodes, which is particularly true expecially with video messages. In particular, we use the natural logaritm of the multiplier.

It accepts values in [1.1 - 1.9] by steps of 0.1.

4.5.15 freefall.PayMonthlyPricePlan

The table *freefall.PayMonthlyPricePlan* inherits from *freefall.OrdinaryPlan* and stores plans which allow performing operations unlimitedly. When a customer performs a new operation which month follows that of the last operation he performed, a new bill is generated. If the total amount of the operations performed in [2012-04-20, 2012-05-20] exceeds a certain threshold, a discount is applied.

4.5.16 freefall.PayAndGoPricePlan

The table *freefall.PayAndGoPricePlan* inherits from *freefall.OrdinaryPlan* and stores plans which allow performing operations as long as the balance is enough. If it is not, operations are rejected. These customers can optionally ask for a bill.

4.5.17 freefall.TemporaryPromotion

The table *freefall.TemporaryPromotion* stores temporary promotions defined on contracts. Temporary promotions provisionally replace regular price plans when the customers performs an operation towards his favorite number, the promotion has not expired yet and the freebies available are enough for the current operation to be totally or partly completed.

4.5.18 freefall.Contract

The table *freefall.Contract* stores contracts. When a new contract is activated, a new cellphone number is assigned from those still available from the *freefall.FreefallCellPhoneNumber* table, while when it is deactivated its telephone number is marked as NULL, the cellphone number comes available again for new contracts and subsequent operations on the deactivated contract are refused then on.

4.5.19 freefall.Bill

The table *freefall.Bill* contains bills generated on a monthly basis. When a customer performs an operation, we check its month. If it follows the month of the last operation the customer made, a new bill is generated. *PayMonthlyPricePlan*s can benefit from discounts on the bills' total amount. This happens when the total amount of the operations made in the period [2012-04-20, 2012-05-20] exceeds a certain threshold.

4.5.20 freefall.BillDetail

The table *freefall.BillDetail* stores the details of each operation contained in the generated bill, like the date, the receiver phone number, the duration for calls, the transmitted KBs for messages and so on.

4.5.21 FREEFALL.ERROR

The table *freefall.Error* stores the error registered with the function *register_error_FNC(int, text)*. It contains the bill ID, the description of the error, the date when the error was registered and a flag indicating whether the error has been fixed or not (with the function *unregister_error_FNC(int)*).

4.5.22 FREEFALL.OPERATION

The table *freefall.Operation* is base abstract table ans as such cannot be directly manipulated. It represents operations like messages and calls storing attributes like the operation date, the operation cost, the location where it has been performed and so on.

We opted for a base abstract table because, like addresses and telephone numbers we cannot know in advance what an operation is in general, but we know some of the attributes it must have anyway. In this way we can derive actual operations from this table simply adding the addributes which distinguish the various operations.

4.5.23 FREEFALL.CALL

The table *freefall.Call* inherits from *freefall.Operation* and represents a precise kind of operation: phone calls, eventually video calls by the attribute *isVideo* inherited from its parent table. Here, the attribute distinguishing this kind of operation is the duration in seconds.

4.5.24 freefall.Message

The table *freefall.Message* inherits from *freefall.Operation* and represents another kind of operation: messages, eventually video messages (MMS) by the attribute *isVideo* inherited from its parent table. Here, the distinguishing attribute is the transmitted KBs.

4.5.25 support.TopKCities

The table *support.TopKCities* is functional to the execution of the function *get_top_K_cities_FNC(int)*. It stores the ISTAT codes of the cities which contain each operation. We then count the rows with the same ISTAT code and join with the table *support.LodiProvinceIstatCodes* to get the cities names given the ISTAT code.

4.5.26 support.CustomerPath

The table *support.CustomerPath* contains the set of operations performed by a given customer, represented as points sorted by the date they were performed on. This is obtained with the function *create_customer_path_FNC(int)*. This table contains points of one customer per time and it is used in QGIS as a Postgis layer to draw a given customer's path.

4.5.27 support.TopCustomers

The table *support.TopCustomers* stores is used as the first of the three steps necessary in order to implement activity 5.3.e: show the names of customers who made operations in the majority of cities, showing the number of these different cities in the function *get_top_customers_in_province_FNC()*.

IV.VI VIEWS

We use views as a support for complex queries which involve multiple joins and conditions. This allows us to break big queries into smaller pieces. Moreover, we can reuse them as a basis for different operations which require the same initial tasks.

4.6.1 SUPPORT.TOWNSSORTEDBYGEOGRAPHICAL EXTENSIONS_V

The view `support.TownsSortedByGeographicalExtensions_v` computes the areas of each city's territory and sorts them by their decreasing values. This is done in order to speed up the process of finding the cities where the majority of operations have been performed, under two fundamental assumptions:

1. statistically, and without considering other factors, a wider territory is more likely to include more operations than a less wide territory;
2. a wider territory is symptomatic of the fact that it pertains to a major city and as such - reinforcing the previous point - a more populated territory is more likely to contain more operations than less populated territories.

Standing these considerations, we sort territories by their decreasing extensions and start looking for the territory where an operation falls in wider territories first.

	pro_com integer	city text	area numeric	city_geom geometry
1	98031	Lodi	41379233.04307870	0103000020F8590000010000
2	98049	San Rocco al Porto	30566611.44099041	0103000020F8590000010000
3	98053	Senna Lodigiana	27018993.26177689	0103000020F8590000010000
4	98011	Caselle Landi	26007364.2749743	0103000020F8590000010000
5	98010	Casalpusterlengo	25612566.48627437	0103000020F8590000010000
6	98004	Borghetto Lodigiano	23641447.44531200	0103000020F8590000010000
7	98019	Codogno	20869832.5108005	0103000020F8590000010000
8	98002	Bertonico	20831978.8836399	0103000020F8590000010000
9	98054	Somaglia	20819406.6868561	0103000020F8590000010000
10	98013	Castelnuovo Bocca d'Adda	20325177.9098166	0103000020F8590000010000
11	98050	Sant'Angelo Lodigiano	20053181.1907497	0103000020F8590000010000
12	98035	Maleo	19833744.9505297	0103000020F8590000010000
13	98061	Zelo Buon Persico	18882681.0581772	0103000020F8590000010000
14	98006	Brembio	17083796.76565102	0103000020F8590000010000
15	98032	Lodi Vecchio	16450523.4215952	0103000020F8590000010000
16	98058	Turano Lodigiano	16377512.87698364	0103000020F8590000010000
17	98017	Cavenago d'Adda	16098286.16797306	0103000020F8590000010000
18	98056	Tavazzano con Villavesco	16068624.4688376	0103000020F8590000010000
19	98024	Corte Palasio	15678410.14649548	0103000020F8590000010000
20	98041	Mulazzano	15579219.2897368	0103000020F8590000010000
21	98060	Villanova del Sillaro	13498814.36554798	0103000020F8590000010000
22	98048	San Martino in Strada	13148599.80063019	0103000020F8590000010000
23	98014	Castiglione d'Adda	12978433.2299986	0103000020F8590000010000
24	98020	Comazzo	12802490.2130871	0103000020F8590000010000
25	98007	Camairago	12772773.42209865	0103000020F8590000010000
26	98030	Livraga	12371273.3445506	0103000020F8590000010000
27	98045	Pieve Fissiraga	12265446.5335258	0103000020F8590000010000
28	98044	Ossago Lodigiano	11525173.41392338	0103000020F8590000010000
29	98057	Terranova dei Passerini	11256907.06444162	0103000020F8590000010000
30	98034	Mairago	11247371.14590886	0103000020F8590000010000
31	98028	Graffignana	10921449.4184516	0103000020F8590000010000
32	98039	Merlino	10730328.0961896	0103000020F8590000010000

This would hopefully require much less spatial operations than simply searching the *comuni* table as is, since as soon as an operation is found in one territory, the search can move to the next operation.

Be n the amount of operations and m the amount of cities. Without any assumptions, it would require $n \times m$ comparisons in the worst-case scenario to determine the city where each operation lies in. For n = 50 and m = 61 it would require 3050 comparisons in the worst-case scenario. Now, suppose our trick would let us find the right city scanning most of the times only 1/3rd of the total number of cities and all of them very few times, so that this cost is marginal and we can ignore it. In this case, our trick would

require about 50 x 20 = 1000 comparisons. In the worst case. It is at least a +300% speed up.

We perform this operation on demand, when the amount of operations could be potentially huge, say $n = 10$M at a given point. Our gain is assured to be even greater if projected forward, since when n is that big, the distribution of operations tends to agglomerate much more on wider areas than on less wide ones.

4.6.2 SUPPORT.TOPCUSTOMERSINPROVINCE_V

The view *support.TopCustomersInProvince_v* computes the customers' and the cities' names where each operation has been performed.

	operation_id integer	customer_id integer	name text	surname text	pro_com integer	city text
1	4	3	Ada	Lovelace	98031	Lodi
2	9	3	Ada	Lovelace	98013	Castelnuovo Bocca d'Adda
3	15	3	Ada	Lovelace	98031	Lodi
4	23	3	Ada	Lovelace	98014	Castiglione d'Adda
5	26	3	Ada	Lovelace	98060	Villanova del Sillaro
6	44	3	Ada	Lovelace	98031	Lodi
7	55	3	Ada	Lovelace	98058	Turano Lodigiano
8	1	4	Dennis	Ritchie	98030	Livraga
9	11	4	Dennis	Ritchie	98030	Livraga
10	12	4	Dennis	Ritchie	98006	Brembio
11	21	4	Dennis	Ritchie	98052	Secugnago
12	24	4	Dennis	Ritchie	98061	Zelo Buon Persico
13	27	4	Dennis	Ritchie	98010	Casalpusterlengo
14	28	4	Dennis	Ritchie	98061	Zelo Buon Persico
15	31	4	Dennis	Ritchie	98061	Zelo Buon Persico
16	32	4	Dennis	Ritchie	98034	Mairago
17	34	4	Dennis	Ritchie	98042	Orio Litta
18	35	4	Dennis	Ritchie	98042	Orio Litta
19	37	4	Dennis	Ritchie	98053	Senna Lodigiana
20	39	4	Dennis	Ritchie	98054	Somaglia
21	41	4	Dennis	Ritchie	98029	Guardamiglio
22	45	4	Dennis	Ritchie	98006	Brembio
23	46	4	Dennis	Ritchie	98006	Brembio
24	47	4	Dennis	Ritchie	98014	Castiglione d'Adda
25	50	4	Dennis	Ritchie	98031	Lodi
26	51	4	Dennis	Ritchie	98031	Lodi
27	54	4	Dennis	Ritchie	98031	Lodi
28	5	5	Carl	Sagan	98037	Massalengo
29	13	5	Carl	Sagan	98018	Cervignano d'Adda
30	14	5	Carl	Sagan	98041	Mulazzano
31	19	5	Carl	Sagan	98017	Cavenago d'Adda
32	6	6	Alan	Turing	98056	Tavazzano con Villavesco

4.6.3 support.CitiesContainingWideGreenAreas_v

The view *support.CitiesContainigWideGreenAreas_v* computes all those cities which include a wide green area. A wide green area is a green area with an extension greater than or equal to 15KM².

	gid integer	pro_com integer	city text	area double precision	city_geom geometry
1	27013	98031	Lodi	15032.4258850156	0103000020F8590000010000
2	25466	98031	Lodi	32644.7445315201	0103000020F8590000010000
3	25322	98031	Lodi	15576.8140065379	0103000020F8590000010000
4	23715	98049	San Rocco al Porto	29171.1214709773	0103000020F8590000010000
5	911	98010	Casalpusterlengo	20472.8692264669	0103000020F8590000010000
6	2225	98010	Casalpusterlengo	162616.435278417	0103000020F8590000010000
7	2219	98010	Casalpusterlengo	134206.980834013	0103000020F8590000010000
8	2165	98010	Casalpusterlengo	20053.5394531137	0103000020F8590000010000
9	2182	98010	Casalpusterlengo	35035.0512354946	0103000020F8590000010000
10	7347	98010	Casalpusterlengo	18540.5048140219	0103000020F8590000010000
11	7356	98010	Casalpusterlengo	21104.7253799955	0103000020F8590000010000
12	713	98010	Casalpusterlengo	18343.676507224	0103000020F8590000010000
13	6918	98010	Casalpusterlengo	36191.3748875808	0103000020F8590000010000
14	1152	98010	Casalpusterlengo	23573.4952599978	0103000020F8590000010000
15	2191	98010	Casalpusterlengo	15987.1547634767	0103000020F8590000010000
16	2226	98010	Casalpusterlengo	48544.100624029	0103000020F8590000010000
17	18460	98004	Borghetto Lodigiano	23023.8434600006	0103000020F8590000010000
18	18908	98004	Borghetto Lodigiano	25705.9549580088	0103000020F8590000010000
19	23726	98019	Codogno	554154.574448396	0103000020F8590000010000
20	12717	98002	Bertonico	15360.6578214552	0103000020F8590000010000
21	13270	98002	Bertonico	18448.6369264964	0103000020F8590000010000
22	282	98002	Bertonico	42969.5554914707	0103000020F8590000010000
23	13243	98002	Bertonico	23718.3878039503	0103000020F8590000010000
24	13224	98002	Bertonico	20659.7956930854	0103000020F8590000010000
25	1310	98002	Bertonico	87539.699650067	0103000020F8590000010000
26	2313	98002	Bertonico	28720.9332706031	0103000020F8590000010000
27	2314	98002	Bertonico	15345.5611065198	0103000020F8590000010000
28	2323	98002	Bertonico	16708.869789097	0103000020F8590000010000
29	2335	98002	Bertonico	31105.3285295158	0103000020F8590000010000
30	2336	98002	Bertonico	15708.7752715374	0103000020F8590000010000
31	2366	98002	Bertonico	32807.2824819879	0103000020F8590000010000
32	2367	98002	Bertonico	65936.7605032168	0103000020F8590000010000

4.6.4 SUPPORT.CITIESBORDERING_V

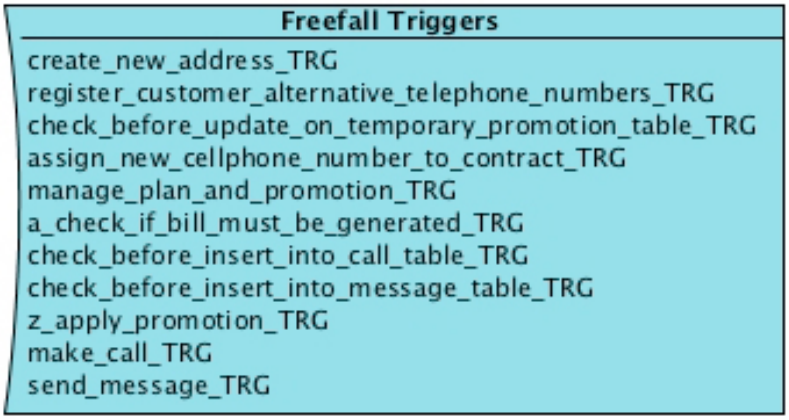
The view *support.CitiesBordering_v* computes all those cities containg at least 1 wide green area bordering cities containing at least 1 wide green area. This is done joining the view *support.CitiesContainigWideGreenAreas_v* with itself with the join condition being a spatial operation: *ST_TOUCHES(geom1, geom2)* which tells if *geom1* borders *geom2*.

	gid integer	pro_com integer	city text	area double precision	city_geom geometry
1	14	98007	Camairago	42373.7860570028	0103000020F8590000010000
2	27	98007	Camairago	26310.4291234552	0103000020F8590000010000
3	75	98016	Cavacurta	17565.1156394898	0103000020F8590000010000
4	213	98007	Camairago	36119.2493875195	0103000020F8590000010000
5	253	98014	Castiglione d'Adda	34287.1634184534	0103000020F8590000010000
6	282	98002	Bertonico	42969.5554914707	0103000020F8590000010000
7	713	98010	Casalpusterlengo	18343.676507224	0103000020F8590000010000
8	737	98007	Camairago	25340.5540385103	0103000020F8590000010000
9	911	98010	Casalpusterlengo	20472.8692264669	0103000020F8590000010000
10	1152	98010	Casalpusterlengo	23573.4952599978	0103000020F8590000010000
11	1310	98002	Bertonico	87539.699650067	0103000020F8590000010000
12	2165	98010	Casalpusterlengo	20053.5394531137	0103000020F8590000010000
13	2182	98010	Casalpusterlengo	35035.0512354946	0103000020F8590000010000
14	2191	98010	Casalpusterlengo	15987.1547634767	0103000020F8590000010000
15	2193	98057	Terranova dei Passerini	17706.8460761137	0103000020F8590000010000
16	2207	98057	Terranova dei Passerini	31086.9979354853	0103000020F8590000010000
17	2215	98057	Terranova dei Passerini	17796.7943990255	0103000020F8590000010000
18	2219	98010	Casalpusterlengo	134206.980834013	0103000020F8590000010000
19	2225	98010	Casalpusterlengo	162616.435278417	0103000020F8590000010000
20	2226	98010	Casalpusterlengo	48544.100624029	0103000020F8590000010000
21	2313	98002	Bertonico	28720.9332706031	0103000020F8590000010000
22	2314	98002	Bertonico	15345.5611065198	0103000020F8590000010000
23	2323	98002	Bertonico	16708.869789097	0103000020F8590000010000
24	2335	98002	Bertonico	31105.3285295158	0103000020F8590000010000
25	2336	98002	Bertonico	15708.7752715374	0103000020F8590000010000
26	2351	98002	Bertonico	16548.890554544	0103000020F8590000010000
27	2354	98002	Bertonico	34423.256297948	0103000020F8590000010000
28	2364	98002	Bertonico	30019.0227738865	0103000020F8590000010000
29	2366	98002	Bertonico	32807.2824819879	0103000020F8590000010000
30	2367	98002	Bertonico	65936.7605032168	0103000020F8590000010000
31	2370	98058	Turano Lodigiano	29832.6477700021	0103000020F8590000010000
32	3517	98034	Mairago	20104.2882589842	0103000020F8590000010000

IV.VII TRIGGERS DEFINITION

We now introduce the triggers (aka *active rules*) which give **Freefall** its active behavior. Triggers listen for operations performed on the tables they're defined on and may be used, for example, as a powerful mean of checking integrity of data input and react accordingly, i.e. refusing or accepting the input data, throwing exceptions and much more. Other times, they can be used to implement Objects Composition. In fact, the Composition's semantic implies that if the container object is deleted, the contained object must too.

Here is the list of all triggers defined in **Freefall**:



4.7.1 CREATE_NEW_ADDRESS_TRG

```
CREATE TRIGGER create_new_address_TRG
AFTER INSERT ON freefall.Customer
FOR EACH ROW
EXECUTE PROCEDURE freefall.create_new_address_FNC();
```

4.7.2 REGISTER_CUSTOMER_ALTERNATIVE_TELEPHONE_NUMBERS_TRG

```
CREATE TRIGGER register_customer_alternative_telephone_numbers_TRG
BEFORE INSERT ON freefall.Customer
FOR EACH ROW
EXECUTE PROCEDURE
freefall.register_customer_alternative_telephone_numbers_FNC();
```

4.7.3 CHECK_BEFORE_UPDATE_ON_TEMPORARY_TABLE_TRG

```
CREATE TRIGGER check_before_update_on_temporary_promotion_table_TRG
BEFORE UPDATE OF favoriteNumber_REF ON freefall.TemporaryPromotion
FOR EACH ROW
EXECUTE PROCEDURE
freefall.check_before_update_on_temporary_promotion_table_FNC();
```

4.7.4 ASSIGN_NEW_CELLPHONE_NUMBER_TO_CONTRACT_TRG

```
CREATE TRIGGER assign_new_cellphone_number_to_contract_TRG
AFTER INSERT ON freefall.Contract
FOR EACH ROW
EXECUTE PROCEDURE freefall.assign_new_cellphone_number_to_contract_FNC();
```

4.7.5 MANAGE_PLAN_AND_PROMOTION_TRG

```
CREATE TRIGGER manage_plan_and_promotion_TRG
AFTER DELETE ON freefall.Contract
FOR EACH ROW
EXECUTE PROCEDURE freefall.manage_plan_and_promotion_FNC();
```

4.7.6 A_CHECK_IF_BILL_MUST_BE_GENERATED_TRG

```
CREATE TRIGGER a_check_if_bill_must_be_generated_TRG
AFTER INSERT ON freefall.Call
FOR EACH ROW
EXECUTE PROCEDURE freefall.check_if_bill_must_be_generated_FNC();
```

```
CREATE TRIGGER a_check_if_bill_must_be_generated_TRG
AFTER INSERT ON freefall.Message
FOR EACH ROW
EXECUTE PROCEDURE freefall.check_if_bill_must_be_generated_FNC();
```

4.7.7 CHECK_BEFORE_INSERT_INTO_CALL_TABLE_TRG

```
CREATE TRIGGER check_before_insert_into_call_table_TRG
BEFORE INSERT ON freefall.Call
FOR EACH ROW
EXECUTE PROCEDURE freefall.check_before_insert_into_operation_table_FNC();
```

4.7.8 CHECK_BEFORE_INSERT_INTO_MESSAGE_TABLE_TRG

```
CREATE TRIGGER check_before_insert_into_message_table_TRG
BEFORE INSERT ON freefall.Message
FOR EACH ROW
EXECUTE PROCEDURE freefall.check_before_insert_into_operation_table_FNC();
```

4.7.9 Z_APPLY_PROMOTION_TRG

```
CREATE TRIGGER z_apply_promotion_TRG
AFTER INSERT ON freefall.Call
FOR EACH ROW
EXECUTE PROCEDURE freefall.z_apply_promotion_FNC();
```

```
CREATE TRIGGER z_apply_promotion_TRG
AFTER INSERT ON freefall.Message
FOR EACH ROW
EXECUTE PROCEDURE freefall.z_apply_promotion_FNC();
```

4.7.10 MAKE_CALL_TRG

```
CREATE TRIGGER make_call_TRG
AFTER INSERT ON freefall.Call
FOR EACH ROW
EXECUTE PROCEDURE freefall.make_call_FNC();
```

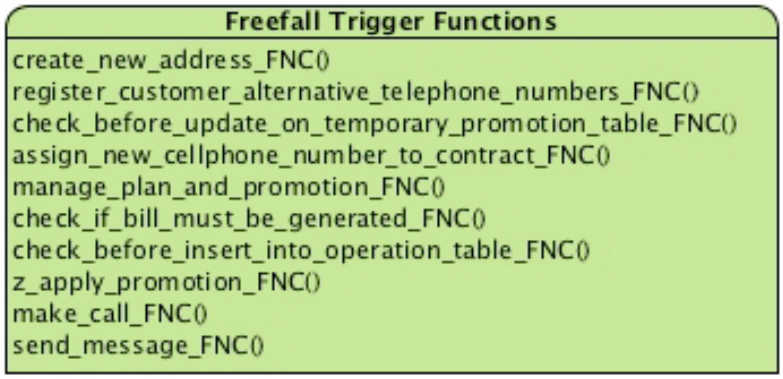
4.7.11 SEND_MESSAGE_TRG

```
CREATE TRIGGER send_message_TRG
AFTER INSERT ON freefall.Message
FOR EACH ROW
EXECUTE PROCEDURE freefall.send_message_FNC();
```

IV.VIII TRIGGER FUNCTIONS DEFINITION

We call *Trigger functions* stored procedures invoked when an event (INSERT, UPDATE [OF], DELETE) defined on a trigger is verified.

Below is the list of all **Freefall**'s trigger functions:



4.8.1 CREATE_NEW_ADDRESS_FNC()

The trigger function *create_new_address_FNC()* extracts the customer's address when a new customer is registered and adds it to the table *freefall.ItalianAddress*.

4.8.2 REGISTER_CUSTOMER_ALTERNATIVE_TELEPHONE_NUMBERS_FNC()

The trigger function *register_customer_alternative_telephone_numbers_FNC()* checks whether the REFs to *freefall.TelephoneNumber* are actually valid, in which case the operation is completed, otherwise an exception is raised and the operation aborted.

4.8.3 CHECK_BEFORE_UPDATE_ON_TEMPORARY_PROMOTION_TABLE_FNC()

The trigger function *check_before_update_on_temporary_promotion_table_FNC()* checks whether the reference to the favorite number is valid, that is whether it has been already assigned to a customer.

4.8.4 ASSIGN_NEW_CELLPHONE_NUMBER_TO_CONTRACT_FNC()

The trigger function *assign_new_cellphone_number_to_contract_FNC()* scans the *FreefallCellPhoneNumber* table and finds the first cellphone number generated by *populate_freefall_cellphone_numbers_table_FNC()* which *assigned* field is FALSE, then sets it to TRUE and assigns this number to the new contract.

4.8.5 MANAGE_PLAN_AND_PROMOTION_FNC()

The trigger function *manage_plan_and_promotion_FNC()* implements object composition between *freefall.Contract* and *freefall.OrdinaryPlan* tables and between *freefall.Contract* and *freefall.TemporaryPromotion* tables.

4.8.6 CHECK_IF_BILL_MUST_BE_GENERATED_FNC()

The trigger function *check_if_bill_must_be_generated_FNC()* checks whether a new bill must be generated. A new bill is generated when the new operation's month is greater by 1 than those stored into the *freefall.Operation* table for a given contract. In this way, we put into the new bill only those operations performed in the month preceding the one of the new operation.

Then, regardless of the fact that a new bill has been generated or not, we check to see if a discount can be applied.

4.8.7 CHECK_BEFORE_INSERT_INTO_OPERATION_TABLE_FNC()

The trigger function `check_before_insert_into_operation_table_FNC()` executes a series of checks before operations are actually performed.

In particular, we check that:

- the contract reference is actually valid;
- the contract is active (its cellphone number is not null);
- the balance is enough for the operation to be started;
- the destination number's prefix is valid;
- the source and destination number are different.

4.8.8 Z_APPLY_PROMOTION_FNC()

The trigger function `z_apply_promotion_FNC()` automatically activates a temporary promotion for each `PayAndGoPricePlan` for which an expense of at least 150€ has been registered in the period [2012-04-20, 2012-05-20]. If the customer already has an active temporary promotion, its thresholds are updated, otherwise a new one is created.

One problem is that of determining what would be the favorite number when a new promotion is activated. Since there's no plausible way of knowing it a priori, we decide that when a new `TemporaryPromotion` is activated, its reference to the favorite number is set to NULL. Then an UPDATE operation is required to specify the favorite number imitating the fact that even if the promotion has been (*automatically*) activated, then a call to the operator by the customer is necessary to associate that

promotion with his favorite number. In the meantime, he will not benefit from it.

In order to simplify things a little bit, we let customers benefit from a promotion only once. In the future, we could add the date when the promotion has been activated and let future promotions be activated only once in - say - 6 or 9 months.

4.8.9 MAKE_CALL_FNC()

The trigger function *make_call_FNC()* performs calls provided that the checks specified in *check_before_insert_into_operation_table_FNC()* have been passed. It prints some useful information on the **Freefall**'s big screen (remember from Section 1.11?) regarding the operation in place, like the date, the destination number, etc.

Then it checks whether a promotion is applicable. If it is and the free seconds left are enough for the current operation to be totally completed, then the operation is completed, the duration of the call is subtracted from the free seconds left and the operation cost is calculated on the actual duration, which is zero, and as such the cost of the opration will be zero, too. Instead, if the free seconds left are less than the duration of the operation, free seconds are set to zero and the actual duration is the original duration minus the free seconds left before they were set to zero.

If no promotion is applicable, the operation cost is then calculated dinamically based on various parameters, like the mode of the operation (video and/or international), the duration of the operation and the price of the operation for each one of the modes.

Then the *operationCost* field in *freefall.Call* is updated, the balance updated accordingly and the operation completed.

4.8.10 SEND_MESSAGE_FNC()

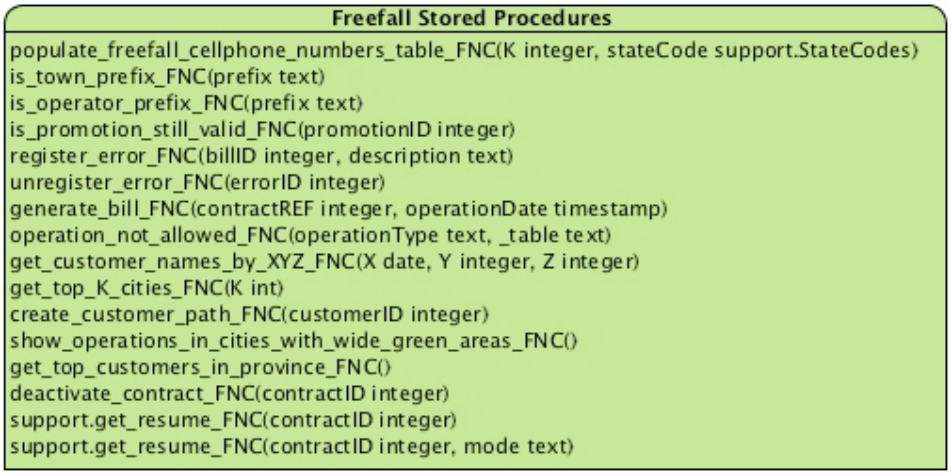
The trigger function *send_message_FNC()* performs messaging operations when the checks specified in *check_before_insert_into_operation_table_FNC()* have been passed.

It behaves in the same manner as the trigger function *make_call_FNC()* above: it prints some useful information, checks for a promotion to be applied and computes the operation cost based on various parameters, then the *operationCost* field in *freefall.Message* is updated, the balance updated accordingly and the operation completed.

IV.IX STORED PROCEDURES DEFINITION

Stored procedures are used to implement custom operations, like registering and unregistering errors from bills, deactivate contracts, get statistics on customers and so on.

The list of all stored procedures defined in **Freefall** follows:



4.9.1 POPULATE_FREEFALL_CELLPHONE_NUMBERS_TABLE_FNC (K INTEGER, STATECODE SUPPORT.STATECODES_D)

The stored procedure *populate_freefall_cellphone_numbers_table_FNC()* is one of the extensions built into **Freefall**.

It generates K cellphone numbers which prefix is one of those present in *freefall.FreefallPrefixes* and puts them in the table *freefall.FreefallCellPhoneNumbers* with the attribute *assigned* initially set to FALSE. If, during an iteration, a number is generated that has already been generated previously, it attempts to generated [K - *alreadyGeneratedNumbers*] new cellphone numbers by recursively invoking itself. The process is assured to stop at some point, because the condition over K (K > 0) will be evaluated as false. All this is only theorethical, as the actual realization of the function is slightly different: since we have to provide a case in which a customer calls his favorite number, we generate 3 static freefall cellphone numbers first and then K-3 random freefall cellphone numbers. So, in the actual realization, if at some point a duplicate freefall cellphone number is generated, the recursive call would generate the same initial 3 static freefall cellphone numbers again, but since the 3 static cellphone numbers are only a workaround to test the case when a customer calls his favorite number, this does not worry us, in fact in real conditions these 3 INSERTs would be absent.

4.9.2 IS_TOWN_PREFIX_FNC (PREFIX TEXT)

The stored procedure *is_town_prefix_FNC()* checks to see whether an alternative number specified upon a new customer registration is a home phone number. It is used in conjunction with the stored procedure *is_operator_prefix_FNC()* che reject the operation in the case the specified prefix is nor a town prefix or a town prefix.

4.9.3 IS_OPERATOR_PREFIX_FNC (PREFIX TEXT)

The stored procedure *is_operator_prefix_FNC()* is used in two ways:

- alone, to determine in which of either *freefall.HomePhoneNumber* or *freefall.CellPhoneNumber* store each of the alternative numbers specified when a new customer is registered; If it is an operator prefix we store the new number into the *freefall.CellPhoneNumber* table, otherwise we store it in the *freefall.HomePhoneNumber* table.
- in conjunction with the stored procedure *is_town_prefix_FNC()*, to reject the operation in the case the specified prefix is nor an operator prefix or a town prefix.

4.9.4 IS_PROMOTION_STILL_VALID_FNC (PROMOTIONID INTEGER)

The stored procedure *is_promotion_still_valid_FNC()* checks that an active temporary promotion is actually valid, that is all of the following conditions are evaluated as TRUE:

1. the specified *promotionID* is found in *freefall.TemporaryPromotion;*
2. *freeSMSsLeft*, *freeSecondsLeft* and *freeKBsLeft* are not 0*;*
3. current timestamp is not greater than *dateValidUntil*;

4.9.5 REGISTER_ERROR_FNC (BILLID INTEGER, DESCRIPTION TEXT)

The stored procedure *register_error_FNC()* creates a new error associated to a bill, setting the *fixed* attribute to FALSE.

4.9.6 UNREGISTER_ERROR_FNC
(ERRORID INTEGER)

The stored procedure *unregister_error_FNC()* unregisters an error previously created with the stored procedure *register_error_FNC()*. Fixing an error simply consists in setting the *fixed* attribute to TRUE.

4.9.7 GENERATE_BILL_FNC
(CONTRACTREF INTEGER, DESCRIPTION TEXT)

The stored procedure *generate_bill_FNC()* generates a new bill. The tables *freefall.Call* and *freefall.Message* are scanned and operations performed in the month preceding that of *operationDate* and pertaining to the same *contractREF* are included in the new bill. The bill total is then computed by summing the total amounts of all the included operations.

4.9.8 OPERATION_NOT_ALLOWED_FNC
(OPERATIONTYPE TEXT, _TABLE TEXT)

The stored procedure *operation_not_allowed_FNC()* is fired by rules defined on parent abstract table and raises an exception which prevents operations from being executed on these tables.

4.9.9 DEACTIVATE_CONTRACT_FNC
(CONTRACTID INTEGER)

The stored procedure *deactivate_contract_FNC()* sets the contract's *cellPhoneNumber_REF* to NULL, so that new operations are prevented by the stored procedure *check_before_insert_into_operation_table_FNC()*.

Moreover, it sets the old contract's freefall cellphone number *assigned* attribute to FALSE, so that it can be assigned again on a new contract.

4.9.10 GET_RESUME_FNC (CONTRACTID INTEGER)

The stored procedure *get_resume_FNC()* prints the basic contract's info included in the type *support.Resume_t* described in section 4.4.1.

4.9.11 GET_RESUME_FNC (CONTRACTID INTEGER, MODE TEXT)

The polymorphic stored procedure *get_resume_FNC()* takes an additional *mode* parameter, which causes printing of additional contract's info when set to *EXTENDED*. These additional info are described in section 4.4.2.

Stored procedures *get_resume_FNC(contractID integer)* and *get_resume_FNC(contractID integer, mode text)* are other extensions made in **Freefall**.

4.9.12 GET_CUSTOMER_NAMES_BY_XYZ_FNC (X DATE, Y INTEGER, Z INTEGER)

The stored procedure *get_customer_names_by_XYZ_FNC()* returns the names of customers who, on day X, made more than Y operations in places which distance from Lodi's city centre is less than Z metres.

Examples follow:

	name text	surname text	operation_id integer	date date	query distance integer	computed distance numeric
1	Ada	Lovelace	15	2012-04-23	8000	480.61
2	Alan	Turing	7	2012-04-23	8000	2410.07
3	Alan	Turing	8	2012-04-23	8000	7998.26

```
SELECT * FROM get_customer_names_by_XYZ_FNC('2012-04-23', 0, 8000);
```

	name text	surname text	operation_id integer	date date	query distance integer	computed distance numeric
1	Ada	Lovelace	15	2012-04-23	16000	480.61
2	Alan	Turing	7	2012-04-23	16000	2410.07
3	Alan	Turing	8	2012-04-23	16000	7998.26
4	Alan	Turing	10	2012-04-23	16000	11072.30
5	Carl	Sagan	13	2012-04-23	16000	12999.23
6	Carl	Sagan	14	2012-04-23	16000	15307.28

```
SELECT * FROM get_customer_names_by_XYZ_FNC('2012-04-23', 1, 16000);
```

	name text	surname text	operation_id integer	date date	query distance integer	computed distance numeric
1	Ada	Lovelace	44	2012-05-15	150000	1382.29
2	Dennis	Ritchie	46	2012-05-15	150000	17304.06
3	Dennis	Ritchie	45	2012-05-15	150000	17726.05
4	John	Von Neumann	43	2012-05-15	150000	25561.38
5	Dennis	Ritchie	47	2012-05-15	150000	26195.41
6	John	Von Neumann	42	2012-05-15	150000	26493.36

```
SELECT * FROM get_customer_names_by_XYZ_FNC('2012-05-15', 1, 150000);
```

4.9.13 GET_TOP_K_CITIES_FNC (K INT)

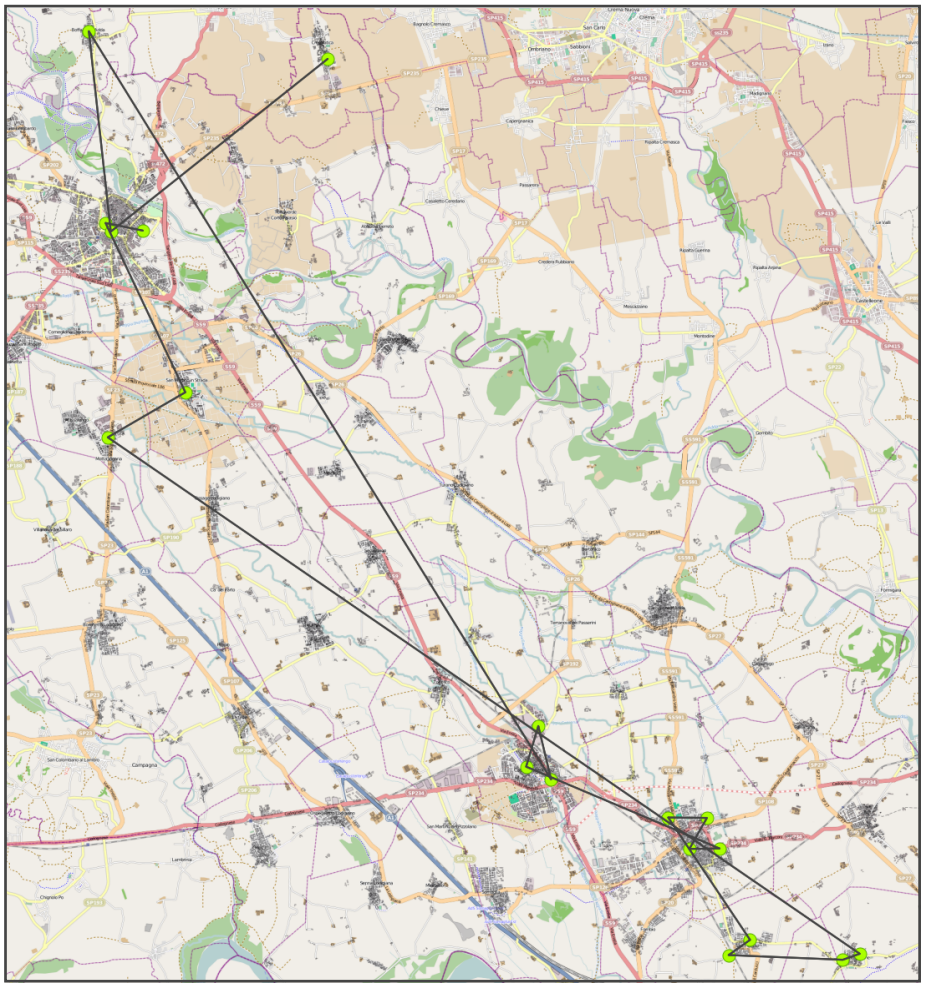
The stored procedure *get_top_K_cities_FNC()* shows the names of the K cities where the majority of operations have been performed. The list is ordered by the decreasing values of number of operations.

	istat code integer	city text	# operations bigint
1	98031	Lodi	10
2	98010	Casalpusterlengo	4
3	98019	Codogno	4
4	98006	Brembio	3
5	98037	Massalengo	3
6	98061	Zelo Buon Persico	3
7	98014	Castiglione d'Adda	2
8	98022	Corno Giovine	2
9	98030	Livraga	2
10	98042	Orio Litta	2

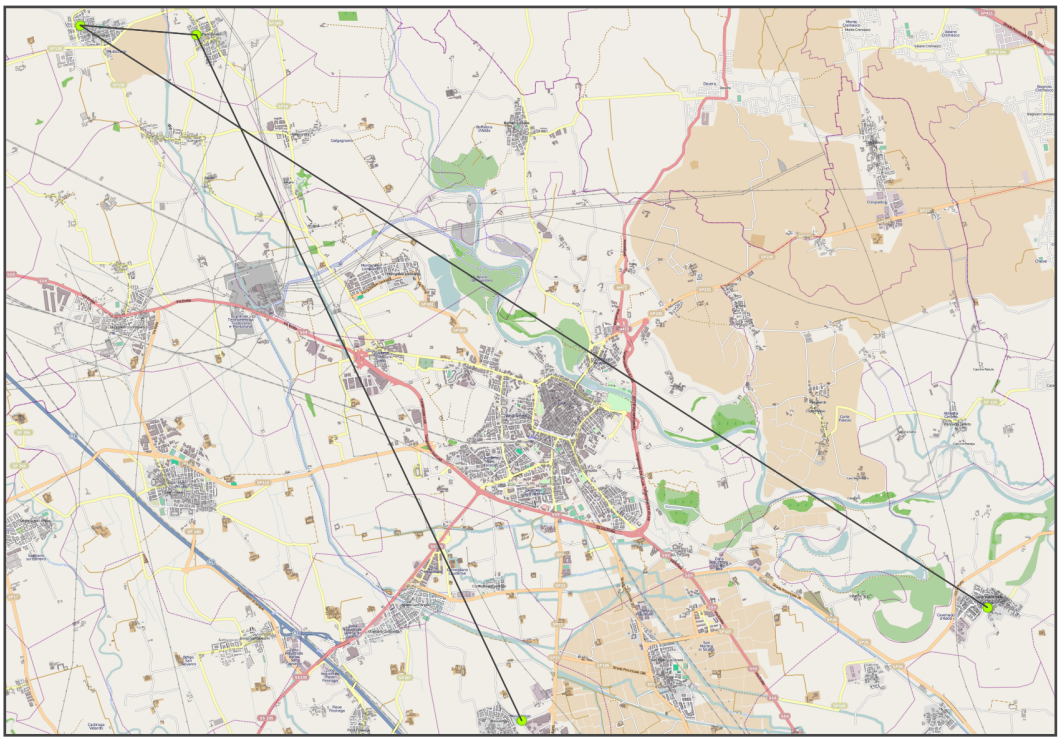
```
SELECT * FROM get_top_K_cities_FNC(10);
```

4.9.14 CREATE_CUSTOMER_PATH_FNC (CUSTOMERID INT)

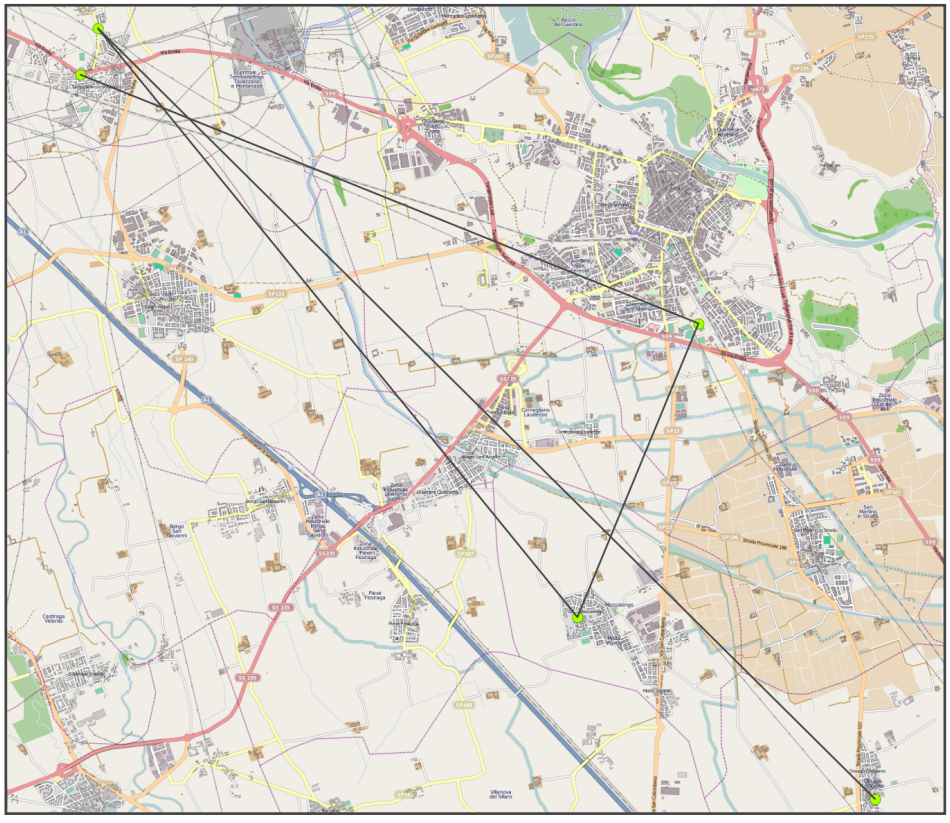
The stored procedure `create_customer_path_FNC()` returns a set of points representing all the operations performed by `customerID`, ordered by date.



```
SELECT * FROM create_customer_path_FNC(2);
```



```
SELECT * FROM create_customer_path_FNC(5);
```



```
SELECT * FROM create_customer_path_FNC(6);
```

4.9.15 SHOW_OPERATIONS_IN_CITIES_WITH_WIDE_GREEN_AREAS_FNC()

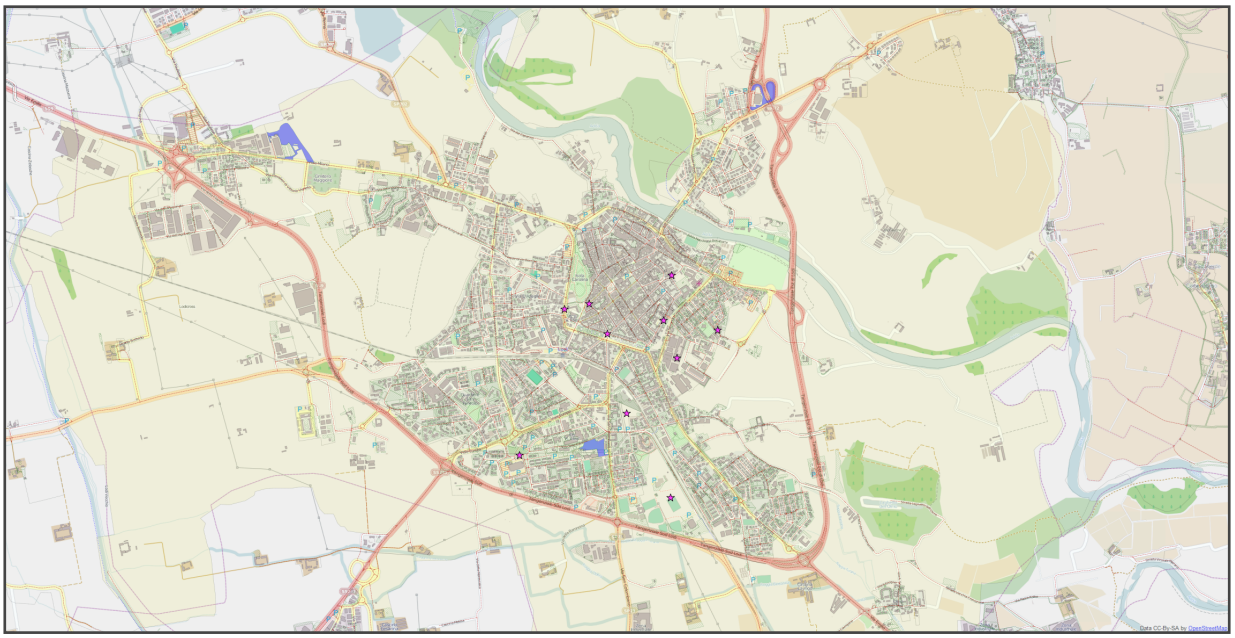
The stored procedure *show_operations_in_cities_with_wide_green_areas_FNC()* shows how many operations have been performed in a city where the following two conditions are true:

1. at least one wide green area is present;
2. it borders a city where at least one wide green area is present.

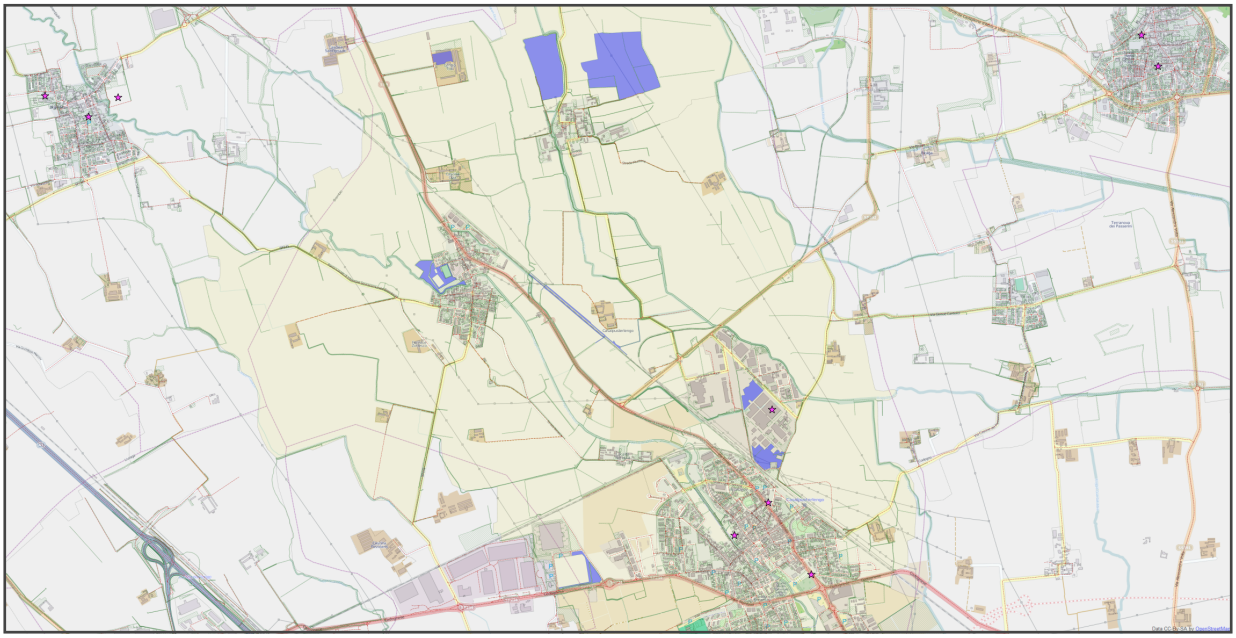
The following image shows the IDs of green areas contained in the top ranked cities:

	gid integer	extension numeric	istat integer	city text	# operations bigint
1	25322	15576.81	98031	Lodi	10
2	27013	15032.42	98031	Lodi	10
3	25466	32644.74	98031	Lodi	10
4	1152	23573.49	98010	Casalpusterlengo	4
5	2165	20053.53	98010	Casalpusterlengo	4
6	2182	35035.05	98010	Casalpusterlengo	4
7	2191	15987.15	98010	Casalpusterlengo	4
8	2219	134206.98	98010	Casalpusterlengo	4
9	2225	162616.43	98010	Casalpusterlengo	4
10	2226	48544.10	98010	Casalpusterlengo	4
11	6918	36191.37	98010	Casalpusterlengo	4
12	7347	18540.50	98010	Casalpusterlengo	4
13	7356	21104.72	98010	Casalpusterlengo	4
14	713	18343.67	98010	Casalpusterlengo	4
15	911	20472.86	98010	Casalpusterlengo	4
16	23726	554154.57	98019	Codogno	4
17	9737	17427.60	98006	Brembio	3
18	10145	15939.75	98006	Brembio	3
19	253	34287.16	98014	Castiglione d'Adda	2
20	26634	23447.06	98056	Tavazzano con Villavesco	2
21	26466	15572.57	98056	Tavazzano con Villavesco	2
22	11047	17589.03	98017	Cavenago d'Adda	1
23	13433	32123.30	98017	Cavenago d'Adda	1
24	11991	29902.02	98025	Crespiatica	1
25	12281	20774.23	98025	Crespiatica	1
26	11975	17080.52	98025	Crespiatica	1
27	8399	17340.95	98034	Mairago	1
28	13292	47431.99	98034	Mairago	1
29	3517	20104.28	98034	Mairago	1
30	9620	20128.44	98044	Ossago Lodigiano	1
31	9824	15076.16	98048	San Martino in Strada	1
32	11414	22486.60	98048	San Martino in Strada	1
33	10042	23479.23	98052	Secugnago	1
34	4153	23588.70	98052	Secugnago	1
35	23652	31016.82	98054	Somaglia	1
36	23669	21910.67	98054	Somaglia	1
37	23671	40025.42	98054	Somaglia	1
38	2370	29832.64	98058	Turano Lodigiano	1
39	7853	21831.31	98058	Turano Lodigiano	1
40	8127	16502.00	98058	Turano Lodigiano	1

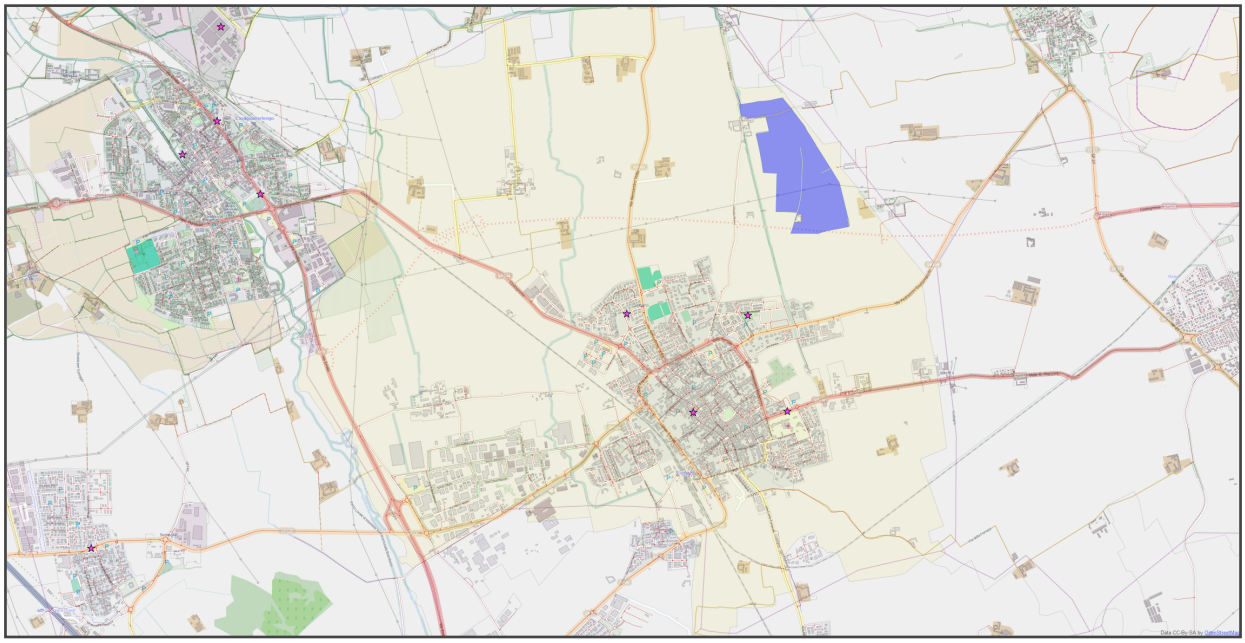
This table was used to identify the green areas in QGIS, which are shown in purple in the examples below along with the operations performed:



Lodi



Casalpusterlengo



Codogno

4.9.16 GET_TOP_CUSTOMERS_IN_PROVINCE_FNC()

The stored procedure *get_top_customers_in_province_FNC()* shows the names of customers who made operations in the majority of cities.

	customer_id integer	name text	surname text	# cities bigint
1	4	Dennis	Ritchie	12
2	2	John	Von Neumann	9
3	3	Ada	Lovelace	5
4	5	Carl	Sagan	4
5	6	Alan	Turing	4

```
SELECT * FROM get_top_customers_in_province_FNC();
```

V. OUTPUT

Here you can find the complete output of the **Freefall** script.

VI. FURTHER REFERENCES

http://goo.gl/SDXtb
Development process on my blog

http://goo.gl/81q2m
PostgreSQL Documentation

http://goo.gl/4f5yH
Stackoverflow.com

http://goo.gl/U6Wp8
Course Web Page

http://goo.gl/5it6b
Object-Relational Mapping

http://goo.gl/e31kc
PostgreSQL examples

http://goo.gl/L6usQ
Difference between Postgis and PostgreSQL geom columns

http://goo.gl/f9G3x
Some QGIS tutorials

http://goo.gl/iTXj8
OpenStreetMap Wiki about Italian borders

http://goo.gl/NWiF2
Article about indices in PostgreSQL

http://goo.gl/cndqT
PostgreSQL Inheritance

http://goo.gl/2uOxt
PostgreSQL Composite Types

http://goo.gl/nBdw8
PostgreSQL Error Codes