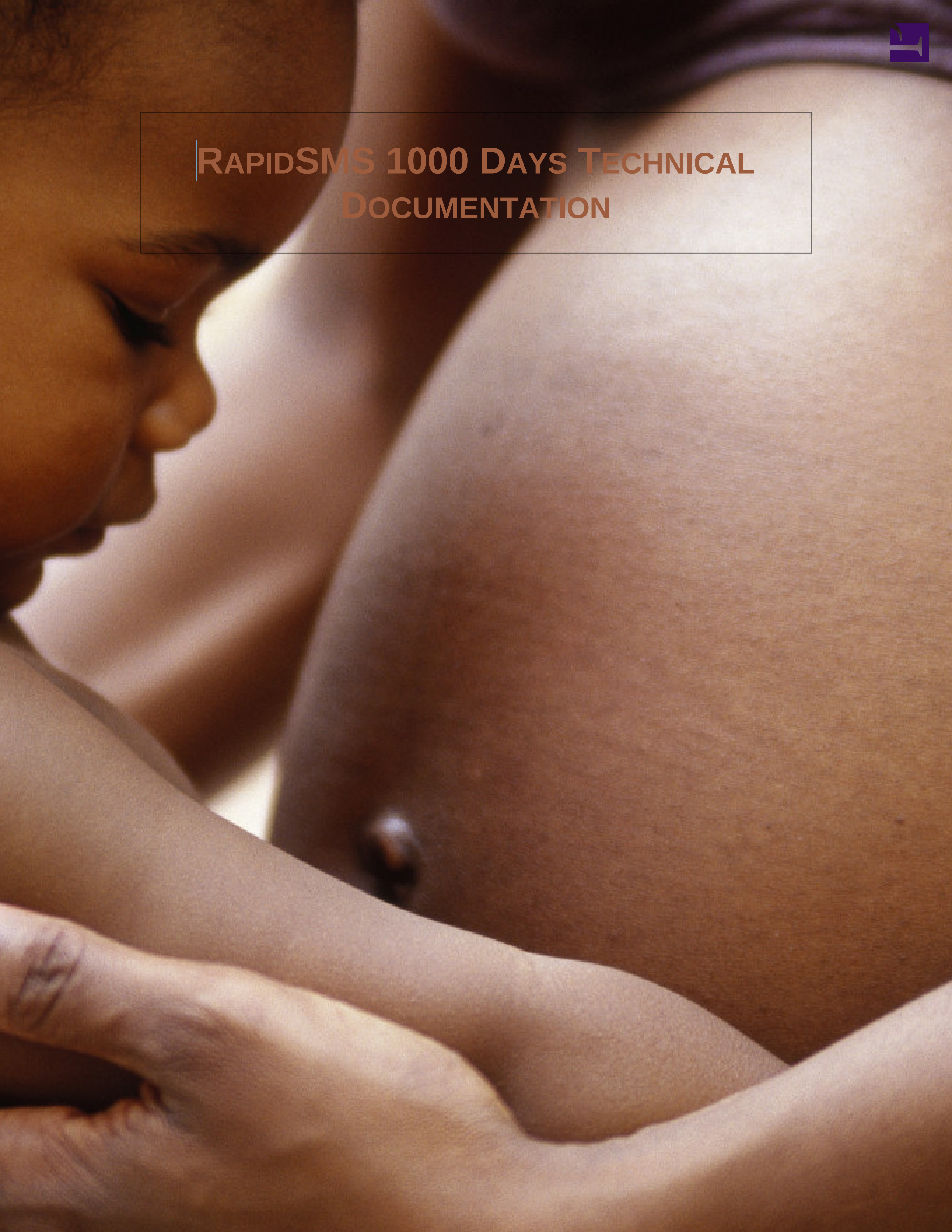


RAPIDSMS 1000 DAYS TECHNICAL DOCUMENTATION

RapidSMS 1000 Days Technical Documentation

Table of Contents

2014-05-28 21:56:24 +0300

AUDIENCE

The audience of this document includes both the technical personel responsible for the day-to-day operation of the *RapidSMS 1000 Days* system and the project managers of UNICEF Rwanda and Pivot Access.

In particular, the programmers of the system will find information that complements the documentation available with the source code. Tutorials, FAQs, tips, tricks, and general guides for programmers shall be included in this document.

System administrators will also find the *RapidSMS 1000 Days* system requirements and dependencies specified in this document, to facilitate installation on a fresh server.

THE DATABASE SYSTEM

THE TWO SENSES OF "SCALING"

Not every relational database management system (RDBMS) is suited for the same tasks; and, often, as a project evolves it grows out of one RDBMS and works better with another. The good system designer knows when that time arrives, and makes the decision accordingly.

Database systems are designed and built with several considerations and trade-offs in mind, and two of them are relevant to the concept of "scaling" as we will discuss it in the present section. These are also the main considerations that have influenced the database decisions in this phase of the project.

It should be noted that trade-offs are necessitated in engineering by the hard physical limits that are a fact of nature. An algorithm that works best on large amounts of data is usually ill-suited for small amounts of data, and would be wasteful if used. Similarly, naïve algorithms are often easy to implement and cheap to execute, but will often break down when they encounter unusual input.

An example that could be given is that a house designed for a small family of five would not be able to accomodate a large family of twelve. Yet since most nuclear families are small, one finds that most houses can comfortably accomodate five people. It would be wasteful to build large houses as a matter of routine, in a culture where one finds no extended families. If this were to change, so would the average house size.

SCALING WITH DATA

Scaling with data refers to the ability of an RDBMS to accomodate increasing amounts of data without adverse degeneration in performance. While a gradual change in performance is always expected as the amount data increases, some database systems are designed to be used in scenarios of small-to-medium datasets. Most websites, for instance, will never have to deal with millions of subscribers, and so the database systems designed for the average website are not suitable for use in national-scale projects, and *vice versa*.

SCALING WITH RESOURCES

This refers to the ability of the RDMS to accomodate more resources (what is referred to as "vertical scaling" and "horizontal scaling") without affecting the functioning of the database adversely. In most use-cases, the database is a single system on a single machine, and this is a case that ought to be highly-optimised because it is very common. However, this is always a trade-off, since a system optimised for this very common case can generally not be extended to scale well with increased resources.

In the final analysis, the *RapidSMS 1000 Days* project, having outgrown the initial assumptions and design, and evidently become more of a national-scale project, is better-served by a mature RDBMS designed for large-scale deployment and consequent scaling. This is a demonstrative list of the database systems that are designed for this type of project:

1. Microsoft SQL Server
2. Oracle Database System
3. PostgreSQL Database System

Given other considerations, such as availability of adapters, programmability, and usability, the open-source PostgreSQL Database System is found to be the best fit for the project.

PostgreSQL *versus* MySQL

Given that the RapidSMS project has historically used the MySQL database, a brief comparison of the two is warranted. Both support the same standard SQL syntax, and are semantically-equivalent. Nevertheless, there are large differences both by design and by circumstance.

PostgreSQL

Starting development in the 1990s, and derived from an RDBMS whose history goes back to the 1980s, the PostgreSQL system has been refined and improved steadily by a large and dedicated community of open source developers, with the support of both large and small companies.

PostgreSQL is also the best-documented database system. For this reason, it is deployed in such sectors as the telecom industry, where its abilities are tested, developed, and widely appreciated.

MySQL

MySQL, on the other hand, has seen about half as much time of development, and far less involvement from varied situations, having been always a simple database for the simple website. While it is tempting to thing of RapidSMS as a web application, because it exploits web technology, data collection systems like RapidSMS have very different concerns (as we will discuss shortly).

Honestly, MySQL has one main benefit: programmers are commonly well-practiced with it, because it can work with extremely small resources, such as those found on standard laptops and desktop PCs. It is mainly for this reason—and its integration with the popular programming language PHP, on which most programmers cut their unfortunate little teeth—that it is considered a good database. But in comparison with PostgreSQL—and particularly given our requirements—it has no saving graces.

On the programming level, there is no significant difference in code written for PostgreSQL and code written for MySQL. All web development frameworks, and in particular the Django framework that we use in RapidSMS, provide an abstraction layer that hide the details of the database, such that to switch from one to another is a matter of changing one line in a configuration file.

At present, we use MySQL simply due to circumstance. In other words: it is what we found, so we use it. There is no particular feature of MySQL that we desire in the project, and certainly none that cannot be got from another good relational database.

On the other hand, there is a particular feature of the PostgreSQL Database System that is required in the RapidSMS project, one that MySQL doesn't (yet) have.

PostgreSQL handles symbolic data in a very efficient way, both for storage and manipulation. by "symbolic data", we mean (for instance) the short strings that are used as "codes" in the RapidSMS application. PostgreSQL has a collection of very complex but very efficient algorithms for processing such data.

The Message & Report Models

Purpose

This section introduces the two core objects in the *RapidSMS 1000 Days* code, with the goal of familiarising the reader with them and their implications for the rest of the code and data.

They affect the rest of the program on all levels. Since the main item of the project is the report, which is delivered as a message, these two objects need to be described and understood.

Prior Situation

In the previous RapidSMS installations, the code-base relied heavily on a Report object which was a composite of report codes which were kept in their own separate database table.
The consequence was that a request for a single report generated at least two different queries, one of them on a table that grew in size exponentially. For every report, there are several codes. But if, for example, a report has 10 codes, a query for 10 reports would result in 100 requests. This doesn't really scale, especially in the deployment scenarios of the *RapidSMS 1000 Days* project.

There is also an organisational problem with having report objects whose core data is stored in disparate locations, even if in the same database. In the standard Object Relational Mapper used in the project considers these two—the reports and their fields—As distinct and separate items which do not have to be kept in synchrony.

It is this design decision in particular that resulted in a lot of the scaling problems that were encountered in the previous deployments of RapidSMS, which in large part have necessitated this phase of the redesign.

The Goals of the New Database Design

Semantic Backward-Compatibility

The most-fundamental feature of the new database design is that it doesn't break semantic compatibility with the previous database design. In all instances, there is a strict equivalence of capabilities.

Specialised Alternative Object-Relational Mapper

The previous database design was dependent entirely on the Django Object Relational Mapper. This resulted in a very simple database structure for representing the reports (specifically, the isolation of a report's attributes to a table of their own) that implemented the well-known database normal forms (1NF, 2NF, *etc.*). However, when one has considerations other than pure relational calculus, such strictness leads to a loss of efficiency.

Extensibile Core

Due to our experience in having to extend the functionality of the previous system, we are convinced that the Object Relational Mapper should be be specialised to some degree for the purposes of storing reports for efficient analysis later on. Furthermore, it should expose extensive database-related metadata about the objects and the database connection to the programmer, to permit further extensions of the ORM in a direction that is conducive for any specific project, without having to diverge from the core code-base of RapidSMS.

Flat Report Tables

A crucial feature of the new database design is that it would take O(n) time complexity to process the commonest query executed against a set of reports—since all the crucial data is now in the same place, as it is supposed to be—which was not the case with the previous system.

The Goals of the New Object Design

The core objects of the RapidSMS 1000 Project have also been redesigned to improve extensibility and code-reusability.

Tightly-Coupled Report and Message objects

The relationship between the Report and the Message is also better-expressed by the explicit use of the factory model, generating Reports conditionally from Messages, and creating Messages unconditionally from the SMS delivered.

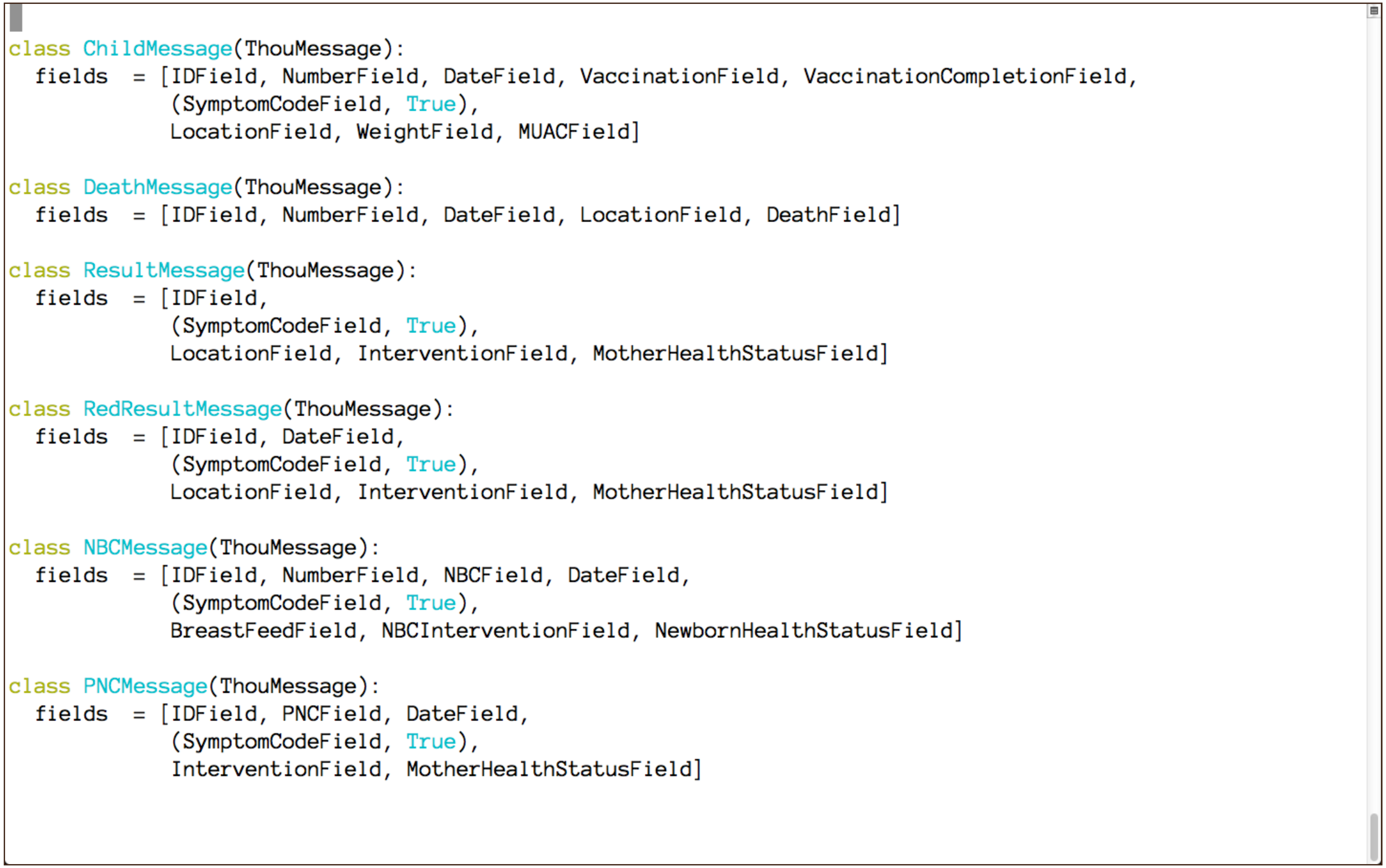
Tightly-Coupled Report and Message objects

The Message and the Report replace the App as the core application object. In the previous App model, every keyword is considered a separate application, which was always responsible for dealing with the text in a programmatic way.

In the new model, a lot of the basic assumptions of a reporting SMS are already codified in the base classes. This means that the consistencies that all “apps” share are already described programmatically in one place, and enables the structure of the expected Messages to be described declaratively, and not programmatically.

Message Description

This leads to a rough equivalence between the descriprion of the Message object in the code and the description in the documentation of the message it handles.



```
class ChildMessage(ThouMessage):
  fields  = [IDField, NumberField, DateField, VaccinationField, VaccinationCompletionField,
             (SymptomCodeField, True),
             LocationField, WeightField, MUACField]

class DeathMessage(ThouMessage):
  fields  = [IDField, NumberField, DateField, LocationField, DeathField]

class ResultMessage(ThouMessage):
  fields  = [IDField,
             (SymptomCodeField, True),
             LocationField, InterventionField, MotherHealthStatusField]

class RedResultMessage(ThouMessage):
  fields  = [IDField, DateField,
             (SymptomCodeField, True),
             LocationField, InterventionField, MotherHealthStatusField]

class NBCMessage(ThouMessage):
  fields  = [IDField, NumberField, NBCField, DateField,
             (SymptomCodeField, True),
             BreastFeedField, NBCInterventionField, NewbornHealthStatusField]

class PNCMessage(ThouMessage):
  fields  = [IDField, PNCField, DateField,
             (SymptomCodeField, True),
             InterventionField, MotherHealthStatusField]
```

Granularised Message Validation

The base classes are also mostly abstract, describing generic predicates for validation and handling, so that complicated validations can be programmed into the system without having to extend the fundamental objects of the code-base.
Checks can vary from simple bounds-checking to querying external databases.

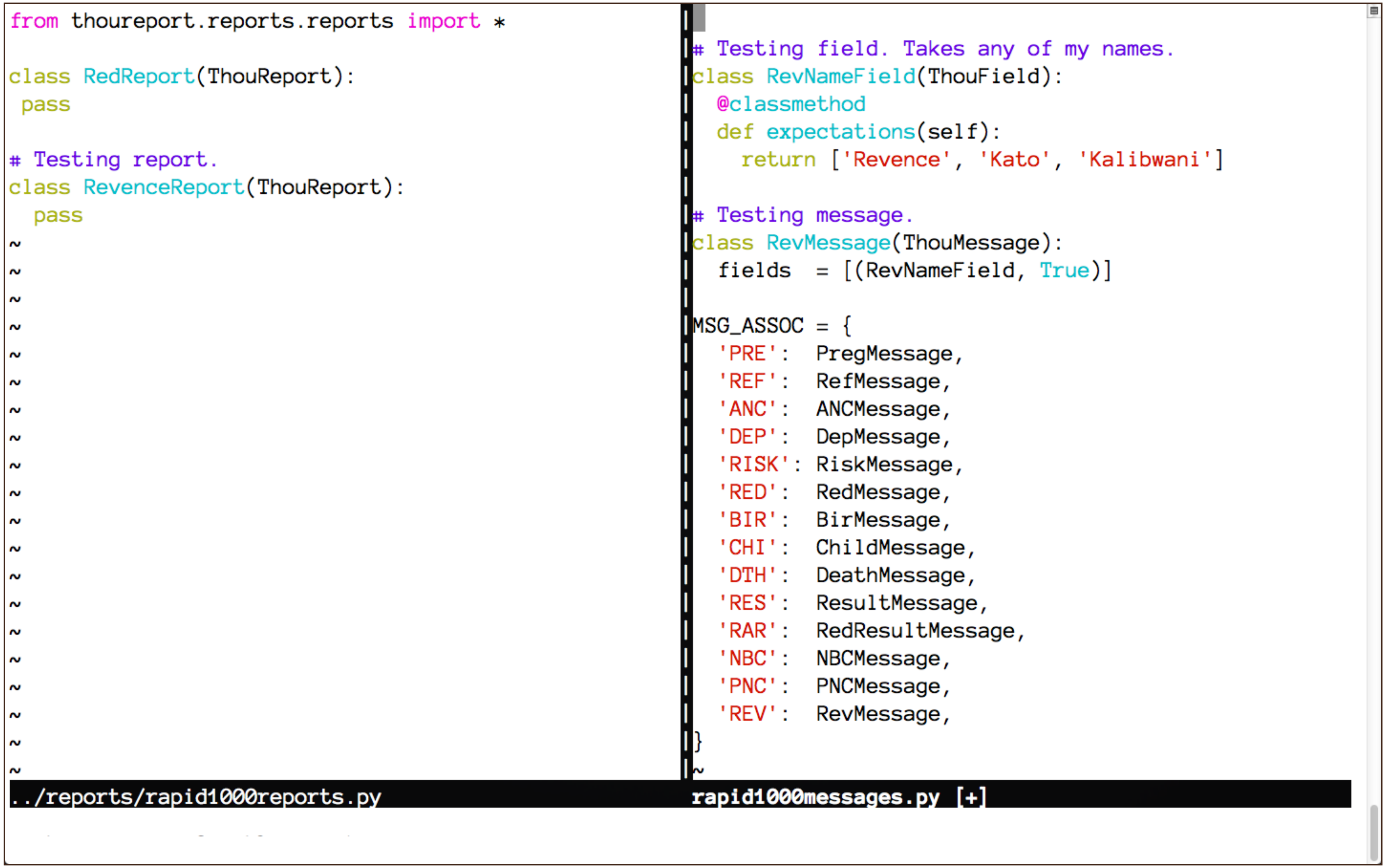
These checks are placed on separate levels of validation, such that a keyword can be described at different levels of granularity. The keyword that only needs to implement simple checks on the data need not be described in many lines of code; yet if it is necessary to implement elaborate logic, it is still possible to program the low-level predicates.

One App, One Framework, Two ORMs

The new Object Relational Mapper is strongly influenced by our particular situation and experience in the past. This ORM is therefore not suited for the more-normal cases, for which the Django ORM being used in the previous installations was suited.
Therefore we have found it reasonable and sound to leave the Django ORM accessible to the rest of the application, where it may be used for the rest of the non-crucial objects of the *RapidSMS 1000 Days* project.

Sample Application

The code comes with a pre-created sample application which describes a report type that is widely divergent from any that we have to deal with in the *RapidSMS 1000 Days* project. This sample application proves the extensibility and wide applicability of the new base system, since in fact it was developed with the *RapidSMS 1000 Days* project use-cases in mind.



```
from thoureport.reports.reports import *

class RedReport(ThouReport):
 pass

# Testing report.
class RevenceReport(ThouReport):
  pass
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
../reports/rapid1000reports.py
```

```

# Testing field. Takes any of my names.
class RevNameField(ThouField):
  @classmethod
  def expectations(self):
    return ['Revence', 'Kato', 'Kalibwani']

# Testing message.
class RevMessage(ThouMessage):
  fields  = [(RevNameField, True)]

MSG_ASSOC = {
  'PRE':  PregMessage,
  'REF':  RefMessage,
  'ANC':  ANCMessage,
  'DEP':  DepMessage,
  'RISK': RiskMessage,
  'RED':  RedMessage,
  'BIR':  BirMessage,
  'CHI':  ChildMessage,
  'DTH':  DeathMessage,
  'RES':  ResultMessage,
  'RAR':  RedResultMessage,
  'NBC':  NBCMessage,
  'PNC':  PNCMessage,
  'REV':  RevMessage,
}
~
rapid1000messages.py [+]
```

Version: 1.0