A Problem Frame-Based Approach to Evolvability: The Case of the Multi-translation

Gianna Reggio, Egidio Astesiano, Filippo Ricca, Maurizio Leotta.

Abstract:

The problem frame approach allows to precisely pin the software development problems before starting to work on them, thus avoiding to solve the wrong problems. Furthermore, the problem frames allow to develop tailored methods and schematic solutions to handle the tasks required to solve the corresponding problems. In this paper we adopt this approach to study the problem of developing a large class of software systems able to translate in different ways some inputs in outputs (e.g., hybrid mail or big brothers filtering digital communications for suspicious words). Our interest in this kind of systems has been prompted by a cooperation with a big company producing systems of this kind and by their search of techniques and approaches to handle predictable and unpredictable changes. We want to investigate how and if the problem frame based approach will help to master the aspects relative to predictable and unpredictable changes in the context, in the domain and in the requirements. We thus present the Multi-Translation Frame.

Digital Object Identifier (DOI):

http://dx.doi.org/10.1007/978-3-642-21292-5_9

A Problem Frame-based Approach to Evolvability: the Case of the Multi-translation

Gianna Reggio, Egidio Astesiano, Filippo Ricca, and Maurizio Leotta

DISI, Università di Genova, Italy
gianna.reggio|astes|filippo.ricca|maurizio.leotta@disi.unige.it

Abstract. The problem frame approach allows to precisely pin the software development problems before starting to work on them, thus avoiding to solve the wrong problems. Furthermore, the problem frames allow to develop tailored methods and schematic solutions to handle the tasks required to solve the corresponding problems. In this paper we adopt this approach to study the problem of developing a large class of software systems able to translate in different ways some inputs in outputs (e.g., hybrid mail or big brothers filtering digital communications for suspicious words). Our interest in this kind of systems has been prompted by a cooperation with a big company producing systems of this kind and by their search of techniques and approaches to handle predictable and unpredictable changes. We want to investigate how and if the problem frame based approach will help to master the aspects relative to predictable and unpredictable changes in the context, in the domain and in the requirements. We thus present the Multi-Translation Frame.

1 Introduction

Evolvability, i.e., the ability to evolve software over time to meet the changing needs of its stakeholders, is one of the principal challenges currently facing software engineering[1]. Software architecture decay over the years, aged programming languages, software written by other developers, all contribute to create what is typically an evolution nightmare. In the literature, this challenge has been faced in several ways. Among the several proposals, the more accredited is building from scratch the system using specific and rigorous techniques/methodologies for evolvability [4, 6, 17]. While we share with these authors the same forward engineering vision, we propose *a frame-driven development approach* [15] to cope with evolvability problems.

Our interest in evolvability has been prompted by a cooperation with a local big company producing various kinds of complex systems. One of them, is the hybrid mail system XYZ[2]. XYZ is used by the postal organizations that offer

[1] Third International IEEE Workshop on Software Evolvability at IEEE International Conference on Software Maintenance (ICSM) Paris, France 1 October 2007 `http://homepages.feis.herts.ac.uk/~comqcln/EN/software_evolvability07.html`

[2] For privacy reason, we cannot report here the name of the system.

their customers — mainly big companies such as banks — specific services to produce big amounts of electronic/physical mail (e.g., invoices and bank statements) starting from electronic data files. Therefore, the main activities of a hybrid mail system are: receiving customer data in several formats (e.g., XML and PDF), processing them to produce the required mails and printing them (close to their final destinations). Afterwards, the produced physical mails are supplied to a logistics service for the delivery.

The main problem of this company is handling in reasonable time predictable and unpredictable changes in XYZ. Predictable changes can be forecasted by looking at the current domains and at the requirements of the system, instead unpredictable changes cannot be imagined (e.g., a new law changing the business rules of XYZ is promulgated).

In this paper we propose the motto "developing for change" to characterize a software development method able to cope with the changes that may be required in the future by the stakeholders. In some sense we try to enlarge the scope of the old motto "design for change" [20, 19] trying to cover also the other phases and activities of the software development. Moreover, we investigate how and if *a frame-driven approach*, a cornerstone of our principle, will help to master the aspects relative to predictable and unpredictable changes. Following the indications of Jackson, who claims that *a clear understanding of requirements (the problem) is crucial to building useful systems, and that to evolve successfully, their design must reflect problem structure*, we have created a specific problem frame called Multi-Translation Frame (shortly MTF). The MTF offers the first help for structuring, abstracting and documenting and thus follows, during the development of a system, the "developing for changes" principle. Moreover, to better explain our proposal, we have used, as running example, a simple toy case called Toy-HMS. Toy-HMS is a toy example of a hybrid mail system and, at the same time, an instantiation of the MTF.

The paper is organized as follows. Sect. 2 describes both the MTF and its instantiation Toy-HMS, also showing the use of the frame to qualify the types of the possible changes. Sect. 3 illustrates our approach "developing for change", based on the MTF, by sketching the Domain Modeling, the Requirement Specification and the Design development phase for the Toy-HMS and then showing how to cope with possible changes. Finally, Sect. 4 presents some related works and concludes the paper.

2 Proposal of a specific problem frame

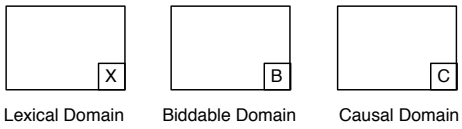
M. Jackson "Problem Frames" [15] are a good tool to tackle with a first structuring of software development problems. For each problem frame, a diagram is settled, showing the involved domains, the requirements, the design, and their interfaces. Five basic problem frames, plus some variants, have been originally provided by M. Jackson in [15]. Other problem frames have been recently proposed together with some extensions to the original notation for the frame [10, 11]. Problem frames are also presented with the idea that, once the appropriate

problem frame is identified, then the associated development method should be given "for free", as shown in [10, 11] where development methods based on the UML are provided for the various frames.

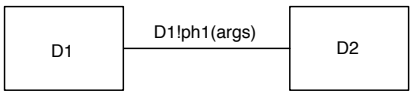
In problem frames presented by M. Jackson, there is a distinction between existing domains and the system to be built as a new part in that world. This implies that the various entities considered in the existing domains are not modified (or removed) when the new system is introduced.

Following Jackson's notation, different types of domains are used to represent the frames. As shown below, a machine domain (the software to be built) is denoted by a box with a double stripe, a designed domain (data structures or subsystems that may be freely designed and specified) by a box with a single stripe, and a given domain (a problem domain whose properties are given) by a box with no stripe. To help to grasp this classification consider these examples: the commands sent to the system controlling a dam can be changed up to some extent, e.g., by using different ways to name them or add shortcuts for special cases or combinations of commands and they were designed by someone, this is a case of designed domain. The dam is a given domain since in the dam control frame it is assumed that the dam is not going to be modified.

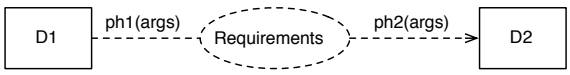
Letters in the lower right corner reflect a coarse classification of domains, orthogonal with the previous one: - *lexical* (data), *biddable* (people or systems, with no predictable internal causality), or *causal* (predictable causality, controlling and controlled by some phenomena).



A solid line connecting two domains is an interface of shared phenomena. Below, phenomena ph1 is controlled by domain D1 and is shared between domains D1 and D2.

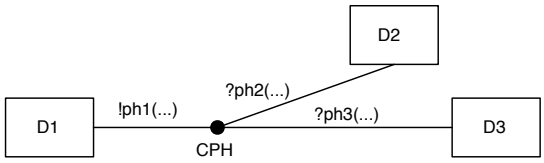


Requirements are denoted by a dashed oval. Dashed lines connect a domain and a requirement (constraining if with an arrow).



A frame diagram is a graph the nodes of which are domains, and its arcs are interfaces of shared phenomena. A *problem frame* is a particular frame including at least a machine and a requirement.

It is possible to connect several domains with an hyper-arc to denote a complex interaction built out of various basic phenomena, that we name composite phenomena [11]; in the picture below CPH is a composite phenomenon corresponding to a complex interaction among three domains, and involving the shared phenomena ph1, ph2 and ph3 (where D1 is responsible to initiate the interaction by means of the shared phenomena ph1).



2.1 MTF: A problem frame for multi-translation

For the multi-translation we propose a specific problem frame called MTF. The MTF offers the first help for structuring, abstracting and documenting and thus follows the principle of "developing for changes". Precisely, it provides: the separation between the translation rules and the dispatch rules, their explicit description and the possibility to find the commonalities among the producers/consumers/input/output data.

Fig. 1 presents the MTF using the M. Jackson notation [15] together with the extensions of one of the authors for the composite phenomena [11].

The input and the output data are given lexical domains, whereas the producers and the consumers are given biddable domains, but here for simplicity we omit to mark them with the letters in the lower right corners.

In the MTF there are various kinds of input data (domains $\mathsf{ID}_1, \ldots, \mathsf{ID}_h$) generated by producers of various kinds (domains $\mathsf{Prod}_1, \ldots, \mathsf{Prod}_s$) and of output data (domains $\mathsf{OD}_1, \ldots, \mathsf{OD}_k$) that will be sent to consumers of different kinds (domains $\mathsf{Cons}_1, \ldots, \mathsf{Cons}_r$)[3]. For simplicity we represent only a generic representative of these series of domains in Fig. 1.

The Multi-Translator is the machine (i.e., the system to be built, denoted by a box with a double stripe on the left); it receives some input data from

[3] In the most general form of MTF, some input and some output data are given but the remaining ones are designed; for simplicity here we assume that all of them are given, but this distinction may be relevant for the coping with the changes, in such cases a richer version of the MTF should be developed.

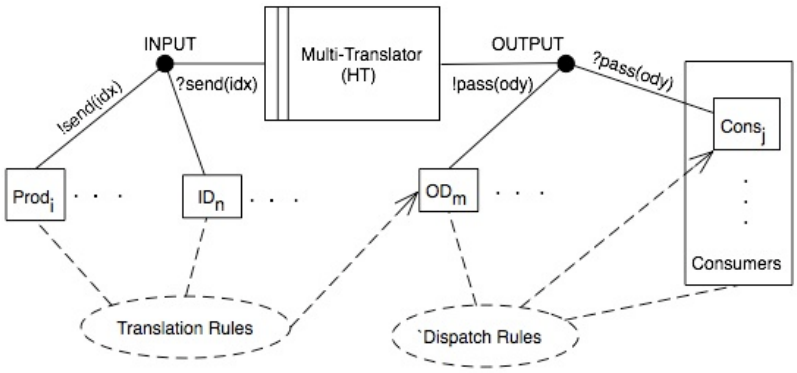


Fig. 1. Multi-Translation Frame (MTF)

some producers (complex interaction INPUT, whereas send(idx) is a shared phenomenon having an argument typed by ID_n and Prod_i has the responsibility to initiate the sharing) and gives out some output data to some consumers (complex interaction OUTPUT). In this frame the requirements are split in two parts: the TranslationRules that define the relationships between input data with their producers and the resulting sets of output data, whereas the DispatchRules determine to which consumer send the output data. In the picture a dashed line means that a domain is referred in the requirement (thus the TranslationRules refers also to the producers and to the input data, and since the current situation of existing producers may affect the choice of the consumers to whom send an output data, the DispatchRules refer to the Consumers), whereas a dashed line with arrow head means that the requirement may affect a domain (thus an output data may be affected by the TranslationRules and a consumers by the DispatchRules).

We assume that the number and the kinds of the producers and of the consumers may vary dynamically while the system is running (for example some new one may appear and some may disappear, but their types are already known); thus the context of the system may change and so the machine must be a context aware system, able to cope with the changes in the context.

A Hybrid Mail System (HMS) can be viewed as an instantiation of the MTF. As we have already said, HMSs are postal systems used to compose and print mails close to their final destinations[4]. These systems are called Hybrid Mail systems because transform mails/data given in "electronic" form to "physical/papery" mails. As particular case of MTF an HMS: (i) requires data (ID) to create mails, (ii) produces the mails ready to print (OD), (iii) executes some procedures to compose/create mails (Translation Rules), (iv) executes some procedures to dispatch mails to the correct printing center (Dispatch Rules). The

[4] Here we do not consider the hardware of these systems (e.g., machines used to print mails).

input data are provided by the clients of the HMS (Producers) and mails are sent to different printing centers (Consumers).

2.2 Toy-HMS: A toy-case instantiation of MTF

Toy-HMS is a toy case of hybrid mail system and an instantiation of the MTF. Toy-HMS is a HMS used by only one company, i.e., there is a unique producer, that sends periodically mails to its customers. Toy-HMS has three *printing centers*: North-PC, Center-PC and South-PC (located in Milan, Rome and Naples), for the Northern, Central and Southern Italy respectively. Each printing center is used to print the mails sent to addresses of its geographic competence. All printing centers can print B/W, whereas North-PC and South-PC can print also in colour.

When a *clerk* of the company has to send mails to customers, (s)he submits the data in electronic mode to the Toy-HMS. The set of all required data is called *batch*. From a batch, Toy-HMS produces a set of mails ready to print and organizes the mails in groups (called *print batch*) to be sent in electronic mode to the three printing centers following some routing rules. All the mails of a batch follow a schema called *template* that defines the common structure of the mails.

Toy-HMS is suitable for handling advertising letters, notices etc, i.e., mails which differ from each other only for the address and the name of the customer.

In Fig. 2 we present the instantiation of the MTF for the case of the Toy-HMS.

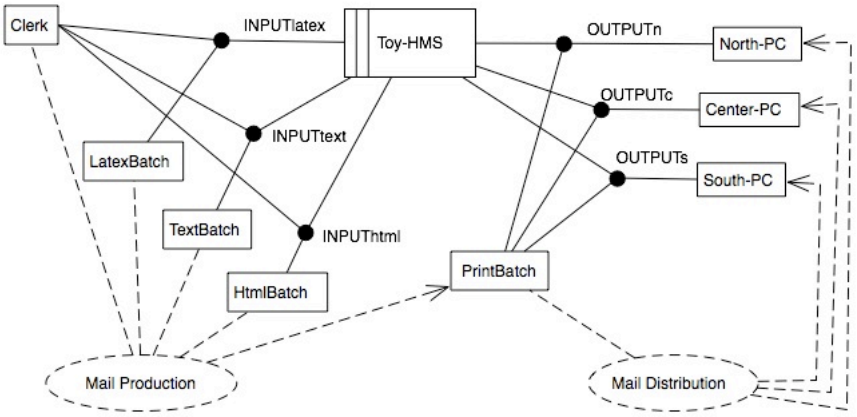


Fig. 2. Multi-Translation Frame (MTF) instantiated for the Toy-HMS case

In this case we have a unique producer (the Clerk) and three consumers (North-PC, Center-PC and South-PC), three types of input data (LatexBatch[5], TextBatch and HtmlBatch) and a unique type of output data (PrintBatch).

[5] `http://www.latex-project.org/`

Any batch is composed of the print type required (B/W or colour), the list of the addresses of the receivers, and the template. A template represents the model for each mail of the batch, and consists of a file in the proper format (either Latex or HTML or Text), where there are some place-holders corresponding to the parts of a correct address.

A print batch consists of a PDF file containing the generated mails, the indication of whether it requires colour printing, and a ZIP Code (all the mails in a print batch are sent to addresses with that code).

The requirements of the Toy-HMS are given by the MailProduction and the MailDistribution.

The MailProduction requires to control and correct up to some extent the address list, to divide the address list in sub address lists one for each ZIP Code, then to compose the mails using the template and the addresses in the various sub-lists (the templates are assumed to be always well formed), and finally to pack them in various print batches consisting of mails sent to the same ZIP Code.

The MailDistribution requires to send the B/W print batches to the nearest printing center since all the three of them can print B/W, and to send the colour print batches to the nearest printing center chosen between North-PC and South-PC (those that can print in colour).

To determine which is the closest printing center a function distance between the print batch ZIP Code and the printing center ZIP Code is used. This function is defined taking into account a large set of factors, including the availability of routes, railways, motorways and these factors have been already considered when the ZIP Codes were defined in Italy.

To validate Toy-HMS, we have implemented a SOA-based Java prototype. Moreover, we have compared the architecture of the prototype with the actual architecture of the hybrid mail system XYZ [18].

2.3 Relating the variety of changes to MTF

The MTF also offers the possibility to describe and classify the possible changes of the Multi-Translator system that its developer may have to cope with. We classify the changes in three broad categories: changes in context, proactive standard changes and unpredictable changes.

Changes in context: The number and the features of the producers and of the consumers connected to the system may change while the system is running (new producers/consumers may be added, however their type belongs to an existing list; existing producers/consumers may be eliminated; some producer/consumer may change its features, but the modification is relative to an existing list of features, whose possible forms are already known). These are the possible changes that affect the context of the system Multi-Translator (context intended as the entities interacting with the system itself).

Example of changes of this kind in the case of Toy-HMS are: a printing center breaks down and it is not available till it will be repaired, a new B/W printing

center with the same characteristics of the existing ones is built in another Italian city (for example Palermo or Venice), and the printing center in Rome becomes able to print also in colour.

Proactive standard changes: These are changes that can be forecasted by looking at the domains and at the requirements of the MTF. We name these changes "proactive" since the developers should be proactive and propose them to the client as a way to increase their business, and "standard" since they are specific of the MTF. Here we illustrate two typical sample categories.

- New kinds of "derived" functionalities are required for the Multi-Translator. A derived functionality is a new one that just requires to collect or to elaborate information already computed for the existing functionalities, or just to log the performing of some activities. These changes modify the requirements enriching them in a conservative way. Some examples of changes of this kind in the case of Toy-HMS are the request to: – produce a report associated with each batch listing information on the sent paper mails, – collect data on the number and types of the processed batch for the accounting department, – record the number of wrong address files.
- The Multi-Translator system should perform simplified variants of its functionalities; for example a client provides pre-elaborated input data that do not require to perform the initial elaborations, or a producer requires to receive some intermediate data skipping some of the final elaborations. This kind of changes should lead to a nice simplification of the requirements and the update of the system should be very easy, but in practice they may result in hard work in case of wrong architectural choices.
  Example of changes of this kind in the case of Toy-HMS are the request to: – skip the control and the correction of the address files since the addresses are surely correct, – send to the printing center a print batch in the form of a Latex or HTML source file or as a text file, instead of a PDF file (the PDF file generation will be performed by the printing center), – receive as input data some print batches so that only the dispatching activities will be performed.

Unpredictable changes: The unpredictable changes are those that cannot be imagined by examining the current domains and requirements of the frame. For example, completely new kind of producers or consumers or completely new kinds of input and output data may appear, or new laws or rules that disrupt the current requirements may become in effect.

Example of changes of this kind in the case of Toy-HMS are: – the appearing on the market of new printers (e.g., printers using paper of different weighs); – the emergence in the market of the private email services that completely change the rules for determining the most convenient printing center (e.g., someone following a policy where everything goes first to Rome); – the appearing of a new document format (e.g., docx).

Obviously, in this case a socio-economic analysis may help to get a rough idea on which changes of this kind are more likely in the future (e.g., in the case

of Echelon the char set used for the English language is not going to change very soon, whereas a change of the kinds of communication devices to monitor is highly probable).

3 Developing for Change

In this section we outline the approach "developing for change" that we have mentioned in the introduction; an approach characterizing a software development method able to cope with the changes that may be required in the future by the stakeholders. With respect to the old motto "design for change" [20, 19], we try to cover also the other phases and activities of the software development. Obviously this is the exact contrary of other current approaches (e.g., extreme programming [5]), where not even the design for changes is recommended, and obviously a cultural/technological/business analysis has to be performed initially to see whether the system to be developed will be a long lived one and whether it will ever have to cope with changes (e.g., chess games have not changed too much in the last centuries, whereas taxing mechanisms change continuously).

In this section we show how we can use the MTF to develop its instantiations in a way that makes easier and more efficient tackling the need for changes that may arise during and after the development. The general guidelines that we propose are cast within a development method based on the rigorous well-founded usage of the UML as notation for expressing all the required artifacts, see, e.g., [1–3].

3.1 Key Principles in Developing for Change

To make our treatment more understandable and convincing, we recall in this subsection some key principles for good design, that are preliminary to cope with changes. We will refer frequently to those principles in the following sections and thus we label them for an easy reading without repetitions. Most of them are classical, but in our approach they will be used not only at design level, but everywhere.

KeyP1 Structure and document everything, not just the code. It is not sufficient to document the code or the detailed architecture of some modules, but all the aspects of the system should be documented, e.g., also the producers and the consumers, and the rationale behind the dispatch rules should be properly documented; and obviously everything should be structured to allow the human reader to grasp it.

KeyP2 Provide high-level presentation/documentation. It not sufficient to attach a lot of comments to the code or to some architectural very detailed schema or to complex XML definitions of data structures; rather, all the key concepts of the system should be presented in a brief and compact way avoiding too many technical details. For example, a large part of the documentation should be understandable by the business expert (e.g., the core

of the dispatch rules should not be hidden in a small stored procedure in a database written in a proprietary language, even if well commented; instead it should be presented by a set of conditional rules written using well structured natural language referring to business entities).

KeyP3 Explicit (abstract) typing, i.e., give a precise abstract type to each entity used in the system. Consider the following example, a fundamental data structure, as an input data, should be explicitly defined by giving the operations for acting on it, and then by saying how it will be realized using the chosen technical means. For example it cannot happen that a fundamental data structure is implicitly realized by some XML files and some records in two different databases without no way to see which modules are using it and how; it becomes abstract typing when put together with the next principle.

KeyP4 Encapsulation, i.e., give a precise type/interface to each part of the system and of the software, as data structures, modules, components, functionalities and external subsystems (e.g., the support offered by an external subsystem should be presented by means of a function/procedure with a precise type and well defined meaning; any external device should have a high-level interface to be accessed, it is not sufficient to say that to pass an input to some external systems one has to put a file starting with some lines written using some cryptic codes in some specific point of the file system).

KeyP5 Separate commonalities from particular aspects in any part of the system not only in the data types, and this requires also to organize the data types and the other parts in explicit specialization/refinement structures (e.g., the various kinds of output devices may be very conveniently presented as a specialization hierarchy having an abstract device at the top; a group of requirements may be nicely organized in a hierarchy making explicit the common parts and the variants).

KeyP6 Use generative techniques, namely try to use high-level presentation of data structures and function that can be automatically transformed into the corresponding code (e.g., provide conditional rules that may be automatically transformed into code, instead of writing directly such code; provide a BPMN diagram [9] to be transformed into BPEL codes instead of writing directly some Java code to orchestrate some Web services).

3.2 Guidelines for a MTF-Based development

We assume to have instantiated the MTF having determined who are the producers, the consumers, the input and the output data and which are the translation and the dispatch rules.

We note from the beginning that the use of the frame leads to apply *KeyP1*: indeed it obliges to provide a first documentation clarifying the four parts of the frame and a first initial structuring (e.g., the organization in terms of the domains parts of the frames and the separation of the translation requirements from those concerning the dispatching).

Domain Modelling The domains in the MTF are the input/output data and the producers/consumers.

It will be highly probable that the various input and output data, the producers and the consumers share some commonalities; thus, following *KeyP5*, we look for specialization relationships among domains, possibly by introducing "abstract domains", where domain has to be intended in the sense of Jackson [15], and to try to encapsulate them (*KeyP4*).

In our approach (see, e.g., [1–3]) the domains are modelled by means of UML classes. Thus the domain model in the case of the MTF is a UML class diagram with a class for each kind of producer, consumer, input and output data present in the frame. The classes corresponding to the input/output data, which are lexical domains using Jackson's terminology, are static whereas those corresponding to the producers/consumers (biddable domains) are active classes with an associated behaviour, which may be modelled by a state machine. The producers/consumers must have an operation corresponding to the send/pass a data (shared phenomena with the Multi-Translator). It is important to fully model the data classes whereas it may happen that the model of the producers/consumers may be quite underspecified.

The fact that the various domains should be modelled by UML classes is in agreement with *KeyP2*, *KeyP3* and *KeyP4*. In Fig. 3 we show the Domain Model for the Toy-HMS case. In this diagram the class Clerk represents the producer type and the class PrintCenter the consumer type. The input data is represented by the class Batch specialized in LatexBatch, TextBatch and HtmlBatch. The output data is represented by the class PrintBatch. The other classes in the diagram (e.g., TextFile) are used for typing the attributes.

Requirement Specification In agreement with *KeyP1*, the requirements in the case of the MTF are already split in two parts: the TranslationRules and the DispatchRules, where the TranslationRules should associate a producer and an input data with a set of output data, and the DispatchRules should associate an output data with a consumer.

Technically the TranslationRules should define a family of partial functions, called translation functions,

$trans_k : ID_i \times Prod_x \longrightarrow Set(OD_j)\ k = 1, \ldots, p$

where $trans_k(id, p) = ods$ means that the translation of id sent by p will result in ods, whereas $trans_k(id, p)$ undefined means that in that case the translation failed[6].

The TranslationRules will be modelled by an abstract UML class, with the same name, having a static operation for each translation function. Obviously such functions should share some common aspects, otherwise the system would be just an aggregation of single-translation subsystems performing) and we should make explicit such commonalities; technically the translation functions should be expressed by composing/combining a set of functions corresponding to basic translation blocks (we call them *translation blocks*).

[6] For the moment we do not consider the error messages.

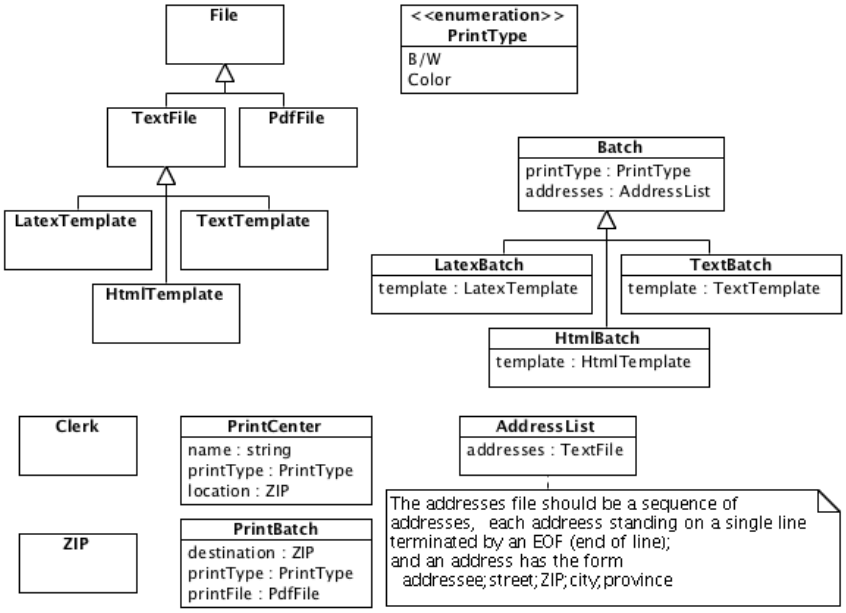


Fig. 3. Toy-HMS case: the domain model

The decomposition of the translation functions in translation blocks is a form of structuring (*KeyP1*, while providing a precise type to the translation functions and the translation blocks is an application of *KeyP3*).

In the UML model the translation blocks will be modelled by other static operations of the same class TranslationRules. The effort of decomposing the translation functions and of extracting the translation blocks and the classes/ types needed to type their parameters and their results will be the application of the principle of decomposing and structuring (*KeyP1*) and of high level documentation (*KeyP2*); indeed all these operations should have meaningful names and should be properly documented.

The operations corresponding to the translation operations and to the translating blocks will be defined using the means offered by the UML, such as OCL constraints, methods, and activity diagrams. For example, an operation may be modelled textually in the UML giving its (purely functional) body, or visually by means of an activity diagram, with matching input and output parameters where the basic activity are some translating blocks.

The DispatchRules should define a family of partial functions

$dispatch_h : OD_h \times Set(Consumer) \longrightarrow Consumer\ h = 1, \ldots, r$

where *Consumer* is an abstract class having as specializations all the consumer classes, and $dispatch_h(od, cons) = con$ means that the dispatch of *od* when all the consumers *cons* are available, will result in sending it to *con* (obviously

$con \in cons$), whereas $dispatch_h(od, cons)$ undefined means that in that case the dispatch failed[7].

The DispatchRules will be modelled by an abstract UML class, with name DispatchRules, having a static operation $dispatch_h$ for each dispatch function. Again these operations should be decomposed using some basic dispatch operations making explicit any existing commonalities among them.

Every operation needed for modelling the TranslationRules and the DispatchRules may be defined at a varying degree of abstraction and thus the requirements may be as much abstract as needed. The rules may be quite complex and may be written in a way that will help cope with the future change requests.

Another application of *KeyP5* is to avoid duplications in the definition of TranslationRules and the DispatchRules, that is to avoid that the same function fragment be written several times (same as the hint for avoiding duplicated code). Using the UML this means to introduce auxiliary operations and classes to represent the duplicate fragment. There is no need to say that the same suggestions apply even more stringently if the translation definitions are utterly complex.

If no commonalities or just a little amount of them are found among the definitions of the various translations and dispatching functions, then we should consider if the problem is about the development of a unique product or just a collection of very loose related different products. For example in the Echelon case the part concerning the search of hot words in text should be common to filtering email, chats, social network walls and SMS, whereas the capability of extracting text from PDF and JPEG files will be again used in various of the filtering functions, which are the translation functions in this case. On the other side a product offering a bunch of file compressing operations based on totally unrelated techniques should not be a case of the MTF.

In Fig. 4 we present the requirements for the Toy-HMS case using the UML. To make the diagram more compact we have omitted the classes already described in the Domain Model in Fig. 3, and the definition of the various functions can be found in Appendix A.

Design The design phase requires to produce the machine, in Jackson's sense, of the problem frame schema, that is the Multi-Translator. The software architecture of the Multi-Translator will be represented using the UML, where each component will be modelled by a class.

It is possible to provide various architectural schema and to compare them w.r.t. the support for changes (for example using SAAM, a strategy presented in [12, 16]). Moreover depending on further information concerning the non-functional aspects, such as the dimension of the input and output data, the frequency of the arrival of data to translate and whether there are real time constraints on when to return the translated data, further architectures may be proposed. However, there are a few general constraints on the possible architec-

[7] Again, for the moment we do not consider the error messages.

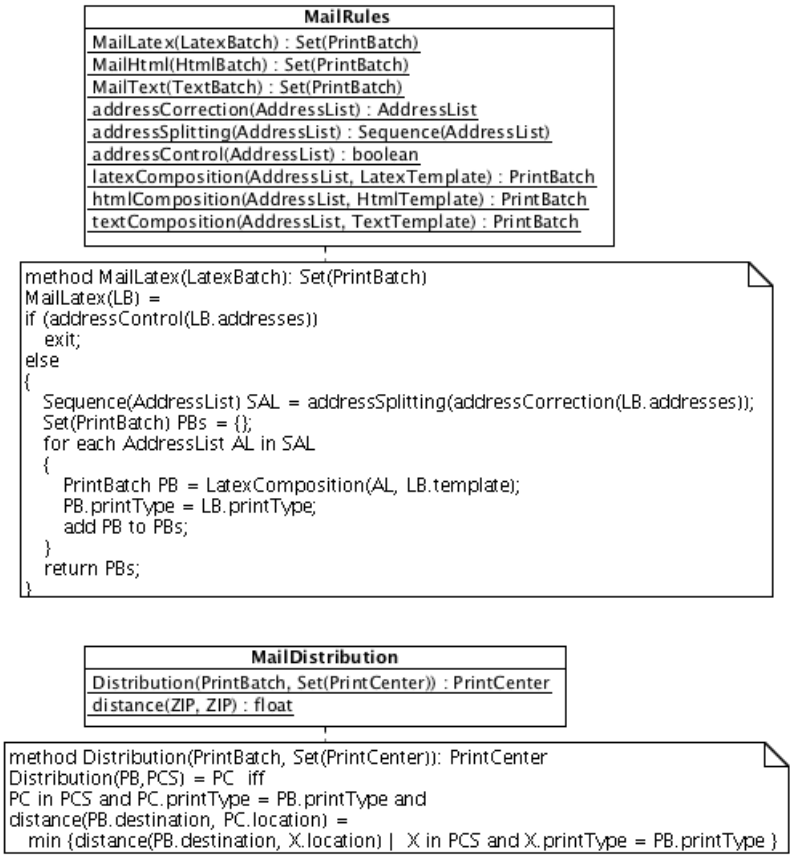


Fig. 4. Toy-HMS case: the requirements (classes of the domain are not repeated)

ture to be able to cope with the changes, that we summarize below and visually present in Fig. 5.

We use the same notation of the Jackson's frame to present the hints on the architecture of the Multi-Translator, and we depict the machine domain icon with dashed lines to suggest that it is not a complete architectural definition. For simplicity in the picture we depict only some representatives of the various domains (input and output data, producers and consumers). The Executor is a machine (denoted using the Jackson's notation by the double stripe on the left), the other domains denoted with two lines on the left are designed domains.

The lexical designed domain Consumers (composed of: name, printType, location) is a representation of the current situation of the consumers (which are the current available ones and their relevant features), which should be recorded in a persistent way inside the Multi-Translator, and there should be the means to keep it updated. Thus we have an Operator, a biddable given domain that will take care of updating the consumers, (s)he will know when new consumers will start/stop to operate or will change their characteristics and (s)he will interact with the Multi-Translator to update this information. Providing the domain Consumers is an application of *KeyP3*: we must give an explicit type structure for representing the consumers. To define this data structure we should look at the requirements to discover what is relevant of the consumers for the dispatching (for Toy-HMS just the type and the location of the printing center).

Then, there will be some parts corresponding to the translating blocks, the translating functions and the dispatching functions appearing in the requirements. They are given domains and may be lexical or causal, indeed they may be sub-machines able to perform such functionalities or descriptions of such functions that another component will be able to execute. Thus we do not fix the kind of domain corresponding to them.

According to *KeyP6* (generative techniques), the translation blocks, the translations functions (and the dispatch rules and their basic sub-functions) are the ingredients that the executor uses to put together the translations to apply to the input data (for example, they may be: – code written in some domain specific languages, – compiled procedure written in some programming language, – software components or services). Thus the Multi-Translator has not some parts that are able to perform the various translations, but it includes an engine (the Executor) able to execute in a very general sense the definition of the various functions, in some sense it will generate the translation functions starting from the various ingredients.

3.3 Coping with changes in the Multi-Translation Frame

Now we consider, as examples, some of the most probable changes in the MTF and see how the proposed method will help to cope with them.

Changes in context The considered changes in the context are the appearing and disappearing of producers/consumers and changes in their features. The

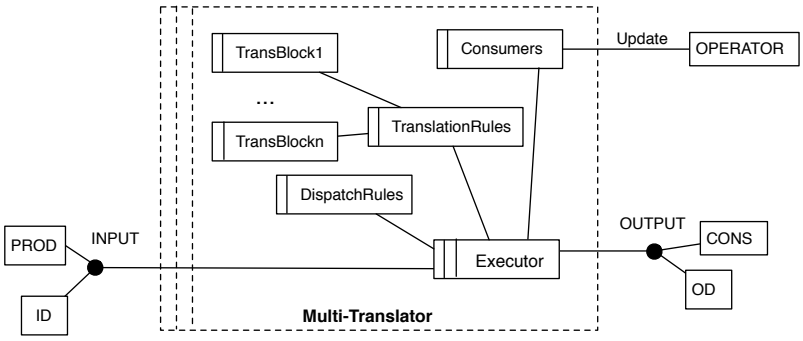


Fig. 5. Multi-Translation Frame: assumption on the design

presence of the Consumers domain, and of the Operator that takes care to update it, allows the Multi-Translator to cope with the modifications in its context. Thus is quite easy to cope with this changes, and the proposed solution is a classical one.

Proactive standard changes *Simplified versions of a translation.* The simplification may require to drop some of the transformations and/or to receive as input data what was before an intermediate data or to dispatch what was before an intermediate data. The transformation just requires to introduce as new input and output data the previous intermediate data; it is needed to check that they are suitable for the associated consumers or that they may be sent by the producer.

Unpredictable changes *Appearing of a new type of input data (without modifications in the output data and in the dispatch rules).* It is highly probable that the new kind of input data will be modelled by a subdomain of the existing ones or that it will be modelled by reusing parts of the existing ones (i.e., already existing UML classes), and so it will be easy to give its model. Then we have to define the corresponding translation functions (one for each possible producer of this kind of input data) reusing the existing translation blocks; it may happen that we have to introduce new translation blocks and/or classes to type their arguments and results. Thus, the modification of the Multi-Translator will just amount to possibly add some new translating blocks and a new values to the translation rules, whereas the Executor is not changed.

Other changes of the same type (e.g., new consumers or new kind of consumers, new dispatch rules) can be handled in the same way.

Disappearing of a type of input data. The input data is eliminated by the domain, and we have to check whether there was some producers producing it and investigating whether they have to be deleted as well. At the requirement level the corresponding translation function will be eliminated, then it has to be checked whether there are some translation blocks used only by itself, in this

case also they are to be eliminated together with the relative types. Finally, we have to check if the eliminated translation function was the unique one to produce some kind of output data, that have in turn to be eliminated, and as last steps we need to investigate if there are some consumers that were receiving only such output data, that have to be deleted in turn. Doing this chain of deletions we have to do several checks helping to detect possible inconsistencies in the proposed change. At the Multi-Translator level, it is sufficient to eliminate the useless translation blocks and translation functions, and this is not a difficult operation, since all of them are items of the proper kind.

4 Related Work and Conclusions

To the best of our knowledge, this is the first work that proposes a *frame-driven development approach* able to cope with evolvability problems in a specific domain.

Among the software development methodologies specific for evolvability we can cite, e.g., [4, 6, 17]. Bastani in [4] presents a new analytical framework, called "Abstraction-oriented Frames" and a method specifically designed to support the requirements analysis and design of open evolvable software systems. To meet flexibility in changing requirements, architecture and design at any phase, the requirements framework presented in that paper consists of dynamics that facilitate subsystem modifications at any level of abstraction or any part of the system. Another framework able to cope with evolvability problems is presented in [17] by HP. That framework, named ORBlite, provides a substrate that allows systems to be composed of components that can evolve independently over time. ORBlite has been successfully used by HP to build several evolvable real systems. Instead, authors in [6] propose and explain "Goal sketching": a simple technique used for requirements engineering purposes. This technique starts with the creation of a goal graph which expresses the high level motivations behind the intention to develop the software. Then, a series of developments are planned using the goal graph as a guide (similarly to use Scrum sprints [21]). Authors claim that Goal sketching can be successfully used to develop evolvable systems. Finally, Schmidt [21] presents arguments in favour of implementing evolvable software systems using the SOA paradigm [13]. Indeed, SOA provides an extraordinary mechanism for supporting the evolution and rapid response to change in business rules.

Even if several proposals have been advanced by researchers in this direction, Brcina et al. [7] claim that existing software methodologies do not provide sufficient support for managing the evolvability. For this reason, they present a meta-model based and goal oriented process for controlling and optimizing the evolvability of a given software system. Breivold [8] in his PhD thesis introduces a method for analyzing software evolvability at the architecture level. More precisely, he identifies some sub-characteristics (e.g., Testability) that are of primary importance for an evolvable software system, and outlines a software evolvability model that provides a basis for analyzing and evaluating the evolvability of

a given system. Instead Shiri et al. [22] present a novel approach to estimate the effort of potential modification and retesting associated with a modification request, without the need of analyzing or understanding the system source code. The approach is based on Use Case Maps and concept analysis [14].

In this paper we have presented a problem frame based approach to study the problem of developing a large class of software systems able to translate in different ways some inputs in some outputs (e.g., hybrid mail or big brothers filtering digital communications for suspicious words). In particular we have shown the use of problem frames in the case of the Multi-Translator with a simple instantiation, the Toy-HMS. Also, we have shown that the problem frame approach helps to produce systems that are able to handle predictable and unpredictable changes according to the new coined motto "developing for changes".

Even if our method may seem too simple for realistic big systems, we believe that is usable and effective. For this reason, we want to apply it in future to obtain a new evolvable version of XYZ (the real huge system of interest of the company that has prompted our investigations). In future works, the capability to cope with predictable and unpredictable changes of the new HMS will be empirically analyzed and compared with the actual one using a "what if approach", i.e., looking at how easy/hard will be to cope with a given list of real change requests following the Software Architecture Analysis Method (SAAM) proposed in [12, 16].

A The Translations and Dispatches Blocks for **Toy-HMS**

Function	Signature	Description and Example
addressControl	AddressList ⟶ boolean	checks whether the addresses are correct with respect of the format ForInd and each address is separated from each other by a end-of-line. ForInd format: addressee;street;ZIPCode;city;province
addressCorrection	AddressList ⟶ AddressList	examine the input address list correcting various mistakes. For example, in the two addresses below there are two mistakes; the first presents an incoherence between the city and the province; the second presents a misspelling in the city name: Mario Rossi;Via Verdi, 23;00100;Roma;MI Mario Bianchi;Via Verdi, 23;00100;Romw;RM The two addresses are corrected in: Mario Rossi;Via Verdi, 23;00100;Roma;RM Mario Bianchi;Via Verdi, 23;00100;Roma;RM
addressSplitting	AddressList ⟶ Sequence(AddressList)	groups the addresses that present the same ZIP Code in separated AddressList, one for each ZIP Code.
latexComposition	AddressList × LatexTemplate ⟶ PrintBatch	creates a PDF file containing the mails created joining the Latex template with the data within the address list (notice that however the print type attribute of the produced print batch is not defined)
htmlComposition	AddressList × HtmlTemplate ⟶ PrintBatch	creates a PDF file containing the mails created joining the HTML template with the data within the address list (notice that however the print type attribute of the produced print batch is not defined)
textComposition	AddressList × TextTemplate ⟶ PrintBatch	creates a PDF file containing the mails created joining the Text template with the data within the address list (notice that however the print type attribute of the produced print batch is not defined)
distribution	PrintBatch × Set(PrintCenter) ⟶ PrintCenter	PrintCsOk = { PC ∈ PrintCs | PC.printType = PB.printType } return PCselect s.t. PCselect ∈ PrintCsOk and distance(PC.location,PB.destination) = min{ distance(X.location,PB.destination) | X ∈ PrintCsOk}
distance	ZIP × ZIP ⟶ Float	return the distance between the two ZIP

References

1. E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proceedings 14th International Conference on Software Engineering and Knowledge Engineering*, pages 143–150, New York, Usa, 2002. ACM Press.
2. E. Astesiano and G. Reggio. Towards a well-founded UML-based development method. In *1st International Conference on Software Engineering and Formal Methods*, 22-27 September 2003.
3. E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future, Proc. 9th Monterey Software Engineering Workshop, Venice, Italy, Sep. 2002.*, number 2941 in LNCS. Springer Verlag, Berlin, 2004.
4. B. Bastani. A requirements analysis framework for open systems requirements engineering. *SIGSOFT Software Engineering Notes*, 32(2):47–55, 2007.
5. K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change, 2nd Edition.* Addison Wesley, Boston, 2004.
6. K. Boness and R. Harrison. Goal sketching: Towards agile requirements engineering. In *Proceedings of International Conference on Software Engineering Advances*, pages 71–77, 2007.
7. R. Brcina, S. Bode, and M. Riebisch. Optimization process for maintaining evolvability during software evolution. *IEEE International Conference on the Engineering of Computer-Based Systems*, 0:196–205, 2009.
8. H. P. Breivold. *Software Architecture evolution and software evolvability, PhD thesis.* Mlardalen University Press Licentiate Theses No. 97, 2009.
9. Business Process Management Initiative (BPMI). Business Process Modeling Notation (BPMN). Available at `www.bpmn.org/Documents/BPMN%20V1-0%20May%203%202004.pdf`, Website, last access 13 December 2007, 2004.
10. C. Choppy and G. Reggio. A UML-Based Approach for Problem Frame Oriented Software Development. *Journal of Information and Software Technology*, 2005.
11. C. Choppy and G. Reggio. Requirements capture and specification for enterprise applications: a UML based attempt. In J. Han and M. Staples, editors, *Proc. ASWEC 2006*, pages 19–28. IEEE Computer Society, 2006.
12. P. Clements, L. Bass, R. Kazman, and G. Abowd. Predicting software quality by architecture-level evaluation. In *Proceedings of the Fifth International Conference on Software Quality*, volume 5, pages 485–497. Software Engineering Institute, 1995.
13. T. Erl. *SOA Principles of Service Design.* The Prentice Hall Service-Oriented Computing Series from Thomas Erl, 2007.
14. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer-Verlag NY, 1997.
15. M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley, 2001.
16. R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, 1996.
17. K.E. Moore and E.R. Kirshenbaum. Building Evolvable Systems: The ORBlite Project. Available at `www.hpl.hp.com/hpjournal/97feb/feb97a9.pdf`, Website, last access 7 November 2010, 1997.

18. M. Leotta, F. Ricca, G. Reggio, and E. Astesiano. Comparing the maintainability of two alternative architectures of a postal system: SOA vs. non-SOA. In *Proceedings of 15th European Conference on Software Maintenance and Reengineering (to appear)*, March 2011.
19. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.
20. D. L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
21. V. A. Schmidt. Using service oriented architecture for evolvable software systems. In *11th International Command and Control Technology Symposium*, 2006.
22. M. Shiri, J. Hassine, and J. Rilling. A requirement level modification analysis support framework. *IEEE International Workshop on Software Evolvability*, 0:67–74, 2007.