Combining Domain Driven Design and Mashups for Service Development

Carlos A. Iglesias, José I. Fernández-Villamor, David del Pozo, Luca Garulli and Boni García

Abstract This chapter presents the Romulus project approach to Service Development using Java web technologies. Romulus aims at improving productivity in service development by providing a tool-supported model to conceive Java Web Applications. This model follows a Domain Driven Design approach, which states that the primary focus of software projects should be the core domain and domain logic. Romulus proposes a tool-supported model, *Roma Metaframework*, that provides an abstraction layer on top on existing web frameworks and automates the application generation from the domain model.This metaframework follows an object centric orientation, and complements Domain Driven Design by identifying the most common cross-cutting concerns (security, service, view, ...) in web applications. The metaframework uses annotations for enriching the domain model with these cross-cutting concerns, so-called aspects. In addition, the chapter presents the usage of mashup technology in the metaframework for service composition, using a web mashup editor *MyCocktail*. This approach is applied to a scenario of the Mobile Phone Service Portability case study for the development of a new service.

Carlos A. Iglesias
Informática Gesfor, Av. Manoteras, 32 - 28040 Madrid (Spain), e-mail: cif@germinus.com

José Ignacio Fernández-Villamor
Universidad Politécnica de Madrid, ETSI Telecomunicación, Ciudad Universitaria s/n - 28050 Madrid (Spain), e-mail: jifv@gsi.dit.upm.es

David del Pozo
Informática Gesfor, Av. Manoteras, 32 - 28040 Madrid (Spain), e-mail: dpozo@grupogesfor.com

Luca Garulli
AssetData S.r.l., Via Rhodesia, 34 - 00144 Rome (Italy), e-mail: luca.garulli@assetdata.it

Boni García
Universidad Politécnica de Madrid, ETSI Telecomunicación, Ciudad Universitaria s/n - 28050 Madrid (Spain), e-mail: bgarcia@dit.upm.es

1 Introduction

Web software development is one of the most active areas and fastest growing industries in software and services development in Europe, and, in particular, Java Enterprise Edition is the mainstream European technology option for one million European developers. Since web development is not still a mature area, the proliferation of frameworks and components has both increased the required skills of web engineers, and has affected considerably their productivity. For that reason, the evolution of existing Java based web applications and services is a very hard and time-consuming task.

Currently, the wide range of technologies and frameworks available for Java Web Development gives it a wide range of attributes when compared to other solutions that have emerged, such as Ruby on Rails. Nevertheless, this is also one of its main shortcomings, as developers spend a substantial amount of time learning new technologies and frameworks and new versions of these frameworks, which in turn decreases productivity. In addition, simple tasks require too much coding. New solutions such as Ruby on Rails have shown that web development can be easier, based on important concepts, such as (1) conventions over configuration, (2) providing a framework that automates up to 80 per cent of the most common tasks and (3) a simple and modular MVC model for developing applications. Romulus aims to learn from the lessons of Ruby on Rails and provide a productive solution based on Java. Romulus is pushing to improve Java web development in several directions such as improving the productivity with the provision of a Domain Driven Design (DDD) Roma Metaframework and IDE integration, and involving soft goals in the development process such as providing web and security testing facilities; and research on how mashup technology can improve web development.

This chapter presents the DDD Roma Metaframework for improving productivity in service development. Roma Metaframework follows an object centric approach for conceiving services and applications. Then, the chapter presents how services and mashups have been modeled in Roma Metaframework, with the purpose of composing or customizing existing services, or developing new ones.

The chapter is structured as follows. First, Sect. 2 collects the Mobile Phone Service Portability (MPSP) case study. In particular we are interesting in the *invitation service*, which is used throughout the chapter to show how Romulus addresses service development. Then, Sect. 3 outlines the main steps to develop a service in Romulus. Sect. 4 overviews the main DDD principles and their application. Next, Sect. 5 describes the design and model of Roma Metaframework, a tool supported environment developed in Romulus. Sect. 6 details how Roma has taken advantage of mashup technology for service composition providing a uniform view for Enterprise and Web mashups. Finally, Sect. 7 and 8 conclude with comparison with existing related work and the main conclusions of the chapter.

2 Case study

One of the most common complaints from business development and marketing units is that engineering units fail in providing a suitable "time-to-market" solution when developing new services. The MPSP includes two scenarios, TELCO_BG_O5 and TELCO_S_03 that deal with this topic: how can we speed up service development?

The scenario described in TELCO_S_03, here reproduced, is detailed in this section.

Case Study: New Service for Promotions

The innovation department of the telecom company decides to launch a new service which offers their customers a promotion to see free pay-per view movies. Customers can send up to 10 invitations to their friends of a social network to watch one free movie per invitation. Customers should select both the friends as the movies, and can add a message to the invitation. Friends can accept the invitation, which includes providing some marketing info. Once the friends accept the invitation, the customer is also allowed to see the movie. The service will provide a set of reports about the success of the promotion based on the collected marketing data.

In order to analyze this scenario in detail, a use case diagram is shown in Fig. 1, which collects the usage scenarios as well as the actors identified, which are specified in Table 1.

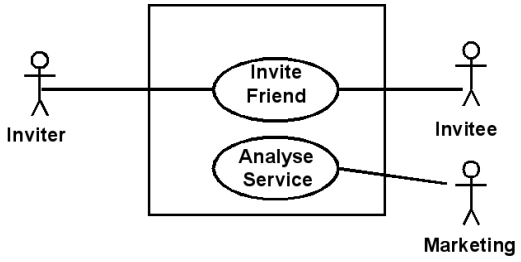

Fig. 1 Use case Diagram of the Invitation Service

For the purpose of this chapter, we are going to develop in detail just one of these use cases, the use case *Invite Friend*, which is described in Table 2 and Fig. 2.

Field	Description
Inviter:	Customer of the Telco company who participates in the Invitation Service.
Invitee:	Friends from the *Inviter* who receive the invitation to participate in the service. Although it is not explicitly stated in the service description, it is assumed that the Invitees are not customers of the telco company yet.
Marketing:	Staff from the telco company responsible for launching the service and analyzing its impact.

Table 1 Actors dictionary

Field	Description
Unique ID:	UC_IS_01
Use Case Name:	Invite Friend
Related to:	TELCO_S_03
Summary:	
Actors:	Inviter, Invitee
Preconditions:	The Inviter is a customer of the telco company.
Triggers:	The telco sends a message presenting the promotion and giving access to the Inviter application
Steps:	1. Inviter receives the promotional message from the telco company. 2. In case Inviter is interested in the promotion, Inviter accesses to a web application where he can select his friends and movies to send the invitation. 3. The system checks the invitation and recipient info, the maximum invitations allowed in the promotion, and if the email of invitee is correct. 4. In case any Invitee confirms the invitation, the inviter receives a free access to a movie.
Alternatives:	1. If Inviter is not interested in the promotion, after receiving the promotion, the service does not start. 2. If the recipient address is wrong, the system notifies Inviter. 3. If Inviter sends more than 10 invitations, the Inviter receives a message pointing out that he has reached the allowed limit. 4. It can be assumed that the promotion code expires after some period. If an invitee accepts the invitation after the expiration deadline, the service will notify inviter and invitee about this issue, without contacting the Billing WS.
Postconditions:	If the use case succeeds, the following postconditions are met: 1. Inviter has free access to so many movies as invitees have accepted the promotion. 2. Invitees accepting the promotion have free access to the movie selected by the Inviter. 3. Telco company has collected marketing data from Invitees.
Additional material:	See Fig. 2.

Table 2 Use case Invite Friend

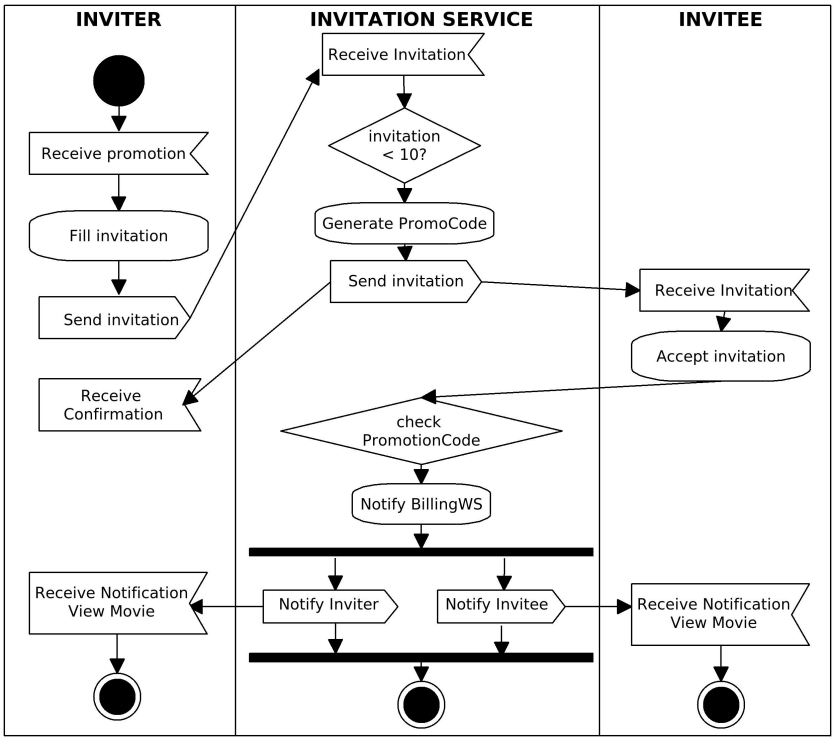

Fig. 2 Activity Diagram of Invitation Service

3 The Romulus approach

Romulus (*Domain Driven Design and Mashup Oriented Development based on Open Source Java Metaframework for Pragmatic, Reliable and Secure Web Development*) [42] is a European R&D project whose aim is improving productivity, reliability and security of Java Web Applications. With this purpose, the Romulus consortium has combined industrial partners, such as Informática Gesfor (Spain), Asset Data (Italy), IMola Informatica (Italy) and Liferay (Germany), as well as academic partners such as Universidad Politécnica de Madrid (Spain), NUIG-DERI (Ireland) and ICI (Romania).

This chapter presents Roma Metaframework (Sect. 5) and MyCocktail, the Romulus Mashup Builder tool 6.2, developed within the project Romulus.

Romulus is focused on the definition of a tool-supported lightweight process that simplifies Java web development. The process for developing applications and services is outlined in Fig. 3.

The first step in Romulus is the definition of a domain model, which is built using DDD principles. Before building the domain model, the bounded context of the model is identified using service oriented principles. The overall service landscape is

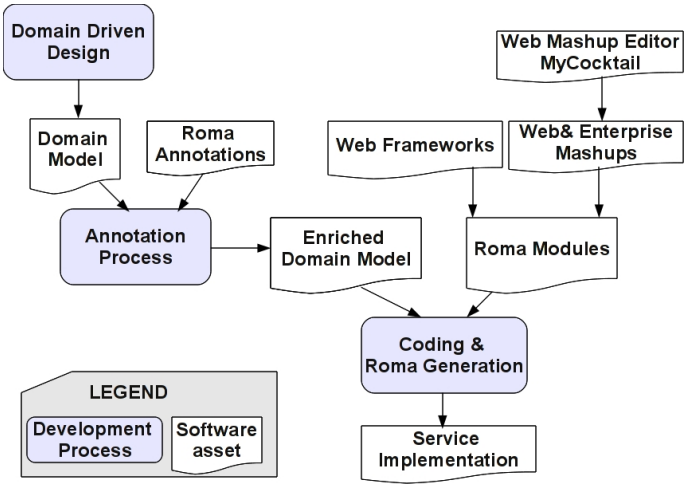

Fig. 3 Romulus Approach to Service development

determined and scoped. Then, the domain model is created to identify and document the key entities and the common domain vocabulary, shared among all the domain stakeholders.

The second step consists of transforming the domain model in annotating the domain model using attribute oriented programming. This step is supported by Roma Metaframework, which provides an abstraction layer for integrating cross-cutting concerns (aspects) in the service. This allows to isolate the domain model from the cross-cutting concerns. An additional advantage of the metaframework is that it provides a generic interface to some of the most popular web frameworks, which also isolates the application from a particular technology.

The third step involves coding, adding Roma modules that implement the Roma aspects, and execute and test the application. This step may include the development of services using Enterprise Mashup or Web Mashup facilities.

The main benefit of this approach is that Roma provides a uniform, consistent and stable environment to the developer, reducing the technological complexity. In this way, the developer can focus on modeling and understanding complex domains. In addition, the concept of metaframework reduces the risk of technological selections, since applications and services can evolve to other technologies without modifying the original application, and only a module for that new technology should be developed or used in case it is already available.

4 Domain Driven Design

4.1 Overview

The term *Domain Driven Design* (DDD) was coined by Eric Evans in his homonymous book [15]. The basic idea of DDD is that engineers should focus not in the technology they use, but in the understanding of the application model. Understanding the system to be developed is the main problem for succeeding in software projects. Traditional software engineering has put emphasis on analysis models based on UML notation, while Agile methodologies have focused on the code itself. DDD proposes a trade-off solution: its model driven approach is based on agreeing on a common language which describes the model and can be assisted by graphical notations. In addition, this model should be maintained very close to its implementation. Its main characteristics are:

- An *ubiquitous language* [15] should be used throughout the development. An *ubiquitous language* is a language that is shared between technicians and domain experts. DDD encourages the usage of this common vocabulary and terminology for building the domain model, which can be extended and understood by technicians and domain experts without extra-translations, resulting in richer semantic models. Although graphical models with notations such as UML can help in the communication, the primary description of the model should be explicit with natural language using the *ubiquitous language*.
- The model should be a *rich semantic model*, where objects have behavior and enforced rules, instead of just a database schema. The purpose is that domain experts should be able to feed the model with their deep knowledge of the business.
- The model should be *bound to its implementation*. Instead of maintaining an analysis model disconnected from its code, DDD proposes to describe the model in a way that the mapping with the code is straightforward. In order to maintain this close mapping, DDD organizes the application in a layered architecture, which offers the building blocks of the domain. In addition, DDD promotes the usage of design patterns (value objects, repositories, etc.) and refactoring techniques.

  Romulus follows a full DDD approach to applications and service development.

Context identification

The goal of this preliminary task is the identification of the bounded context of the domain model, in order to define explicitly the context in which the model is applicable as well as the relationships of the domain model with external systems. With this purpose, a service oriented approach is followed, and the service landscape is determined. A Service Engineering Methodology such as the one by Bayer [6] can be used with this end. Nevertheless, the lack of a rich domain modeling can lead to

a *Fat Service Layer* [38] or an *Anemic Domain Model* [38, 17], with duplicated objects and business logic is distributed among multiple objects.

The need of including domain modeling in service modeling has also been pointed out previously. Boroumand [7] declares "*service-orientation and object-orientation are not the same. Each is distinct with its own goals and approaches. However, by understanding that one has roots in the other, we can leverage established practices and techniques and incorporate them*". Her proposal extends MSOAM (Mainstream SOA Methodology) [1] with RUP (Rational Unified Process) [28].

Domain Model Design

The purpose of this activity is the design of the domain model, using a common vocabulary (*ubiquitous language*) which facilitates the communication among technical and business stakeholders.

The activity starts with the development of an initial design based on users' indications. This initial class diagram should include all the elements required to develop the task on progress.

Then, the initial design is refactored through the application of DDD patterns and best practices, and keeping the original functionality. Next, a set of steps are exposed in order to refactor the initial domain model.

1. *Analyzing associations:* An association establishes a relationship among two or more objects. The model has to be as simple as possible and avoiding complicate associations is a good mechanism to achieve this. Some ideas to simplify associations are: imposing a direction; adding a qualifier, reducing multiplicity; and eliminating non-essential associations.
2. *Entities, value objects and services:* The model must be clear, distinguishing between entities and value objects. An entity is a fundamental concept: it requires an unique attribute to identify it because its state can change during software execution. Meanwhile, a value object only describes a characteristic of the domain. A value object has no identity and can be shared. In some cases, making it immutable improves implementation features.
   Finally, in a domain model there are services. These represent processes that are not responsible of any entity or value object in particular. But sometimes services are overused and functions that correspond to the business logic of an object are implemented as a service. A good practice to avoid this situation is to use a verb to name the service.
3. *Using aggregations:* An aggregation is made up of a set of associated objects, but only one of those (the root) can be referenced by objects that are outside the aggregation. The objects involved in an aggregation acts as a unit, facilitating the management of complex associations between objects.
4. *Selecting repositories:* Another pattern to get a good domain model is the use of repositories. A repository manages the storage of a concrete type of objects. It implies the implementation of typical operations over a database like adding,

editing and removing elements and querying facilities. Repositories are domain objects associated with an aggregate that manage the data storage and retrieval, abstracting persistence mechanisms required to perform these operations.

5. *Using factories to create objects:* Another issue to consider is the object creation and the possibility of using factories for this purpose. A factory defines the creation process of an object. Using a factory is useful only when the object to create is complex. In the rest of cases its use can complicate the process and it is better to use a simple constructor.
   A factory for a value object is not the same thing as a factory for an entity. In the first case, the factory has to completely define the process of creation, because value objects are immutable and they need to be fully described since their creation. Meanwhile, the state of an entity can change during software execution, and its factory only has to define some characteristics of it. Especially, we cannot forget its identifying attributes.
6. *Validating the model:* At this point, using several scenarios to confirm that the model fits the requirements is a good practice to go ahead with security. We must ensure that tasks to develop in this iteration can be implemented.
7. *Dividing the model into modules:* In the development of an enterprise application, the collaboration of many people that work in parallel is required, being necessary to divide the domain model into a set of modules. In order to maintain the integrity of the system, each module has to be defined with a *bounded context*. There has to be a continuous integration and it is also advisable to represent the relationships among the models involved in the system in a map context.

4.2 Case study

The case study involves the interaction of several external services, including a movie renting facility, a social network, a billing service and a SMS messaging service. Based on Fig. 2, several services can be identified that interact with the envisioned Invitation service, as shown in Fig. 4.

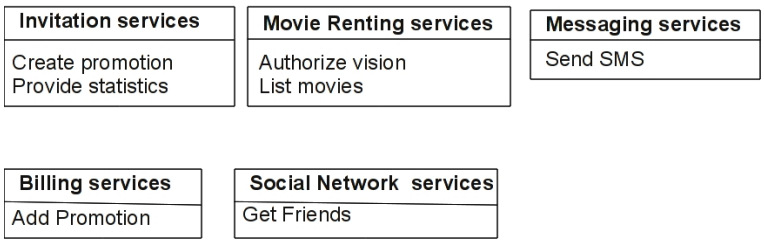

Fig. 4 Service landscape of the Invitation Service

The invitation service should provide two different user interfaces, one for the Inviter and another one for the Invitees. The interactions of the Invitation Service with the services identified are shown in Fig. 5.

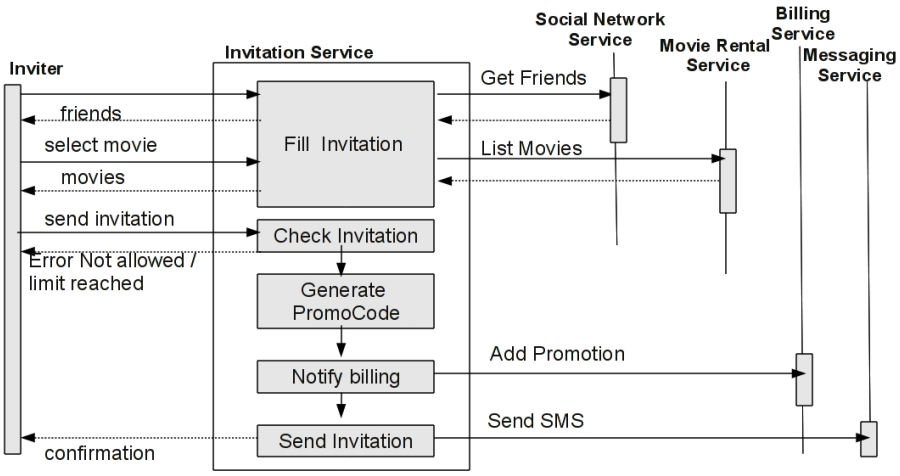

Fig. 5 Interaction Diagram for the Inviter Application

Once the service landscape for the considered bounded context has been analyzed, the next step is the domain design of the service. In this step, domain elements of the service model are identified (therefore creating an ubiquitous language) and modeled. This will produce the domain model of the bounded context.

The service model that can be seen in Fig. 4 involves several concepts that comprise the ubiquitous language of the domain model:

- *Promotion*. This is a promotion set by the telco company. It is associated with a set of invitations. Initially, one invitation is sent to a user, who can create more invitations up to a maximum number that is set in the promotion.
- *Invitation*. This is the invitation that a user receives from another user. It is associated with a movie. It also has a creation date, which determines expiration, and a state, that represents if the invitation has been rejected or accepted.
- *Movie*. This represents the movie that can be watched by invitees.
- *SMS*. This represents the messages that are sent to each user and which contain the invitations. They are sent by a user to another user.
- *User*. A user is the Invitation Service user that can potentially send and receive invitations.
- *Statistics*. These are the statistics that can be retrieved from the Invitation Service, and which are thus associated with a promotion.

The next step is identifying entities and value objects. On the one hand, *entities* are objects that have an identity, and therefore require an internal ID to identify them. On the other hand, *value objects* are identified by the value of their attributes.

In our domain model, movies, invitations, users, and promotions are entities, which require an identifier in order to differentiate them. Therefore, two different movies might have the same title, as well as two users could have the same name, while being different objects. Statistics are value objects, since it is not necessary to track the particular object with an identifier. Similarly, SMSs are value objects, as they act only as temporary objects that are built in order to interact with the Messaging Service.

Services were already identified previously, and typically consist of cross-cutting operations that cannot be included in any particular class. The entities' lifecycle determines the domain model as well. The Invitation Service covers the promotions' lifecycle, the Social Network Service covers the users' lifecycle, and the Movie Renting Service covers movies' lifecycle. Invitations' lifecycle needs to be managed as well, so an InvitationFactory and an InvitationRepository should be defined, which would allow invitations creation and storage, respectively. However, this is not necessary in the Romulus Framework, as persistent objects are managed through a Persistence Aspect, which abstracts these common operations. Finally, SMSs and statistics are value objects and therefore do not require factories or repositories to manage their lifecycle.

After creating an ubiquitous language, identifying entities and value objects, and defining services, factories and repositories, the resulting domain model can be seen in Fig. 6.

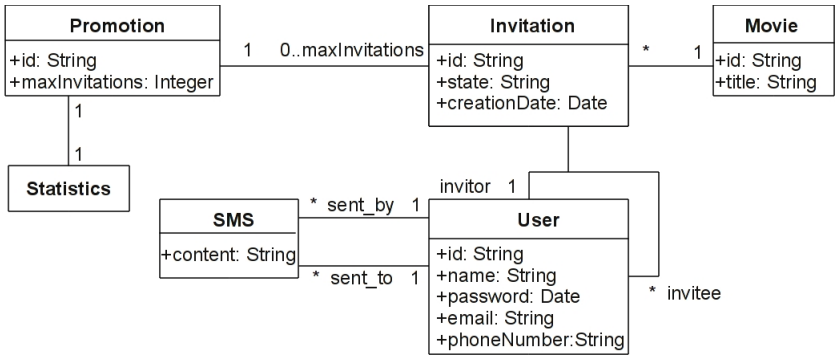

Fig. 6 Domain Model for the Invitation Service

5 Roma metaframework

5.1 Roma overview

Romulus proposes that developers should be focused on understanding and developing a domain model as presented in the previous section. Then, thanks to the usage of Roma Metaframework, services and applications can be developed in a straightforward way.

Roma Metaframework [41, 50] is the main result of the project Romulus, and is available as an open source project. Roma provides a full approach to Java Web based development. It is based on three design decisions: (1) POJO orientation, (2) metaframework notion and (3) Attribute Oriented Programming for enriching the domain.

First of all, Roma follows a full POJO (*Plain Old Java Object*) orientation. POJOs [16, 45] are just simple Java objects which encapsulate the business logic. In Roma, everything is programmed as a POJO. For example, at the interface level, each screen is modeled as a POJO, and each component (menu, button, area, ...) in the screen is also modeled as a POJO. At the service level, invoking a web services is done by calling a method or exposing a web service is done by annotating a method. The main benefits of this approach are [45]:

- *Reduction of complexity*, thanks to the separation of concerns, the developer can focus only on implementing the domain model with POJOs, without worrying about other aspects such as persistence, transactions, or web flow.
- *Improvement in productivity*. The developer can develop and test the service as traditional objects.
- *Better portability*, since the domain implementation is not tight to a specific implementation technology, such as EJB (Enterprise Java Beans).

Second, Roma is based on the notion of a *metaframework*. POJOs are simple and good for testing, but they do not provide support for many of the common needs when developing a service, such as registering services, invoking other services, presenting a user interface, administering users, persisting or maintaining the session through the service lifetime. Web frameworks such as Struts [21], Struts2 [9], Spring [54] or Hibernate [57] come in for solving these needs. Nevertheless, web languages and frameworks [26] lack of consolidation, since the current generation of web languages and frameworks reflects the frenetic pace of technological development in the area. The main undesired consequences of this constant evolution are the (i) increasing complexity of of web applications combining different web technologies [10]; (ii) the need to migrate between framework versions [48] with impacts directly on the application reusability and (iii) the high required skills to develop a full web application using several web frameworks. The Roma metaframework concept tries to overcome these problems by providing an abstraction layer on top of the most popular web frameworks. The metaframework has identified the most common aspects of the existing web frameworks, and has defined interfaces for these aspects.

Then, frameworks can be used by implementing adaptors to these metaframework interfaces, following the *plugin* design pattern [44].

Finally, Roma separates application's business logic from the infrastructure specific concerns through the usage of *annotations*. *Attribute Oriented Programming (A@P)* [49, 53, 47] is a code-level marking technique. Developers can mark code elements (e.g. classes, methods, fields) using *annotations* to indicate that they have application specific or domain specific semantics. This approach has been followed in several languages, such as Java [18] or C# [24]. For example, a developer may mark a method with a *logging* annotation to indicate that the associated calls to this method should be logged. These annotations are preprocessed by an annotation processor that generates the final detailed code. In this example, the generator may insert logging code in the methods annotated with the *logging* annotation.

In order to combine these approaches, Roma proposes to implement the domain model using POJOs. Since "cross-cutting concerns" are not present in the domain model, they are modeled through metaframework *aspects*. Aspects represent independent views of the application that affect different logical or domain units. Roma uses annotations to associate aspects with POJOs. Then, the metaframework provides *modules* which implement one or more aspects, or provide some common functionality. In this way, applications are defined in a technologically independent way according to the different aspects. Then, different modules can be plugged in or out later on. The metaframework will allow that the implementation of the application remains untouched.

To sum up, in a Roma application the domain concepts are defined through POJOs. Annotations are included in these POJOs to define aspect details. Finally, modules are selected to implement these aspects.

Roma brings a number of benefits. First, it provides a a stable framework based on automatic code generation techniques. Second, the manual migration process between frameworks can be avoided thanks to the metaframework notion, saving the company investments. Finally, Roma reduces considerably the need to master different web frameworks and their evolution.

5.2 Roma Metaframework Model

The domain model is the center of Roma Metaframework Model. Roma Metaframework has identified and implemented defines a set of cross-cutting concerns (aspects) that can be needed for developing a web application. Other aspects can be easily integrated thanks to its pluggable architecture.

The main elements of Roma metaframework model are (Fig. 7):

- *Domain Model*. The domain model, as presented before, follows a DDD approach and is implemented with POJOs.
- *Aspects*. Each aspect of the metaframework defines a common functionality needed for developing an enterprise application. Each aspect defines a set of interfaces and has a set of associated annotations. An *enriched domain model* is an

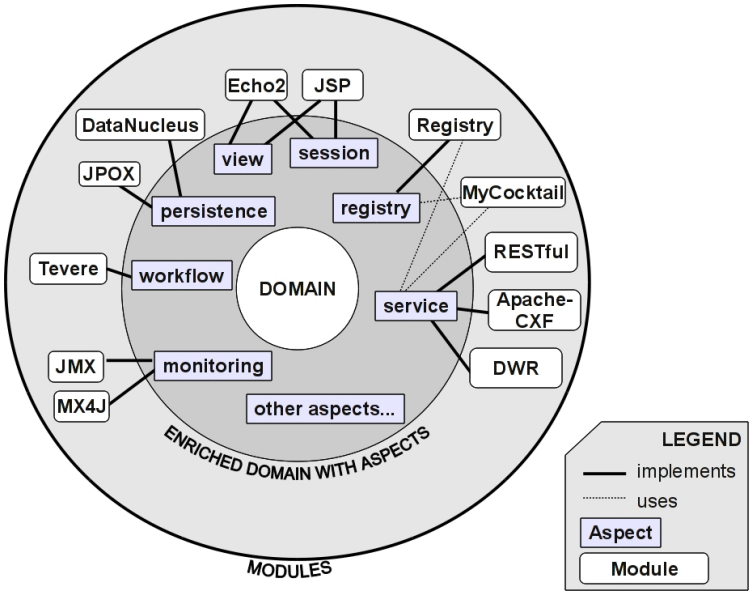

Fig. 7 Simplified Roma Metaframework Model

annotated domain model with annotation aspects, which can use aspect interfaces for its business method implementation. Annotations can be defined using standard Java annotations [18] or Roma XML Annotations. Roma XML annotations have been defined in order to allow to keep the domain model separated from the infrastructure's needs, and improving reusability. In any case, both annotation mechanisms can be used.

- *Modules*. Modules may implement or use one or more aspect interfaces, providing an implementation of some functionality. Each aspect can have more than one module that implements or uses it.

Next, a summary of the main aspects defined in Roma Metaframework is presented, in order to provide a better understanding of its applicability.

View aspect. This aspect is the basis for the automatic generation of a graphical interface for an application, based on an abstract interface layer, which can be mapped on different interface technologies (frameworks). This aspect defines several annotations in order to define the layout (*ViewClass*, *ViewField* and *ViewAction*) of the POJO components. Based on this specification or additional configuration, Roma selects a graphical renderer for each POJO and its fields and methods. Roma provides modules for rendering with Java Server Pages technology [5] and the Echo2 [14] framework. This is one example of the benefits of the metaframework concept. The interface of the application can be changed without modifying the application code, just changing the module it uses.

Validation aspect. It provides validation facilities for POJOs fields. Roma provides the annotation *ValidationField* which provides attributes for defining if the

field is mandatory (attribute *required*), allowed length (attributes *min* and *max*) or required string patterns (attribute *match*).

Internationalization aspect. This aspect enables to present the application in different languages, according to the current location. *I18N Aspect* is implemented as a part of the core module and is based on Java resource bundles.

Authentication aspect. Authentication is the process of verifying that someone is who claims to be. Therefore, this aspect is useful to control users access to some application functionalities that require a security access. Authentication aspect is based on the *users module*, that provides user management and profiling facilities. There are three modules for implementing this aspect using token, LDAP or password authentication methods.

Security aspect. This aspect allows the developer to define the users' permissions on POJOs at class, field or action level. It provides annotations with this purpose (*SecurityClass*, *SecurityField* and *SecurityAction*). Developers can restrict these permissions (read, write and execution) based on user roles or access control lists.

Flow aspect. The flow aspect is used to define the execution flow of the application. Implementation of this aspect is based on core module which presents a Java annotation, called *FlowAction*, that allows developers to define next classes to be executed and another class in case of error.

Session aspect. This aspect collects all the business logic for managing user session, and is used by the modules that implement the View Aspect.

Workflow aspect. This aspect provides a generic interface to a workflow engine. Roma provides their own web-based workflow engine, so called *Tevere*, which has been developed using Roma Metaframework.

Logging aspect. This aspect facilitates the use of logs to control the execution of the application and defines the *LoggingAction* annotation. The logging aspect is implemented by the *admin module*.

Monitoring aspect. With this aspect, business objects can be monitored externally, being accessible outside the application. The annotations *MonitoringClass*, *MonitoringField* and *MonitoringAction* are used to define classes, fields and actions to be monitored. Monitoring aspect can be implemented with JMX [39] or MX4J [34] modules.

Persistence aspect. This aspect is used to store Java objects and retrieve them from a database. Roma provides modules base don the frameworks JPOX [27] and Datanucleus [12] technology. The persistence aspect is based on JDO (*Java Data Objects*) [52]. and provides interfaces for most common persistence tasks, such as creation, updating, query or deletion of objects.

Reporting aspect. This aspect automates the generation of reports from the POJOs annotations. View annotations are used to generate a template that can be customized in order to obtain the desired result. This aspect is implemented by *Reporting-JR module*, which is based on JasperReports library [25].

Service aspect. The purpose of the service aspect is to facilitate the usage of services from a POJO. The Service aspect provides facilities for exposing a POJO as a web service and creating a client for a web service, The aspects provides an annotation (*ServiceClass*) for exposing an interface as a service as well as interfaces

for service invocation. There are three modules implementing this aspect, in order to provide support to Web Services standards (*Apache CXF* module) [4], REST services (RESTful module) and integration with JavaScript client-side services based on DWR [13].

Registry aspect. The registry aspect provides a registry interface and annotation (*RegistryClass*) for WSDL [11] services and REST services described with WADL [19]. There is module implementation based on WSO2 registry [56].

Enterprise Aspect. The enterprise aspect exposes Roma Services in an Enterprise Service Bus (ESB). The aspect provides two annotations: *EnterpriseClass* and *BPELClass*. The *EnterpriseClass* annotation exposes a service in an ESB previously annotated with the service aspect. The *BPELClass* indicates that the service should be exposed in the ESB through a BPEL [2] delegation process. The current module implementation is based on OpenESB [35].

Scripting aspect. This aspects adds server-side scripting capabilities to Roma applications, which leverages the facility to develop and modify functionalities with scripting languages, such as JavaScript.

Semantic aspect. This aspect provides facilities for taking advantage of semantic web facilities. The aspect defines annotations (*SemanticClass*, *SemanticField*) for exposing a POJO as RDF. A Jena [32] based module provides an implementation of this aspect.

These aspects provide a uniform modeling paradigm to the developers, who do not need to integrate, understand, and evolve each of the frameworks that provide the needed facilities.

Further details of these aspects and the usage of the Roma Metaframework can be found in the Roma Handbook [50].

5.3 Case Study

The implementation phase takes advantage of Romulus' DDD-based approach to software development by providing a simple mapping between design and implementation elements. Domain classes and design aspects have their corresponding Java classes or annotations that implement the model that was defined in the design phase.

In Romulus, each element in the domain model is implemented as a POJO, while cross-cutting aspects are implemented through the use of Java annotations. A set of annotations are defined for each of the available aspects considered in Romulus. The resulting annotated POJO is an enriched model of the domain model that was defined in the design phase.

Therefore, by following our case study, the following Java classes should be defined: Promotion, Statistics, Invitation, SMS, User, and Movie. Each of these Java classes are POJOs that include the different private attributes and public accessors, which can be seen in Fig. 6. Also, methods that implement the business logic of each POJO need to be implemented.

Java annotations are added to this basic POJO structure, and enrich the model by defining secondary aspects. The Java classes that were mentioned previously might require configuration for the different aspects, so a set of annotations need to be added to each POJO. Romulus will set sensitive defaults for each aspect, but further configuration is often needed. Namely, we should expect to include annotations regarding the view aspect. Most times, not all the defined fields have to be shown to the users. For each field that defaults do not apply, a *ViewField* annotation should be set. Similarly, methods' visibility can be customized by including a *ViewAction* annotation.

We will focus on implementing the business logic and user interface of the invitations, thus illustrating the usage of Romulus' view aspect (for UI implementation), validation aspect (for the definition of model constraints).

The Invitation class is shown in Fig. 8.

In that code fragment, attributes are defined for the Invitation class according to the model defined in the design phase. Notice that *creationDate* is automatically assigned with the current date whenever an invitation is created. Additionally, methods are defined for each business logic action. For instance, the *getMovies* method queries an external service to obtain the movies that are available in the system.

Also, annotations for Roma Validation and View aspects have been set:

- The *ViewField* annotation is employed to hide fields that should not be visible when showing an invitation object on the presentation layer.
- *ValidationField* annotations are used to indicate that the value for the state field is constrained to a couple of values, as long as invitations can only be pending or accepted, and that other fields are required. Also, a *validation* method has been added to perform further checks before persisting an invitation.

In addition, note that no persistence annotations need to be set, as the Persistence Aspect (which allows storing and querying for objects in the database) abstracts all mappings between class definitions and the database schema.

This class is enough to produce the output that is shown in Fig. 9. The user interface that is shown in that picture makes use of two different presentation frameworks (Janiculum and Echo2). Switching between them requires changing a configuration option, and serves as example of the flexibility of the metaframework approach. Aspects are therefore a powerful mechanism in Romulus that allow keeping a clean POJO implementation while adding important details that extend to other concerns, such as presentation, validation, persistence or security.

6 Mashups in Romulus

6.1 Overview

Mashups [59] are web applications that combine existing data sources to create new applications in order to provide a value added to these data. Mashups may play an

```
public class Invitation {
  @ViewField(visible=false)
  @ValidationField(required=true)
  private String id;

  @ViewField(visible=false)
  @ValidationField(match="(pending|accepted)")
  private String state = "pending";

  @ViewField(visible=false)
  private Date creationDate = new Date();

  @ViewField(visible=false)
  @ValidationField(required=true)
  private User inviter;

  @ValidationField(required=true)
  private User invitee;

  @ViewField(visible=false)
  @ValidationField(required=true)
  private Promotion promotion;

  @ValidationField(required=true)
  private Movie movie;

  @ViewField(render = ViewConstants.RENDER_TEXTAREA)
  private String invitationText;

  @ViewField(selectionField = "entity.movie", render = "select",
             description = "Select movie")
  public Movie[] getMovies() {
     Map<String,Object> paramMovie = new HashMap<String,Object>();
     paramMovie.put(InvokeServiceCommand.SERVICE_URL, "http://www.movies.com/");
     paramMovie.put(InvokeServiceCommand.OPERATION_NAME, "getMovies");
     InvokeServiceCommand isc = new InvokeServiceCommand();
     return (Movie[]) isc.execute(new CommandContext(paramMovie));
  }

  @ViewField(selectionField = "entity.invitee",
             render = "select", description = "Select friend")
  public User[] getInvitees() {
        // Invoke Social network Web Service as in getMovies()
  }

  @ViewAction(visible = false)
  public void validate() throws ValidationException {
      if (!Promotion.availableInvitations(inviter)) {
         throw new ValidationException(this, "inviter",
                          "No invitations left", null);
      }
  }

  @ViewAction(visible = false)
  public void generatePromoCode() {
       // Generate Discount Coupon
  }

  @ViewAction(visible = false)
  public void notifyBilling() {
       // Invoke billing web service as in getMovies()
  }

  @ViewField(visible = false)
  public void sendInvitation() {
       // Invoke SMS web service as in getMovies()
  }
  /* Public getters and setters are ommitted */
}
```

Fig. 8 Sample implementation of Invitation with Roma Annotations

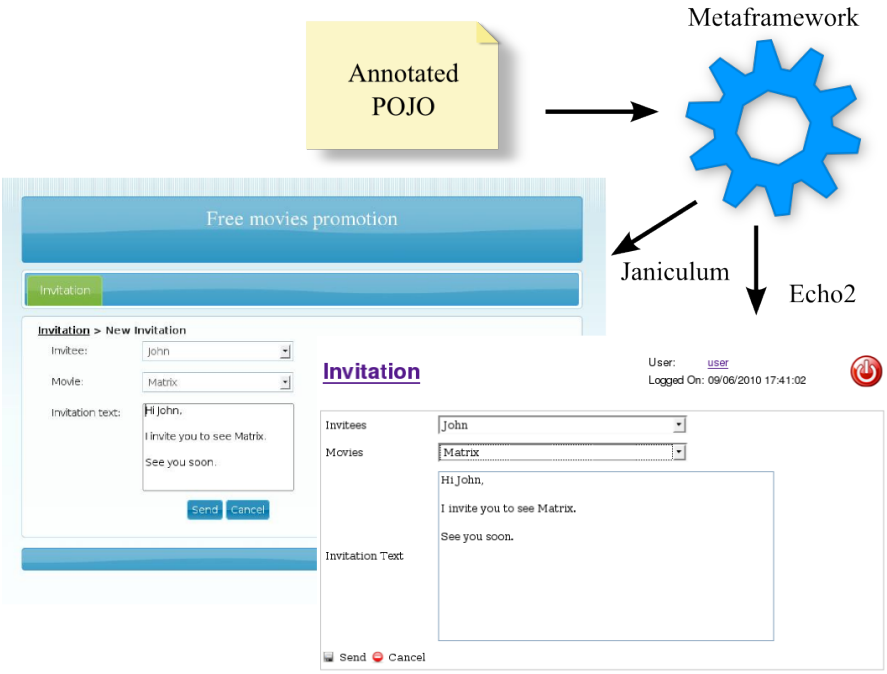

Fig. 9 Invitation Service's user interface

important tool in service engineering [30], since they provides a flexible and easy to use way for service composition on web.

The architecture of mashup web applications [33] is composed of three parts:

- **The content provider:** the source of the data, the data providers often expose their content through web-protocols such as REST, Web Services, and RSS/Atom. To obtain data, mashups can use a technique known as screen scraping which consists in extracting data from the display output of another program that is intended to be display to a human user so it is usually neither documented nor structured for convenient parsing.
- **The mashup site:** part where the mashup logic resides but it is not necessarily where it is executed. Mashups can be implemented using traditional server-side dynamic content generation technologies like Java Servlets, CGI, PHP or ASP and alternatively, mashed content can be generated directly within the client's browser through client-side scripting (JavaScript) or applets. Mashups can also use a combination of both server and client-side logic to achieve their data aggregation.

- **The client web browser:** the user interface of the mashup where the mashup is rendered graphically and where user interaction takes place. As described above, mashups often use client-side logic to assemble and compose the mashed content.

Developers can build a mashup using the conventional web programming technologies. However, in the last years many dedicated mashup tools have been released, like Google Mashup Editor, Microsoft Popfly, Yahoo Pipes [58] or Intel Mash Maker [22]. These tools enable quick development and allow creating mashups in an easy way, some of them even can be used by end users in order to compose their own mashups. The Romulus project provides its own mashup builder, which is called MyCocktail. is fully integrated with Roma Metaframework to retrieve the data exposed by the application through the services to build mashups.

6.2 MyCocktail

MyCocktail [43], the Romulus Mashup Builder, is a web application which provides a graphical user interface for building mashups easily, allowing the user to develop mashups faster and, thus, increasing the productivity.

This tool allows users to combine information obtained from different services. This information can be modified with operators and later presented with a wide variety of renderers. The whole process is developed with a graphical user interface and it is as easy as to drag and drop some components and combine them. This helps to reduce considerably the time for developing a mashup.

MyCocktail allows designers and programmers to use services without dealing with the low-level details. Users only have to fill a simple form and the tool processes it and makes the requests to the different services.

MyCocktail is based on Afrous [40] and provides three kinds of components which should be combined to build a mashup:

- **Services:** Several RESTful services can be invoked: del.icio.us (tags, posts, etc.), Yahoo Web Search, Google AJAX Search, Flickr Public Photo Feed, Twitter, Amazon, etc. These are the default services but MyCocktail allows to get data from any REST service which provides a JSONP response using the JSONP operator. MyCocktail also automatically imports the REST services of the Roma Metaframework which are annotated by the developer. The integration between the Roma Metaframework and MyCocktail is made with WADL [19] as format to interchange information about the services created in the applications made with the Roma Metaframework.
- **Operators:** The information obtained can be processed with the operators. For example, it is possible to sort, filter or group the information by a given parameter.
- **Renderers:** the information can be represented in different renderers:

- **HTML renderers:** these renderers generate HTML elements (image, link or table tags). One renderer can be displayed into others, while the output of a renderer can be used as input of another renderer.
- **Statistic renderers:** two kinds of statics diagrams can be generated: pie chart and bar chart.
- **Google Maps renderer:** a renderer that shows elements on a map.
- **Timeline renderer:** a renderer that shows a time line with elements sorted by date.

MyCocktail (Fig. 10) provides an action panel for designing the mashup flow. Services, operators and renderers may be dragged and dropped onto this action panel for its combination while designing a mashup. All the operators, services and renderers are represented in the panel as a form with some fields. By submitting the form, the result of each operator is shown in the lower part of them. The output of one component is usually used as input of another one to combine them.

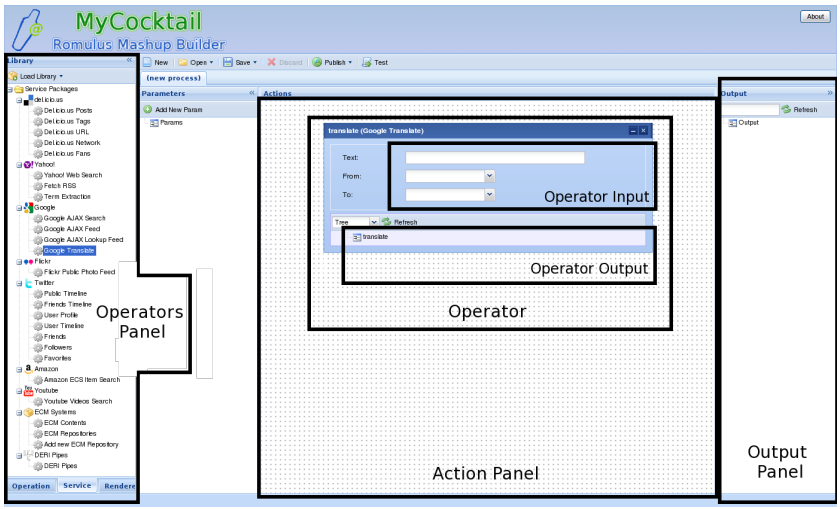

Fig. 10 MyCocktail Mashup Builder Structure

6.3 Mashups in Roma

Once we have presented the mashup technology and the MyCocktail tool, this section presents how Roma Metaframework can benefit from using mashup technology. This section describes two scenarios.

The first scenario consists of mashup combining data sources, external web services, enterprise web services , and Roma services and is shown in Fig. 11.

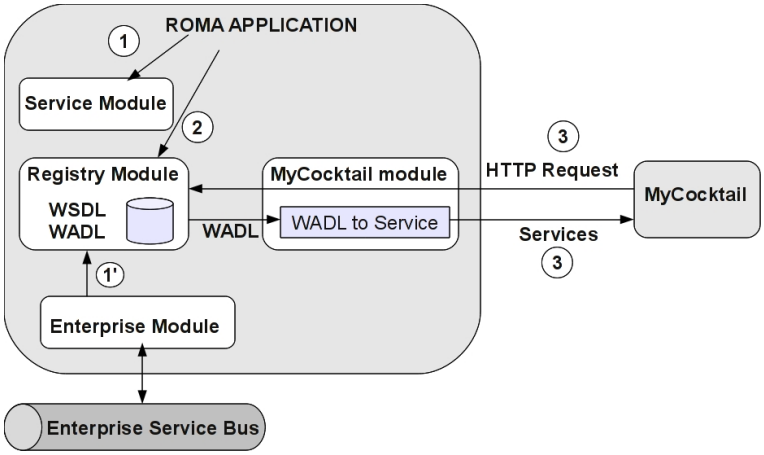

Fig. 11 MyCocktail Module Architecture

As presented before, the *Service Aspect* allows a Roma Application to expose the methods of an interface as web services, which can be registered in a Service Registry using the *Registry Aspect*. The *Registry aspect* has been extended in order to be able to register services developed with MyCocktail and described with WADL. In addition, MyCocktail has been integrated as a module in Roma using the Service and Registry aspects. The main purpose is that MyCocktail can auto discover the registered services. In this way, MyCocktail mashup facilities can be used for combining Roma Services. Furthermore, since MyCocktail registers its services in the Registry, Roma Applications can use directly the mashups developed with MyCocktail.

In addition, Roma provides an *Enterprise aspect* that exposes a service in an enterprise bus. The service can be exposed as a delegated BPEL process. The purpose of this aspect is the integration of *Enterprise Mashups* in Roma. Enterprise mashups are different from web mashups since they are server side and combine business processes. The integration between Roma and Enterprise and web mashups is shown in Fig. 11, First, Roma can expose services in the registry using the service and registry aspects. In addition, the *Enterprise aspect* can be used to publish a service in an ESB. Then, MyCocktail mashup builder discovers all the services of the registry.

As a conclusion, Roma applications can be easily integrated with Enterprise and Web Mashups without requiring to modify its architecture or technology. The usage of mashups can leverage the service development effort. For example, Enterprise mashups based on BPEL orchestration has been used in a project management tools developed with Roma, while the control dashboard has been developed with web mashups.

The second scenario consists of the server-side mashup execution in Roma Metaframework (Fig. 12).

Thanks to the Roma *Scripting aspect*, Roma POJOs can implement methods with scripting languages such as JavaScript. Since MyCocktail exports the mashups in

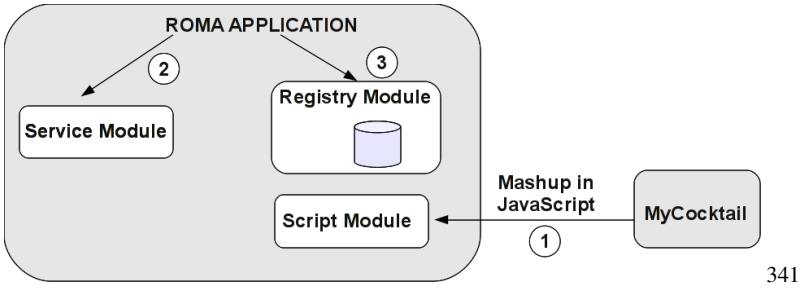

Fig. 12 Server side execution of Mashups in Roma

JavaScript, some mashup processes can be moved to the server side. The main advantages of this approach are:

- Security reasons. The mashup can contain sensible business logic that should be hidden. In addition, this can provide secure access to other server resources, such as legacy applications, intranet services and ESB.
- The mashup can have direct access to server resources without the need to expose them. For example, the mashups can query directly the persistence layer.

6.4 Case Study: Development of movie selection with MyCocktail

The selection of the movie (second step in Fig. 5) will be implemented through a mashup. This mashup should retrieve the movies registered in the promotion (Movie Renting Service). The information provided by this service is not too extensive, so we will obtain these data from an external service which provides the plot, reviews, runtime, etc. and mix these data with the existent information of the movie. All the information will be showed to the users translated to their preferred language to help them choosing the movie to recommend to their friends.

The steps followed for building this mashup are shown in Fig. 13.

The first step consists of retrieving the movie data using the aforementioned service. The service returns all the available movies offered in the promotion, and the tool shows the results in a tree-like mode.

The second step involves invoking an external service to retrieve the information of the movies. This search should be made over each recovered movie. The Iterate operator provides this functionality, and allows to design an internal flow for its application over a collection of elements. In our case, the flow applied to each element of the collection is based on a movie search operation based on their titles. Thus, the information of all the movies is retrieved as a whole.

In order to translate the movie reviews, there is an available translator operator, which can be configured with the origin and target languages. This translator operator is applied in the internal flow of the Iterator operator.

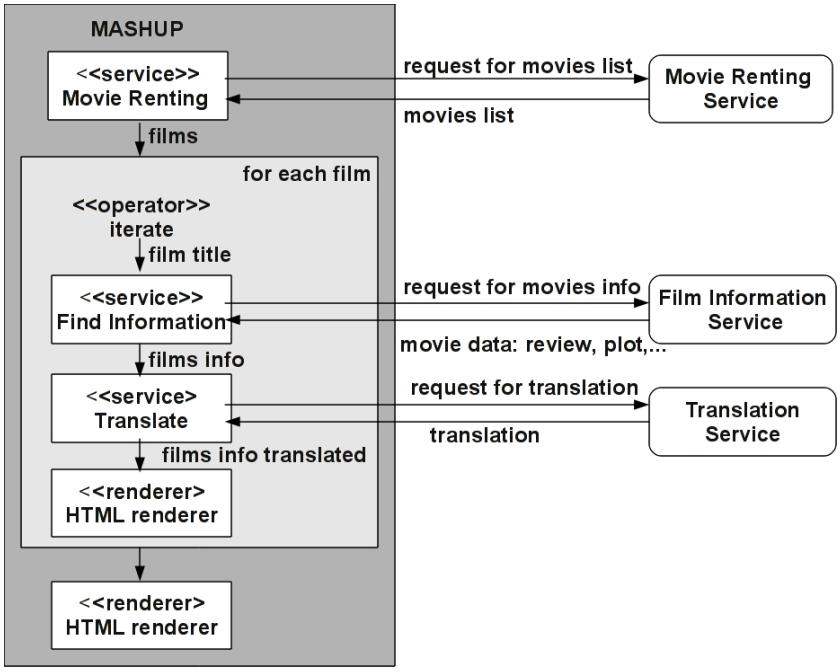

Fig. 13 MyCocktail mashup architecture for movie selection

Finally, when the mashup operations are concluded, the resulting mashup can be represented with some graphical gadgets (pie chart, ...) and then exported as a Google widget, a Netvibes widget, JavaScript or HTML.

As a conclusion, mashup technology provides benefits in development and maintenance tasks. Roma provides several alternatives for its seamless integration presented in this section.

7 Related work

This chapter has proposed an agile service development model based on Roma Metaframework following a DDD approach. In addition, the chapter has shown how this approach can benefit from enterprise and web mashup technologies. In this section, we review other approaches also focused on improving productivity in web development.

7.1 Agile Web Frameworks

Agile web frameworks started with Ruby On Rails [20, 23], which defined a new approach to web development, based on a single web framework. Most of the agile web frameworks follow a DDD approach to domain modeling. Some of the most popular agile web frameworks are Grails, Trails, Ruby on Rails and OpenXava, which are introduced below.

Grails [46] is a Java-based Rails-like development framework that was built in response to Ruby on Rails. As a result, their principles are the same and Grails is heavily Rails inspired. To provide Java integration while providing a dynamic oriented language, Grails is based on the Groovy language, a dynamic object-oriented scripting language for the Java virtual machine and with Java-like syntax.

Trails [51] is a web development framework that is inspired in Ruby on Rails and Naked Objects [29]. Its target is offering domain driven design by providing a full-stack web application framework. As a result, Trails takes advantage of the stability and maturity of a closed set of already existing frameworks. Trails enhancements to the direct use of the frameworks are tight integration and automatic code generation for common tasks.

Ruby on Rails [20] is a framework that is aimed at agile development of web applications. It was mainly developed by David Heinemeier Hansson and was extracted out of Basecamp, a production-ready commercial web application. Ruby on Rails' community argues that this extraction is the best proof of the framework's suitability for the development of web applications. Nowadays, Ruby on Rails have gain great popularity. Some of their mains characteristics are metaprogramming, migration facilities and the active records.

OpenXava [36] is an agile framework that follows a POJO orientation. It is developed by Javier Paniza and is integrated with JPA and portal technologies.

The main difference of Roma metaframework is the notion itself of metaframework. The metaframework offers a common application programming interface to a set of pluggable Java frameworks to transparently provide persistence, presentation or internationalization services. In addition, the Roma target is POJO-based development with a minimum coupling with pluggable underlying frameworks and, following the model driven architecture paradigm, provide framework-specific code generation for eventual fine tuning of code. Another differential characteristic is the integration of mashup technology in Roma. Roma provides a mashup repository and registry to provide server-side facilities.

7.2 Mashups for service composition

Mashup technology is considered a promising technolgoy for closing the gap in Service-to-User interaction in Service Oriented Architecturas (SOA) [**?**], since mashups provide a user-centric and participative approach to service composition [**?**].

Regarding *service composition approaches*, BPEL [2] has shown effective for service orchestration, although it is primarily targeted at professional developers. Given its complexity, several proposals have emerged [37] for its extension for REST services. Other approaches have integrated BPEL in their mashup development environments, such as LiquidApps [**?**], providing a visual representation of the underlying processes. The approach followed on Romulus has been targeted at technical users, which have to develop complex applications on Java platforms. In order to reduce BPEL complexity, Romulus provides BPEL integration through the Enterprise Aspect in Roma and its integration with MyCocktail Mashup tool through the Registry Aspect.

Maximilien et al. [31] address service composition with mashups by defining a domain specific language, that is processed for generating the targetted platform. This work points out one of the challenges of mashup technology, its interoperability between mashup editors, which is still an open issue.

Liu et al. [30] proposes to extend SOA with mashup concepts. The proposed architecture integrates a service component in the mashup component. MyCocktail follows a similar approach, since it integrates a service component that are registered in a REST registry using WADL. Our approach has been tested in the integration of enterprise mashups with the *Enterprise Aspect* and with Semantic DERI Pipes Mashups [55], thanks to the *Semantic Aspect*.

Model driven approaches have been used in order to automate web engineering processes. For instance, a web specific modeling language (WebML) is used for composing web services [8] or web applications [**?**, 3]. WebML conceives data intensive web applications with two models: data model, which describes the schema of data resources, and schema model, which describes how data resources are assembled into information units and pages. WebML is supported by a graphical environment and supported by an IDE tool. Roma follows a different approach based on a rich object model, while navigation is expressed in the Flow Aspect.

8 Conclusions

This chapter has presented the problems in web development due to the frenetic evolution of web frameworks. In order to simplify maitenance and development activities, Romulus proposes to follow a DDD approach and define a rich domain model. This domain model allows the interaction throughout the project among business domain experts and engineers. In this way, business domain experts can contribute in testing activities, as well as in the evolution of the domain model. Since the domain model is close to the code, the evolution of the code is straightforward with the domain model.

Romulus provides Roma metaframework that provides a layer of abstraction on top of existing web frameworks. As shown in the article, the metaframework provides a large set of functionalities, including semantic integration, automatic user interface generation, and integration with web and enterprise mashups.

The chapter has shown several architectures for integrating mashup technology with Roma metaframework.

The chapter has illustrated the main concepts of Romulus through the development of a case study for the development of invitation service.

Acknowledgements Our work has partly been supported by the European Commission under Grant No. 217031, FP7/ICT-2007.1.2, project Romulus – "Domain Driven Design and Mashup Oriented Development based on Open Source Java Metaframework for Pragmatic, Reliable and Secure Web Development"

References

1. *SOA Priciples of Service Design*. Prentice Hall, 2007.
2. S. Askary C. Barreto B. Bloch F. Curbera M. Ford Y. Goland A. Guizar N. Kartha C. Liu R. Khalaf D. Koenig M. Marin V. Mehta S. Thatte D. van der Rijn P. Yendluri A. Yiu A. Alves, A. Arkin. Web services business process execution language (BPEL) version 2.0. Technical report, Committee Specification, OASIS, jan 2007.
3. Roberto Acerbis, Aldo Bongio, Marco Brambilla, Stefano Butti, Stefano Ceri, and Piero Fraternali. Web applications design and development with webml and webratio 5.0. In Richard F. Paige and Bertrand Meyer, editors, *TOOLS (46)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 392–411. Springer, 2008.
4. Apache CXF Project. Apache CXF web site. an open source service framework. available at http://echo.nextapp.com/site/echo2.
5. Karl Avedal, Ari Halberstadt, Danny Ayers, Timothy Briggs, Carl Burnham, Ray Haynes, Hen, Stefan Zeiger, and Mac Holden. *Professional JSP*. Wrox Press Ltd., Birmingham, UK, UK, 2000.
6. Joachim Bayer, Michael Eisenbarth, Theresa Lehner, and Kai Petersen. Service engineering methodology. In *Semantic Service Provisioning*, chapter 8, pages 185–201. 2008.
7. Solmaz Boroumand. Working with SOA and RUP. *SOA Magazine*, (XVI), 2008.
8. Marco Brambilla, Stefano Ceri, Sara Comai, Piero Fraternali, and Ioana Manolescu. Model-driven specification of web services composition and integration with data-intensive web applications. *IEEE Data Eng. Bull.*, 25(4):53–59, 2002.
9. Don Brown, Chad Davis, and Scott Stanlick. *Struts 2 in Action (In Action)*. Manning Publications Co., Greenwich, CT, USA, 2008.
10. Richard Cardone, Danny Soroker, and Alpana Tiwari. Using xforms to simplify web programming. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 215–224, New York, NY, USA, 2005. ACM.
11. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. W3c note, World Wide Web Consortium, March 2001.
12. DataNucleus Project. DataNucleus web site. available at http://www.datanucleus.org.
13. DWR Project. DWR (Direct Web Remoting)web site. available at http://directwebremoting.org/dwr/index.html.
14. Echo2 Project. Echo2 web site. available at http://echo.nextapp.com/site/echo2.
15. Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.
16. Martin Fowler. POJO (Plain Old Java Object). *Martin Fowler, available at http://martinfowler.com/bliki/POJO.html*, 2000.
17. Martin Fowler. Anemic domain model. *Martin Fowler, available at http://martinfowler.com/bliki/AnemicDomainModel.html*, 2003.

18. James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Upper Saddle River, NJ, 3. edition, 2005.
19. Marc J. Hadley. Web Application Description Language (WADL). Available at https://wadl.dev.java.net/wadl20090202.pdf. feb 2009.
20. Steve Holzner. *Beginning Ruby on Rails (Wrox Beginning Guides)*. Wrox Press Ltd., Birmingham, UK, UK, 2006.
21. Ted N. Husted, Cedric Dumoulin, George Franciscus, and David Winterfeldt. *Struts in Action — Building Web Applications with the Leading Java Framework*. Manning Publications, 2003.
22. IBM. Intel mash maker. Available at http://mashmaker.intel.com.
23. José Ignacio Fernández-Villamor, Laura Díaz-Casillas, and Carlos Á. Iglesias. A comparison model for agile web frameworks. In *EATIS '08: Proceedings of the 2008 Euro American Conference on Telematics and Information Systems*, pages 1–8, New York, NY, USA, 2008. ACM.
24. ECMA International. *Standard ECMA-334 - C# Language Specification*. 4 edition, June 2006.
25. JasperReports Project. JasperReports web site. available at http://jasperforge.org/projects/jasperreports.
26. Mehdi Jazayeri. Some trends in web application development. In *FOSE '07: 2007 Future of Software Engineering*, pages 199–213, Washington, DC, USA, 2007. IEEE Computer Society.
27. JPOX Project. JPOX (java persistent objects) web site. available at http://www.jpox.org.
28. Philippe Kruchten. *Rational Unified Process. An Introduction.* Addison-Wesley, 2004.
29. Konstantin Läufer. A stroll through domain-driven development with naked objects. *Computing in Science and Engineering*, 10:76–83, 2008.
30. Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. Towards service composition based on mashup. In *IEEE SCW*, pages 332–339. IEEE Computer Society, 2007.
31. E. Michael Maximilien, Ajith Ranabahu, and Karthik Gomadam. An online platform for web apis and service mashups. *IEEE Internet Computing*, 12:32–43, 2008.
32. B. McBride. Jena: a semantic web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.
33. Duane Merrill. Mashups: The new breed of Web app. Available at http://www.ibm.com/developerworks/xml/library/x-mashups.html. aug 2006.
34. MX4J Project. Open source JMX for enterprise computing (MX4J) web site. available at http://mx4j.sourceforge.net/.
35. OpenESB. OpenESB project, available at https://open-esb.dev.java.net/. Available at https://open-esb.dev.java.net/.
36. openxava. openxava project, available at http://www.openxava.org/web/guest/home. Available at http://www.openxava.org/web/guest/home.
37. Cesare Pautasso. Restful web service composition with bpel for rest. *Data Knowl. Eng.*, 68(9):851–866, 2009.
38. Srini Penchikala. Domain driven design and development in practice. *InfoQueue*, jun 2008.
39. J. Steven Perry. *Java Management Extensions*. O'Reilly, Beijing, 1. edition, 2002.
40. Afrous Project. Afrous project web site, 2009. Available at ttp://www.afrous.com/.
41. Roma Metaframework project. Romulus web site. available at http://www.romaframework.org/.
42. Romulus project. Romulus web site. available at http://www.ict-romulus.eu/.
43. Romulus Project. Mycocktail web site, 2009. Available at http://www.ict-romulus.eu/web/mycocktail.
44. David Rice and Matt Foemmel. *Plugin Design pattern*, page 499. Addison-Wesley, 2002.
45. Chris Richardson. *POJOs in Action: Developing Enterprise Applications with Lightweight Frameworks*. Manning Publications Co., Greenwich, CT, USA, 2006.
46. Graeme Rocher. *The Definitive Guide to Grails (Definitive Guide)*. Apress, Berkely, CA, USA, 2006.
47. Romain Rouvoy. Leveraging component-oriented programming with attribute-oriented programming. In *In Proccedings of WCOP 2006*, 2006.
48. Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 471–480, New York, NY, USA, 2008. ACM.

49. Don Schwarz. Peeking inside the box: Attribute oriented programming in java. *ONJava.com, O'Reilly*, 2004.
50. Emanuele Tagliaferri, Giordano Maestro, Luca Garulli, Luca Molino, Luigi Dell'Aquila, and Marco de Stefano. *Roma MetaFramework Handbook v2.1*. Romulus Project, 2.1 edition, dec 2009.
51. Trails. Trails project, available at http://www.trailsframework.org/. Available at http://www.trailsframework.org/e.
52. Sameer Tyagi, Michael Vorburger, Keiron McCammon, and Heiko Bobzin. Core Java Data Objects. Prentice Hall PTR / Sun Microsystems Press, 2004.
53. Hiroshi Wada and Shingo Takada. Leveraging metamodeling and attribute-oriented programming to build a model-driven framework for domain specific languages. In *In Proc. of the 8th JSSST Conference on Systems Programming and its Applications*, 2005.
54. Craig Walls and Ryan Breidenbach. *Spring in Action*. Manning, 2005.
55. Adam Westerski. Integrated environment for visual data-level mashup development. In *10th International Conference on Web Information Systems Engineering (WISE)*, pages 481–487, 2009.
56. WSO2. Wso2 registry, available at http://wso2.com/products/governance-registry/. Available at http://wso2.com/products/governance-registry/.
57. Ming Xue and Changjun Zhu. Design and implementation of the hibernate persistence layer data report system based on j2ee. In *PACCS*, pages 232–235. IEEE Computer Society, 2009.
58. Yahoo. Yahoo pipes. Available at http://pipes.yahoo.com.
59. Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding mashup development. 2008.