

IST PROJECT 2001-35399



A Governmental Knowledge-based Platform for Public Sector Online Services

Project Number:	IST-2001-35399
Project Title:	A Governmental Knowledge-based Platform for Public Sector Online Services
Deliverable Type:	Public

Deliverable Number:	D51-D61
Contractual Date of Delivery:	31-1-2003
Actual Date of Delivery:	28-2-2003
Title of Deliverable:	Low-level Specifications of SmartGov Services and Applications and the Knowledge-Based Core Platform WP5-WP6
WP contributing to the Deliverable:	WP5-WP6
Nature of the Deliverable:	Report
Editor(s):	Stelios Gorilas
Author(s):	Stelios Gorilas, Pablo Fernandez Pardo, Tomas Pariente Lobo, Costas Vassilakis, Akriki Katifori, Anna Charissi, George Lepouras, Stathis Rouvas, , Nick Adams, John Fraser, Ann Makynthos,

Abstract: This deliverable constitutes the Software Architecture Document of the SmartGov platform. It is a holistic approach and reports the outcome of the analysis and design phase of both the SmartGov services and applications as well as the Knowledge based components.

Project funded by the European Community under the "Information Society Technologies" Programme (1998-2002)

© Copyright by the SmartGov Consortium.

The SmartGov Consortium consists of:

Partner's Name	Acronym	Role	Country
University of Athens	UoA	Project Coordinator	Greece
T-Systems Nova	TNB	Partner	Germany
Indra Sistemas S.A.	Indra	Partner	Spain
Archetypon S.A.	ARC	Partner	Greece
Napier University	NU	Partner	UK
General Secretariat for Information Systems	GSIS	Partner	Greece
City of Edinburgh Council	CEC	Partner	UK

2.5.3.1	Software module structure.....	99
2.5.3.2	SGA-IIG Communication Details.....	100
2.5.3.3	Remote administration of the IIG.....	122
2.5.3.4	Summary	125
2.6	Process View	125
2.6.1	The SmartGov Front-End.....	125
2.6.2	The Integrator	126
2.6.3	SmartGov Agent - Information Interchange Gateway.....	127
2.6.3.1	Synchronous communication.....	127
2.6.3.2	Asynchronous communication	128
2.6.3.3	Periodic events.....	129
2.6.3.4	Notifications.....	129
2.6.3.5	Logging facilities.....	130
2.7	Deployment View.....	130
2.7.1	SmartGov Front-End.....	131
2.7.2	Integrator	132
2.7.3	SmartGov Agent - Information Interchange Gateway.....	132
2.7.3.1	Service delivery platform.....	132
2.7.3.2	Organisational Information System Environment	133
2.8	Data view	136
2.8.1	Introduction	136
2.8.2	Modelling of Transaction Services.....	137
2.8.3	Modeling of Forms.....	140
2.8.4	Modelling of Generic TSEs	143
2.8.5	Modelling of Instantiated TSEs	146
2.8.5.1	Inheritance rules for TSE instantiation.....	149
2.8.6	Modelling of Generic TSE Groups.....	150
2.8.6.1	Repetition information for groups	152
2.8.7	Modelling of Instantiated TSE Groups.....	153
2.8.7.1	Inheritance Rules for TSE Groups	156
2.8.8	Utility types.....	157
2.8.8.1	Complex Type administrativeInfo.....	157
2.8.8.2	Complex Type containedTSE	157
2.8.8.3	Complex Type formSet	157
2.8.8.4	Complex Type formStatistics.....	158
2.8.8.5	Complex Type KUToHelpItem.....	158
2.8.8.6	Complex Type lifeCycleType.....	158

2.8.8.7	Complex Type method	159
2.8.8.8	Complex Type multilingualText.....	160
2.8.8.9	Complex Type repetitionInformation	160
2.8.8.10	Complex Type TSEDataType	161
2.8.8.11	Complex Type TSEGroupStatistics	162
2.8.8.12	Complex Type TSEStatistics.....	162
2.8.8.13	Complex Type TSEToFormElement.....	163
2.8.8.14	Complex Type TSStatistics.....	163
2.8.8.15	Complex Type validationMethod	164
2.8.8.16	Complex Type validationMethodStatistics.....	165
2.8.9	Modeling of Knowledge Units	165
2.8.10	Modeling of Taxonomy	188
2.8.11	Modeling of Taxonomy node	191
2.8.12	Modeling of Workflow.....	194
2.8.13	RDBMS Data Model.....	204
2.8.13.1	Users, Roles, and Work Groups	204
2.8.13.2	Service Design Environment Statistics.....	206
3	Conclusions	208
4	References.....	209

Table of Figures

Figure 1 Overview of the SmartGov platform	12
Figure 2. The 5+1 views of Architecture	15
Figure 3 – XML description of a form.....	27
Figure 4 – XML description of two TSEs	28
Figure 5 – Rendering of identification data form in a browser.....	28
Figure 6 – XHTML code for identification data form	29
Figure 7 – Transformed XHTML code for mapping form elements to TSEs	34
Figure 8 – Rendered form for the purposes of mapping	35
Figure 9 – Main SmartGov Front-End Navigation Diagram	37
Figure 10 – Portal Navigation Diagram Explanation.....	38
Figure 11 – SmartGov Menu Front-End Navigation Diagram.....	39
Figure 12 – SmartGov Task List Front-End Navigation Diagram	40
Figure 13 – KU Life-Cycle	40
Figure 14 – SmartGov TS Editor Front-End Navigation Diagram	41
Figure 15 – SmartGov Form Editor Front-End Navigation Diagram	41
Figure 16 – TS Editor Diagram Explanation.....	42
Figure 17 – Form Editor Diagram Explanation	43
Figure 18 – SmartGov Managerial Statistics Front-End Navigation Diagram	44
Figure 19. The Model-View-Controller Architecture	46
Figure 20 The Struts Architecture	47
Figure 21. The XML Doc Repository Architecture	50
Figure 22 XML Doc Repository API class diagram.....	51
Figure 23 – SmartGov Front-End logical architecture components	53
Figure 24 – SmartGov Front-End interaction between components architecture.	54
Figure 25 – Example of use of Struts in the Front-end tool	56
Figure 26 –Sequence diagram using Struts in the Front-end tool	58
Figure 27 – SmartGov service design objects logical class diagram.....	59
Figure 28 – SmartGov Knowledge Objects logical class diagram	61
Figure 29 – SmartGov KM & Services relationships logical class diagram	62
Figure 30 – SmartGov life-cycle rules	62
Figure 31 – SmartGov life-cycle logical class diagram	63
Figure 32 – SmartGov and Delivery Environment Role architecture.....	64
Figure 33 – User and Roles logical class diagram	66
Figure 34 – SmartGov Front-End XML data access logical class diagram	69
Figure 35: Communication with installed IT systems.....	71

Figure 36: Placement of SmartGov Agents in the overall architecture	71
Figure 37 – Class diagram for the SGA package	76
Figure 38 – The IIG package class diagram	77
Figure 39 – Class diagram for the SGA package	78
Figure 40. The software environment of the integrator.....	81
Figure 41 Performing TSE mappings	83
Figure 42 Struts Files Generator classes.....	84
Figure 43 Transforming an XHTML template to a JSP page.....	85
Figure 44 Creation of internationalized resources	86
Figure 45 Development of a custom validation language	87
Figure 46 Generation of Java validation code from the custom SmartGovValLang	89
Figure 47 Action Form Java file generator.....	90
Figure 48 Generating a DB schema and relevant DB access objects	91
Figure 49 – Three layer Front-end structure implementation diagram	93
Figure 50 – Front-end Presentation Layer implementation diagram.....	94
Figure 51 –Front-end Business Layer implementation diagram.....	95
Figure 52 –Front-end Data Layer implementation diagram	97
Figure 53 Integrator implementation diagram	98
Figure 54 SGA-IIG: Sequence of messages needed for a generalized form of communication.....	100
Figure 55 SGA modules	106
Figure 56 IIG overview	108
Figure 57 IIG modules.....	110
Figure 58 General Structure of the SGA and the IIG-PAQ.....	111
Figure 59 IIG modules.....	117
Figure 60 – SmartGov Front-end web application schema.....	126
Figure 61 SmartGov platform network topology	131
Figure 62 – SmartGov Roles, Groups and Users database schema.	206
Figure 63 – Service design environment statistics database schema.	207

List of Acronyms

Acronym	Explanation
APAQ	Adelante (Outgoing) Pending Actions Queue
API	Application Programming Interface
EPAQ	Entra (Incoming) Pending Actions Queue
IIG	Information Interchange Gateway
IIG-MYP	Information Interchange Gateway – Minimal Yoking Processor
IIG-NI	Information Interchange Gateway Notification Initiator
IIG-SEP	Information Interchange Gateway – Separate External Process
IT	Information Technology
JSP	Java Server Page
KU	Knowledge unit
MVC	Model-View-Controller
PAQ	Pending actions queue (in the context of the SmartGov Agent and the Information Interchange Gateway)
PAQUED	Pending Actions Queue Dispatcher (in the context of the SmartGov Agent and the Information Interchange Gateway)
SGA	SmartGov agent
SGA-NI	SmartGov Agent Notification Interceptor
SGovApp	SmartGov application
TS	Transaction service
TSE	Transaction service element
XML	Extensible Markup Language
JSP	Java Server Page
XHTML	eXtensible Hypertext Markup Language
RUP	Rational Unified Process
UML	Unified Modeling Language
WAP	Wireless Application Protocol
XSLT	Extensible Style sheet Language Template
WML	Wireless Markup Language
JDBC	Java Database Connectivity
API	Application Programming Interface
RDBMS	Relational Database Management System
LDAP	Lightweight Directory Access Protocol
DSN	Data source name
SOAP	Simple Object Access Protocol

DDL	Data Definition Language
-----	--------------------------

Executive Summary

The SmartGov project, fully entitled as "A Governmental Knowledge-based Platform for Public Sector Online Services", commenced on the 1st of February 2002. In the workpackages that have been completed insofar, the state of the art has been reviewed (WP3), the current status of the participating public authorities has been captured (WP3), the user requirements have been analysed (WP4) and the high-level specifications of the system have been derived (WP4). This document, which reports on work carried out in WP5 and WP6 provides detailed system specifications for the architectural modules identified in WP4 as parts of the SmartGov platform.

The low-level specifications of the platform documented in this deliverable address different views of each architectural module, tackling all functionality, modelling, management, development and deployment aspects. Each view may be considered individually, enabling thus more fine-grained task allocation and increasing the degree of parallelism that can be achieved in the implementation phase. The system specifications reported in this deliverable constitute a roadmap for the implementation phase, since they not only provide the implementation details for each individual module, but they also document the interfaces through which different modules interact (procedure calls, shared objects in persistent storage, messages exchanged), catering thus for module interoperability.

In order to enhance the document readability, a brief summary of the platform architecture, as derived in WP4, is included at the beginning of section 2.

1 Introduction

The present deliverable constitutes the Software Architecture document of the SmartGov platform. It is the result of the first iteration of the analysis and design phase as implied by the Rational Unified Process [RUP]. In the context of the SmartGov project it comprises an interim report of the two development work packages namely "WP5: Development of SmartGov Knowledge-Based Core Components" and "WP6: Development of SmartGov Applications and Services". In fact it is a joint deliverable of the two work packages in contrast with what was stated in the technical annex. The decision to join the two deliverables was taken in order to avoid redundancy between the different documents since there is a significant overlapping between the two concepts behind the two work packages. Thus in this document a holistic approach in the SmartGov platform is presented based on different views of its architecture. In the use case view a user approach of the platform is outlined. The basic features and principles of the platform are presented. In the logical view of the platform the functionality of the platform is given by presenting the architecturally significant components. Some of these components have a very low level design to the point where classes, class members and associations are presented. On the other hand other components have not yet reached this level of design since they depend on the design of the former. Elaboration of these will take place at the next iteration of the process. The implementation view of the platform presents the software management of the platform while the process view deals with the computer processes invoked by the applications. In the deployment view of the platform the system topology where the platform will be installed is presented. Finally in the data view section the persistent storage of the platform is covered mainly presenting the XML Schemas [XML] that outline the structure of the data needed for the creation of e-forms applications.

2 SmartGov Platform Architecture

2.1 Overview of the Platform

In the following paragraphs, the high-level architecture of the SmartGov platform, as documented in D4.1, is summarised. This section aims to provide the reader with a global view of the SmartGov platform and outline the modules involved in the development and delivery of transaction services. These modules are detailed in the main part of this deliverable.

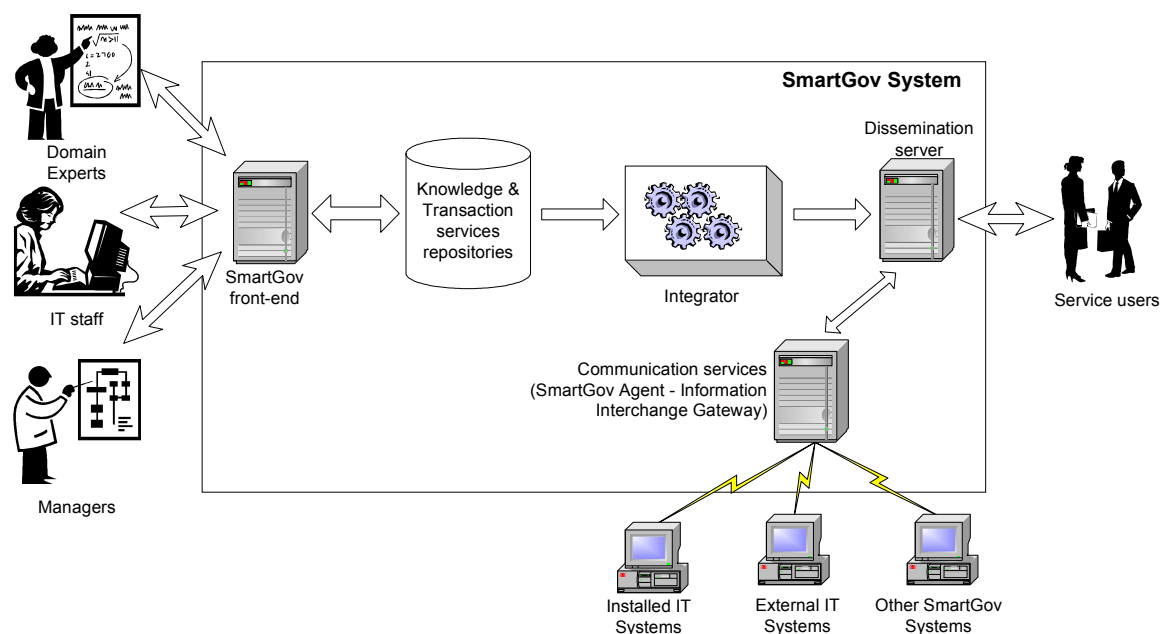


Figure 1 Overview of the SmartGov platform

Figure 1 illustrates the SmartGov platform architecture, in which the following modules may be identified:

- The SmartGov knowledge and transaction services repositories. These are general depots for storing organisational knowledge and information pertaining to the transaction services that are developed using the SmartGov platform. In order to provide a semantically rich environment and facilitate extensibility and interoperability, all data is stored in XML format.
- The SmartGov front-end, which constitutes of personalised application development environments which are available to the actors involved in the lifecycle of electronic transaction services, namely domain experts, IT staff and managers. The actors employ the SmartGov front end to

populate, query and modify the knowledge and transaction services repositories.

- The integrator, a software module that reads the contents of the knowledge and transaction services repositories, and automatically generates all necessary elements (files, objects, components etc) for a fully operational transactional service. These elements are then deployed on a dissemination server, initiating service delivery to the users.
- The communication services, comprising of two units, namely the SmartGov agent and the information interchange gateway. This module provides generic communication mechanisms with installed IT systems for the purposes of data exchange, hiding idiosyncrasies and peculiarities of information system platforms and facilitating resilience against temporary failures.

2.2 Methodology

The SmartGov project will follow the Rational Unified Process after careful consideration of available software engineering methods [Pressman2000][Quadrani2000]. This approach states the software engineering process as an incremental iterative process and uses the new standard: Unified Modelling Language (UML). The incremental iterative process means that phases of development are performed not strictly sequentially, but rather they are partially overlapping. The project life is divided into small phases, which have to be refined with the results of previous phases.

The main characteristics of the Unified Process are the following:

- The Unified Process is an iterative process. Given today's sophisticated software systems, it is not possible to sequentially first define the entire problem, design the entire solution, build the software, and finally test the product. An iterative approach is required to allow an increasing understanding of the problem through successive refinements and to incrementally grow an effective solution over multiple iterations. This approach gives better flexibility in accommodating new requirements or tactical changes in business objectives and allows the project to identify and resolve risks earlier.
- Activities of the Unified Process create and maintain models. Rather than focusing on producing large amounts of paper documents, the Rational Unified Process emphasises the development and maintenance of models -

semantically rich representations of the software system under development.

- The Unified Process focuses on early development and base lining of a robust software architecture, which facilitates parallel development, minimises rework, increases reusability and maintainability. This architecture is used to plan and manage the development around the use of software components.
- Development activities of the Unified Process are driven by use cases. The notions of use cases and scenarios drive the process flow from business modelling and requirements through testing, and provides coherent and traceable threads through both the development and the delivered system.
- The Unified Process supports object-oriented techniques. Several of the models are object-oriented models, based on the concepts of objects, classes, and associations between them. These models, like many other technical artefacts, use the Unified Modelling Language (UML) as the common notation.
- The Unified Process supports component-based software development. Components are nontrivial modules, subsystems that fulfil a clear function, that can be assembled in a well-defined architecture, either ad hoc, or some component infrastructure such as the Internet, CORBA, COM/DCOM. This way reusability of software is underpinned.

To support that methodology, we will use 5+1 views of software architecture as shown in the following figure.

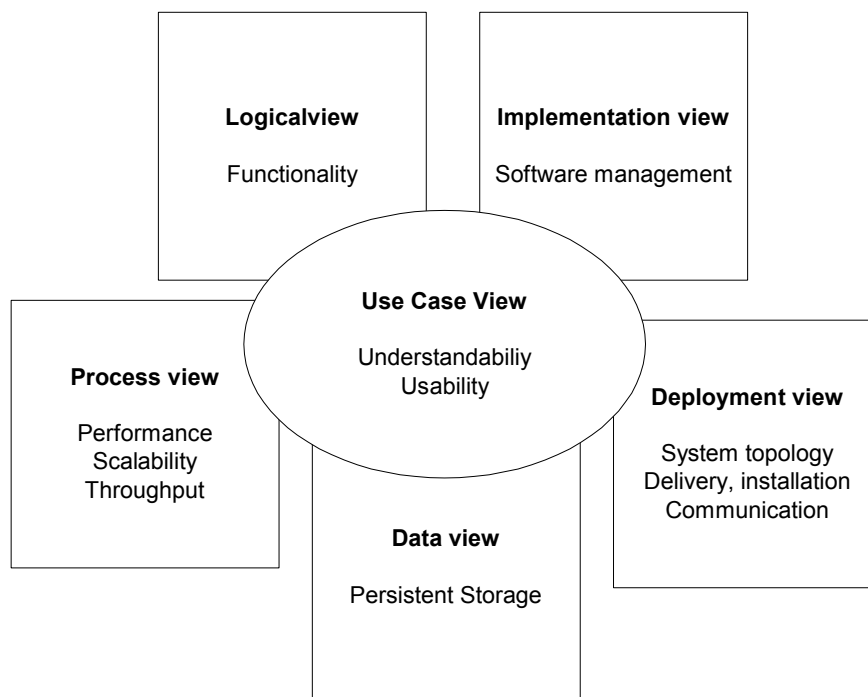


Figure 2. The 5+1 views of Architecture

Within these views of the Architecture some of the UML graphical notations are used in order to formally illustrate aspects of the system. Within this section the used graphical notations are briefly revisited.

2.2.1 Class Diagram

A class diagram is a graphic representation of the static structural model. It shows classes and interfaces, along with their internal structure and relationships. Classes represent types of objects that are handled in a system. A class diagram does not show temporal information, it describes only the classification. The instances of those types (objects) are instantiated only on the runtime and are represented by an object and interaction diagrams.

Classes can be related to each other in a number of ways: associated (connected to each other), dependent (one class depends/uses another class), specialized (one class is a subtype of another class), or packaged (grouped together as a unit - package). A class diagram does not express anything specific about the relationships of a given object, but it does abstractly describe the potential relationships of one object to other objects.

A system typically has a number of class diagrams - not all classes are inserted into a single class diagram. A class may have multiple levels of meaning and participate in several class diagrams.

2.2.2 Sequence Diagram

A sequence diagram shows interaction information with an emphasis on the time sequence. The diagram has two dimensions: the vertical axis represents time, while the horizontal axis represents participating objects. The time axis could be an actual reference point (by placing the time moments labels as text boxes). Horizontal ordering of the objects is not significant to the operation, and may be rearranged as necessary.

2.2.3 Implementation Diagram

Implementation diagrams help the developer describe the logical software structure inside a computer or across a large distributed system. Implementation diagrams show aspects of implementation, including source code structure and run-time implementation structure. They come in two forms: 1) component diagrams show the structure of the code itself and 2) deployment diagrams show the structure of the run-time system. They may also be applied in a broader sense to business modeling in which the "code" components are the business procedures and documents, while the "run-time structure" comprises the organizational units and resources (human and other) of the business.

2.2.3.1 Component diagram

A component diagram represents a physical structure of a code (as opposed to the class diagram, which portrays the logical structure) in terms of code components and their relationships within the implementation environment. A component can be a source code component, a binary component, or an executable component. A component contains information about the logical class or classes that it implements, thus creating a mapping from a logical view to a component view. Dependencies between the components are shown, making it easy to analyze how a change in one component affects the others. Components may also be shown with any of the interfaces that they expose. Components, as with almost any other model element, can be grouped into packages, much like classes or use cases. Component diagrams are used in the later phases of the software development, when there is a need to divide up classes among different components. When working with the CASE facilities, components are used for file-

class mapping during code generation, reverse engineering, and round-trip engineering operations.

2.2.3.2 Deployment diagram

Deployment diagrams show the physical layout of the various hardware components (nodes) that compose a system, as well as the distribution of executable programs (software components) on this hardware. You may show the actual computers and devices (nodes), along with the connections they have to each other, thus specifying system topology. Inside nodes, the executable components and objects are located in a way that shows where the software units are residing and on which nodes they are executed. You may also show dependencies between the components. Deployment diagrams are crucial when dealing with distributed systems.

2.3 Use-case view

2.3.1 The Transaction Service Lifecycle

Within the lifecycle of a transaction service (i.e. a service that includes filling and submission of forms, whose data are then processed by an organisational back-end system), the following phases may be identified:

1. The manager decides about a new service
2. The manager creates a working group, of domain experts, IT staff, managers and service workers.
3. The group creates the service requirements
4. The group creates the service specifications (process model)
5. The group develops the transaction service components
 - a. Forms (domain experts and possibly IT staff)
 - b. KUs: Knowledge Units (working group)
 - c. TSEs: Transaction Service Elements (domain experts)
 - d. Validation checks (IT staff and domain experts)
 - e. Links to back-end systems (Domain experts and possibly IT staff)
 - f. Managerial statistics (Managers and Domain experts)
 - g. IT-related statistics.
6. Integration of components
7. Testing
8. Evaluation
9. Deployment

10. Operation and maintenance
11. Feedback
12. Improvement
13. Discontinuation of a service

Not all of these phases will (or can) be supported by the SmartGov platform. In particular, phases 1-2 involve managerial actions, such as feasibility studies and human resource management. Within phases 3 and 4 the initial definitions and documentation (KUs) are collected and entered in the SmartGov platform. The SmartGov platform comes into full play during phases 5 and 6, where the various transaction service components are developed and integrated. After the integration step in phase 6, the electronic service is instantiated and installed on an internally accessible server for testing and evaluation. These phases may trigger further actions within phase 5, producing new versions of the electronic service, which are again tested and evaluated internally in phases 7 and 8 respectively. The SmartGov platform will not provide tools for service testing and evaluation, but is responsible for generating the instantiated service version.

When the service has reached a satisfactory state, it is deployed on a publicly accessible server (phase 9) so that it can be delivered to the end-users. Service deployment is similar to installing the service on the test environment of phases 7 and 8, with the only difference being the accessibility (and possibly the scale) of the server.

Once deployed a service enters the operation and maintenance phase (10), during which end users access the service. Throughout the operation and maintenance phase, feedback is collected both by end users and via the statistics collection mechanisms of the SmartGov platform (phase 11), which will be exploited for service improvement (phase 12). In these phases, the SmartGov platform offers support for statistics collection, user account management and database backup and recovery.

Finally, a service may be discontinued due to becoming obsolete due to legislation changes, owing to deadline expiration or even because it has not been proven to be popular enough to justify its delivery and maintenance costs. In all cases, delivery of the service through the dissemination platform should cease and, depending on (a) the possibility that the service will go live again and (b) organisational policy, it might be required that the SmartGov platform objects crafted specifically for this service will be purged.

In the following paragraphs, phases 3, 4, 5 and 6 will be covered in more detail.

2.3.2 The phases of developing a transaction service

2.3.2.1 Creation of a service

The creation of a service within the SmartGov platform takes place during phase 3. In this phase, the service name is entered, along with a high-level description of it. The description may document the overall service functionality, the result of the feasibility study and so on. Finally, some keywords may be entered, which will be drawn from the terms in the knowledge taxonomy or domain map.

2.3.2.2 Creation of a service process model

The service process model is created during phase 4. During this phase, the service operation, the roles involved, the business rules governing the service and the data that must be presented and/or collected are identified and documented. Portions of the documentation (e.g. supporting legislation, information regarding the workflow, service development expected time schedule) may be stored within the SmartGov platform as knowledge units associated with the whole service.

2.3.2.3 Development of Transaction Service Components

During this phase the various components of the electronic service are taking a concrete form within the SmartGov platform. The following paragraphs elaborate on the process of creating the different elements. It is important to note that, once the service process model has been defined within phase 4, the individual stages within phase 5 need not be carried out sequentially. For instance, the form layout may be developed in parallel with the TSEs or the KUs that will be placed on the form, and links to external IT systems can be established independently of all other activities.

Restrictions are placed on the development timeline only when a specific object depends on the existence of another: for instance a validation check involving two TSEs cannot be modelled until both involved TSEs have been defined.

2.3.2.3.1 Forms

Forms are the basic presentation and interaction unit for the end user of the transaction service. In the context of the SmartGov platform, a form is divided in two parts:

1. The *semantic part*, which defines what information is entered in the form, the validation checks that apply to the form and the knowledge units, which will be presented to the user.

2. the *layout part*, which defines the appearance of the elements on the client device through which the electronic service is accessed.

Although in an ideal world both these parts would be developed in an integrated environment, in the context of the SmartGov project this is not feasible because (a) developing a web page editor with modeling power and user friendliness comparable to the commercial tools service designers are used to work with, is a huge task outside the scope of the project (b) devoting person power in development of such a module is not in line with the objectives of the key action (c) existing products are "closed" platforms and cannot be extended. Taking these into account, the two parts will be developed independently as follows:

1. The semantic part is developed using the SmartGov development platform.
2. The layout part is developed outside the SmartGov platform using any appropriate tool for form layout definition that targets the dissemination channel through which the service will be delivered. For example, if the service will be delivered through the WWW, HTML form editors should be employed (e.g. DreamWeaver, FrontPage, vi etc.); if the service will be delivered through the WAP, a WAP page editor (3TL WBuilder, Rasquares Wap, vi etc) might be used. For services that will be deployed through multiple dissemination channels, appropriate form sets should be developed, one for each dissemination channel.

Since the two parts will be developed independently, there is a need to integrate them, by establishing links between the elements of the semantic part and the elements of the layout part. This procedure is covered in 2.3.2.3.6. The development of both parts should adhere to the results produced by the service process model creation phase, so these parts will be consistent with one another. Any inconsistencies between the semantic part and the layout part (such as a reference from the layout portion to a TSE or KU that does not exist) will be detected at the integration phase, and users will be advised on the actions that need to be taken to resolve the inconsistencies.

2.3.2.3.2TSEs

Transaction service elements will be the basic building blocks for transaction services. TSEs will be mainly defined by domain experts, and their work will be complemented by IT staff, who will code the IT related portions, and by service workers, who may contribute by adding knowledge units that will serve as help items for the end-users of the transaction service. TSEs will be defined by

assigning values to properties, through an appropriate set of forms (or a series of web pages, if a web interface is implemented).

Some properties of a TSE (which might be contributed by different roles) are mandatory. These may be defined in any order, after the TSE is created. The SmartGov platform will not impose any restrictions on the definition order. The only requirement is that at service instantiation time, all mandatory attributes must have been properly defined. In any other case, errors will be reported by the integrator (discussed in a subsequent paragraph).

2.3.2.3 TSE groups

With layout definition being developed outside the SmartGov platform, a TSE group defines the following:

- a set of TSEs that appear together within services
- Repetition information, indicating whether only one instance or multiple instances of the member TSEs is required. For groups allowing multiple instances, the member TSEs actually form a *table row*, which is repeated as many times as needed, and may be used to model "detail" sections, e.g. the items that are included in an order along with their prices, the customers of an enterprise together with the net value and the tax due for the transactions conducted with each one etc. The repetition information may indicate the initial, minimum and maximum number of instances and the step for adding new rows in the group.
- Validation checks that must hold among the elements of this set
- Knowledge units that apply to the set of TSEs, rather than to individual elements (e.g. for a TSE group representing a citizen's identification data, a KU containing the law that states which information is considered as "required identification data" may be defined)

It is worth noting that a single TSE may participate in more than one TSE group, thus the relationship between TSEs and TSE groups is of cardinality "many-to-many". For instance, the identification number of a citizen may appear in the TSE group "Personal Identification Data" and in the TSE group "Page footer", which can be placed on the bottom of a page to provide an immediate reference to the service context. TSE groups may not be nested, i.e. a TSE group may only contain individual TSEs, not TSE groups. This restriction leads to a more comprehensible and easy-to-manage framework for SmartGov platform users to work in, while it does not downgrade the platform functionality since (a) the same result may be obtained by adding the individual TSEs belonging to the source

group to the target group and (b) the cases that such a functionality will be needed will be –if exist at all- rare.

TSE groups with no repetition requirements are not an indispensable element of the SmartGov platform; they are provided for convenience purposes, since the working team will be able to package in a single entity all the necessary information for TSEs that usually appear together. Determination of whether a set of TSEs should be packed in a group with no repetition specification should follow some “rules of thumb”, such as “if some TSEs will be frequently used together, it will be beneficial if they were grouped together once and used thereafter as a single entity”.

2.3.2.3.4 Validation checks

Validation checks may apply to individual TSEs, TSE groups, forms or services. Validation checks pertinent to specific TSEs will mainly check the data type (integer, string, date etc) and the value range of the data entered. These validation checks may be considered as properties of the relevant TSEs.

Validation checks applying to TSE groups, forms or services will mainly check if a certain relationship between different TSEs holds. The TSEs referenced in the validation check should all appear in the object within which the validation check is defined; for example, a validation check defined at TSE group level should only reference TSEs participating in the TSE group.

Validation checks may be entered either via a graphical interface or in textual form, using a validation rules language that should be defined. In both cases, the definition of complex validation checks will be carried out by IT staff, rather than domain experts. IT staff should be allowed to code validation checks directly in the language used by the service delivery platform (e.g. PHP, Java etc.) if this is found to be convenient, or if the coding language/environment provided is not expressive enough to implement the desired functionality.

For the purpose of determining how much functionality is needed by the coding language/environment, we have computed some statistics on the validation checks that apply to the Greek tax return forms and VAT declaration forms. These statistics have shown that 82% of the total checks fall into one of the categories (1-4) described below, leaving an 18% in category 5.

1. *Value range check.* The value of a field is required to be between a specific lower and upper bound.
2. *A Requires B.* If a value is entered in field *A* then a value must be entered in field *B*. For example, if the Married indication is checked, the Wife’s Name field must be also filled in.

3. *A Precludes B*. If a value is entered in field *A* then field *B* should be left blank. For instance, if the user fills in the field profits from trade business, the field losses from trade business should not be filled in, since an enterprise may not have simultaneously profits and losses from the same activity.
4. *A cmp B * c*, where *A* and *B* are form fields, *cmp* is a relational operator ($=, \neq, >, \geq, <, \leq$) and *c* is a constant value. This validation check category allows for modeling of arithmetic constraints on form fields such as the expenses declared for transports must be less than or equal to the total expenses (in this case, $c = 1$), or pre-paid taxes may not be more than 45% of the total income ($c = 0.45$). Note that a TSE may contain the result of a computation on other TSEs, e.g. the sum of a column table or the product of two TSEs, thus this construct is highly expressive.
5. *Custom check*. This category of validation checks is used to model any constraint that does not fall in groups (1–4).

Based on these results we propose that the development environment should provide the means to express any check in categories (1-4), and allow the coding of checks in category 5 in any format appropriate for direct embedding in the code that will be generated for the service delivery environment.

2.3.2.3.5KUs

Every entity within the SmartGov platform may be associated with any number of knowledge units. These knowledge units may pertain to procedures in the SmartGov platform (e.g. how a TSE is created), to specific items (e.g. for a TSE representing the passport number, an associated KU may describe the format of the passport number or contain the relevant legislation), or constitute help for the end users of the transaction service (e.g. KUs with descriptions, examples etc). Every member of the working group may contribute to the population of the KU repository depending on their position and skills; for example, domain experts are the most suitable ones for providing legislation-related KUs, whereas service workers, having experience from interacting with citizens, are the most likely to provide comprehensive lists of dos and don'ts to serve as help items.

While KUs may be developed independently of other SmartGov entities and linked to them using an appropriate management tool, it is expected that in most cases the working team will be willing to perform KU creation and linking while creating or reviewing SmartGov entities. For instance, when creating the TSE representing the passport number, the domain expert may create the KUs containing the relevant legislation or link to them; when a service worker later reviews the TSE,

he/she may create KUs with examples for the service end-users. Therefore, the process of creating/reviewing (editing) a SmartGov entity should provide facilities for creating/linking KUs.

2.3.2.3.6 Links to back-end systems

Links to back-end systems should be established for the purposes of both retrieving data from and storing data to either organisational or third party systems. Links to back-end systems are defined mainly by the organisation's IT staff using the SmartGov agent technology. The IT staff should also provide the appropriate SEPs that will be plugged in the Information Interchange Gateway coupled with the organisational or third party system, and will actually implement the request.

2.3.2.3.7 Managerial statistics (Managers and Domain experts)

In this stage managers and/or domain experts define the statistics that they need to view, in order to assess the service operation.

- Number of submissions for a service
- Number of accesses to the service from a particular platform (e.g. desktop vs. handheld, browser, delivery channel).
- Number of saved sessions that were not finally submitted
- Number of submissions that were rejected due to errors
- Number of submissions that involved warnings
- Number of times a form was requested
- Number of times a specific check failed
- Number of times a specific TSE was used
- Number of times a specific knowledge base item was accessed.
- Time taken to complete a specific form

The SmartGov platform should offer a set of pre-defined statistics, pertaining to various SmartGov objects (transaction services, forms, KUs, TSEs etc). The managers and domain experts should be able to enable the collection of any of these statistics during service operation, for subsequent viewing and analysis. The definition of ad-hoc statistics should be also supported, by providing a set of guidelines through which the IT staff of the organization may insert appropriate code that will collect and store the relevant data.

2.3.2.3.8 IT-related statistics.

In this stage IT staff define statistics that are of interest to them, for the purposes of monitoring the behavior and optimizing the performance of the SmartGov platform. These statistics may include (the list is not exhaustive):

- Time taken to serve a specific request
- Time taken to communicate with a back-end system
- Profiling of the pending actions queue size
- Number of communication failures with a specific IT system

For a more complete picture of the platform behaviour, the IT-related statistics supported by the SmartGov platform should be examined in combination with the statistics that may be gathered by the other software modules within the platform, such as the operating system, web server, database server etc.

2.3.2.4 Integration

Once all necessary elements for a transaction service have been defined, the integration phase will arrange for performing a synthesis of these elements into an operational instance of the transaction service. In more detail, the integration step will perform the following actions:

1. It will access the service definition, extracting from it the links to the forms that implement the service, the validation checks pertaining to the service as a whole and the associated KUs.
2. It will retrieve the form definitions and the definitions of the TSEs appearing on each form, together with the associated validation checks and KUs. If a TSE group has been placed on a form, all TSEs belonging to the group will be retrieved, together along with their descriptions, KUs and validation checks. KUs and validation checks pertaining to the TSE as a whole will also be retrieved.
3. It will load the information regarding the statistics that need to be collected.

Once this information is available to the integrator, the service instantiation task may proceed. The integrator module will generate a page for each form defined within the service, using the form layout specification. Forms belonging to the same service will be suitably linked, based on form sequence information specified for the service; "submit" buttons will also be placed on the forms that have been designated to provide such functionality. At this stage, the completeness of references to TSEs should be verified: each TSE declared to participate in a form, should be linked with an element of the form layout. If this is not the case, the SmartGov platform user should be informed of the discrepancy, in order to amend the situation.

Validation checks defined at TSE level and form level will be used to generate code that will validate user input. This code may be executed:

1. *At the front-end* (client-device side), if the service designers have designated that this is desirable and if the client device supports active features. Regarding the timing of the execution, these checks may be performed either when the user changes a field value (usual case when the validation check pertains to the field data type or the field value range), or when the user leaves the page (typically when the validation check involves multiple fields).
2. *At the server-side*. All input should be always validated at the server-side, since in a distributed environment clients should be considered untrustworthy, and thus the system may not rest on the perception that all client-side checks have been properly executed. Server-side checks may be run when the user leaves a page or when a final submission is made, depending on the timing specified by the service designers.

The integrator will also generate code for the validation checks defined at service level. These will be executed on the server-side upon the final submission, since in general they involve TSEs appearing in different forms, which inhibits execution at the front-end upon form change (it is not guaranteed that all involved values will have been provided).

Finally code will be generated to arrange for the communication with third-party systems through the SmartGov agent. This communication will be mainly performed when the user invokes a service, in order to retrieve values for TSEs that need to appear pre-filled in with values obtained from registries or databases.

Knowledge units that are associated with TSEs, TSE groups, forms and the transaction service and that have been designated as "help items for end-users" should be appropriately linked to the forms. The integrator should arrange for the proper generation of help pages from KUs (possibly translation of XML to HTML or WML through XSLT templates) and embedding of the hyperlinks to the appropriate anchors.

Statistics definitions will also be translated to pieces of code that will arrange for collection and storage of relevant statistics. For example, if the sum of the values filled in a specific form element has been requested to be computed, the integrator will generate code that will add the value of each submitted form to a database element; if the number of submissions should be counted, code will be generated for adding up one to a specific database element upon submission.

Once the final pages and the associated programs have been generated, the files produced may be installed on a restricted access server for testing and evaluation purposes, or on a public access service for full service deployment.

2.3.2.5 Testing, Evaluation, Deployment, Operation and Maintenance, Feedback and Improvement

These phases are not performed through the SmartGov platform; however the SmartGov platform may provide input for some phases, such as the evaluation or operation and maintenance phases, through statistics, error reporting, or issuing notifications to the service administrators when changes that affect the service operation occur.

2.3.2.6 Link Establishment Between Form Layout and Form Semantics

Since the two parts of a form (semantic and layout) are developed independently, an integration step is required, in order to establish links between the elements of the semantic part and the elements of the layout part.

As stated in section 2.3.2.3.1, the semantic part of a form defines what information is entered in the form, the validation checks that apply to the form and the knowledge units that will be presented to the user. This information may be represented by an XML document, such as the one depicted in Figure 3.

```
<?xml version="1.0" encoding="ISO-885901"?>
<form>
  <formId>personalDataVAT</formId>
  <name>Personal Data for VAT Declaration</name>
  <includedTSE>VAT_TSE_surname</includedTSE>
  <includedTSE>VAT_TSE_name</includedTSE>
  <includedTSE>VAT_TSE_vat_id</includedTSE>
  <linkedKUNode>KU_essential_identification</linkedKUNode>
  <linkedKUNode>KU_vat_id_example</linkedKUNode>
  <workGroup>VAT task force</workgroup>
  <formLayout>http://devel-srv/e-svc/VAT/form1</formLayout>
</form>
```

Figure 3 – XML description of a form

The form description contains references to various SmartGov platform entities (TSEs, KUs), which are described as XML documents in the SmartGov platform database; example documents are presented in Figure 4. The form description also contains one reference to the form layout, which is stored in an XHTML file. This file has been developed outside the SmartGov platform using a suitable XHTML editor. A possible rendering of the layout in a browser window is illustrated in Figure 5, while the associated XHTML code is presented in Figure 6.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<instantiatedTSE>
  <instantiatedTSEId>VAT_TSE_surname</instantiatedTSEId>
  <instanceOf>SMARTGOV_generic_string</instanceOf>
  <name>Tax payer's surname in VAT form</name>
  <maxLength>32</maxLength>
  <linkedKUNode>KU_essential_identification</linkedKUNode>
  <linkedKUNode>KU_VAT_name_example</linkedKUNode>
  <isVisible>TRUE</isVisible>
  <isReadOnly>TRUE</isReadOnly>
  <isMandatory>TRUE</isMandatory>
  <workGroup>VAT task force</workgroup>
</instantiatedTSE>

<?xml version="1.0" encoding="ISO-8859-1"?>
<instantiatedTSE>
  <instantiatedTSEId>VAT_TSE_name</instantiatedTSEId>
  <instanceOf>SMARTGOV_generic_string</instanceOf>
  <name>Tax payer's name in VAT form</name>
  <maxLength>32</maxLength>
  <linkedKUNode>KU_essential_identification</linkedKUNode>
  <linkedKUNode>KU_VAT_name_example</linkedKUNode>
  <isVisible>TRUE</isVisible>
  <isReadOnly>TRUE</isReadOnly>
  <isMandatory>TRUE</isMandatory>
  <workGroup>VAT task force</workgroup>
</instantiatedTSE>

```

Figure 4 – XML description of two TSEs

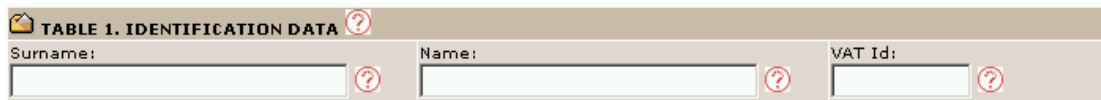


Figure 5 – Rendering of identification data form in a browser

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>VAT statement - Personal data</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link rel="stylesheet" href="html/colors.css" type="text/css">
</head>
<body bgcolor="#FFFFFF" text="#000000" background="html/images/back.gif">
<table width=700 border=0 align="center">
<tr>
<td>
<form name="form1" method="post" action="">
<table border="0" cellspacing="1" cellpadding="1" width="700">
<tr>
<td class="colordark" colspan="3">
<b>TABLE 1. IDENTIFICATION DATA</b> </td>
</tr>
<tr>
<td class="colormedium" valign="middle" width="260">
Surname:<br> <input name="surname" size="30" maxlen="30">
</td>
<td class="colormedium" valign="middle" width="260">Name:<br> <input name="name" size="30" maxlen="30">
</td>
<td class="colormedium" valign="middle" width="180">VAT Id:<br> <input name="vat_id" size="9" maxlen="9">
</td>
</tr>
</table>
</form>
</td>
</tr>
</table>
</body>
</html>

```

Figure 6 – XHTML code for identification data form

The issue at hand is establishing the links between (a) the input fields on the form with the corresponding TSEs and (b) the help anchors on the form (marked with the question mark icon) with the corresponding knowledge units. Two options for achieving this goal are described in the following paragraphs.

1. The form layout designers are aware of the identities of the transaction service elements and the knowledge units, as assigned and maintained by the SmartGov platform and use these identities to tag the relevant items on the form. For example, the form layout designers could use the id of the TSEs as the *name* property of input fields, or the id of the KU as the *alt* property of images. In this case, the processor can easily map each layout item to the respective entity of the SmartGov platform. The drawback of this approach is that form layout designers must deal with the identities of the SmartGov platform (which will most probably be machine-generated and thus counter-intuitive), and this will hinder the development and maintenance process. Moreover, this will impose the restriction that form layout can only proceed *after* the relevant entities (TSEs and KUs) in the SmartGov platform have been defined.
2. The form layout designers proceed independently of any activities within the SmartGov platform; the layout designers are not aware of any entities of the SmartGov platform, but assign arbitrary values to attributes of the form elements. (This is the case of Figure 6). The links between the layout elements and the SmartGov platform entities may be established in an intuitive manner using the following procedure:
 - a. A processor reads the definitions of TSEs and KUs pertaining to the form in question (appropriate information exists within the SmartGov "form" entity), extracting identifications and descriptions (or any other human-readable form suitable for identifying the SmartGov platform object). Note that TSEs may be included in the form directly or indirectly, through their participation in a TSE group that has been placed on the form.
 - b. The processor scans the form layout (HTML, WML, etc), locating tags that correspond to input fields (e.g. <input>, <select> etc. for HTML forms) to drop down-lists, whose selections correspond to the TSE elements. The processor adds a "submit mappings" button for notifying the SmartGov platform of the desired mappings, and when this button is pressed, the mappings are sent to a program that performs this exact function (storing of mappings). The

translated HTML code is illustrated in Figure 7 and will be presented to the SmartGov platform user as depicted in Figure 8. Note that in the option parts of the drop-down lists the TSE ids are used as values, whereas the textual portion of the option (the label presented to the user) is the TSE description, as found in the XML description of the TSE.


```

<head>
<title>VAT statement - Personal data</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link rel="stylesheet" href="html/colors.css" type="text/css">
</head>
<body bgcolor="#FFFFFF" text="#000000" background="html/images/back.gif">
<table width=700 border=0 align="center">
<tr>
<td>
<td>
<form name="form1" method="post" action="/bin/layoutMapper.jsp">
<input type="hidden" name="SmartGovFormName" value="personal_data_vat">
<table border="0" cellspacing="1" cellpadding="1" width="700">
<tr>
<td class="colordark" colspan="3">
<b>TABLE 1. IDENTIFICATION DATA</b></td>
</tr>
<tr>
<td class="colormedium" valign="middle" width="260">
Surname:<br>
<select name="surname">
<option value="null">Please select a TSE</option>
<option value="VAT_TSE_surname">Tax payer's surname in VAT form</option>
<option value="VAT_TSE_name">Tax payer's name in VAT form</option>
<option value="VAT_TSE_vat_id">Tax payer's VAT id in VAT form</option>
</select>

</td>
<td class="colormedium" valign="middle" width="260">Name:<br>
<select name="name">
<option value="null">Please select a TSE</option>
<option value="VAT_TSE_surname">Tax payer's surname in VAT form</option>
<option value="VAT_TSE_name">Tax payer's name in VAT form</option>
<option value="VAT_TSE_vat_id">Tax payer's VAT id in VAT form</option>

```

```
        </select>
        
    </td>
    <td class="colormedium" valign="middle" width="180">VAT Id: <br>
        <select name="vatid">
            <option value="null">Please select a TSE</option>
            <option value="VAT_TSE_surname">Tax payer's surname in VAT form</option>
            <option value="VAT_TSE_name">Tax payer's name in VAT form</option>
            <option value="VAT_TSE_vat_id">Tax payer's VAT id in VAT form</option>
        </select>
        
    </td>
</tr>
</table>
<hr width="100%">
<div align="right"><input type="submit" value="Submit mappings"></div>
</form>
</td>
</tr>
</table>
</body>
</html>
```

Figure 7 – Transformed XHTML code for mapping form elements to TSEs

Figure 8 – Rendered form for the purposes of mapping

By comparing Figure 5 and Figure 8 it can be seen that the XHTML page may be slightly distorted (e.g. question mark icons appear on a separate line from the associated input elements), since the size of the input element on the original form may be different than the size required by the drop-down list widget on the transformed form. However, the analogy to the original form is quite clear.

The final step is mapping of the help anchors to knowledge units. A similar approach to the mapping of the form elements to TSEs may be used. The processor must be aware of the text used in the form layout description as a help anchor (in the previous example the text is a reference to the question mark image, expressed in the HTML jargon as ``). These help anchors will be replaced by drop down lists, whose items will be a list of the KUs pertinent to the form (this includes all KUs related to the service, the form and any TSE groups or TSEs placed on the form). In this case, the layout is bound to be more distorted, since help anchors are usually quite small, compared to the textual descriptions of KUs. However, alternative approaches may be used, such as displaying the form in its original layout and popping up a new window with the selections when the user clicks on the anchor.

At the end of the link establishment process, the SmartGov platform should verify that all TSEs declared in the form's semantic part to participate in the form have counterparts in the layout part. If some TSEs remain unmapped, notification messages should be issued to the SmartGov platform user.

When all form elements have been mapped to TSEs and KUs, as appropriate, the integrator may proceed as described in section 2.3.2.4.

This approach may be followed for WML documents; in this case, the processor's job of scanning the layout code for user input tags is much simpler, since the only such tags in WML are `input`, `optgroup`, `option` and `select`. Considering the limited expressiveness of WML and the restricted hardware it is usually processed on, only simple services should be considered for deployment through WML channels.

2.3.2.7 Maintaining the associations between visual elements and SmartGov entities

The procedure described in section 2.3.2.6 establishes links between the visual elements appearing on the form and the SmartGov platform entities (TSEs and KUs). It is important for the purposes of service maintainance that these links are "remembered" by the SmartGov platform so that, in the event of some service modification, the domain experts and/or KU staff involved in the maintenance process will only have to add/delete/modify the mappings that have changed, as compared to the last version, and not perform the whole process anew.

The maintenance of the association between XHTML form elements (input areas, drop down lists etc.) and TSEs is straightforward, since (a) the TSEs have a unique identity within the SmartGov environment and (b) the XHTML form elements have a name, which identifies them uniquely within the scope of the form. Thus, it suffices to record within the SmartGov platform the association between each TSE identifier and the corresponding XHTML form element name.

For KU form anchors, however, there is no guarantee that the visual element will have a unique identity, since this anchor may be a piece of text, an image, or any other allowable XHTML entity; moreover, the same XHTML entity may be used as a placeholder for multiple KU anchors. In order to tackle this issue, the SmartGov platform may insert into the XHTML file unique identities for the KU anchors. The association between these unique identities and the KU identifiers will then be saved within the SmartGov platform to maintain the association between KUs and their anchors. The identities will be inserted as XHTML comments within the XHTML file, thus no modification to the visual layout or the form semantics will result.

For example, consider the XHTML code of Figure 6, in which KU anchors are denoted through the XHTML code fragment ``, the mapping procedure will generate for each KU anchor a unique id (the uniqueness constraint must hold in the scope of the form) and will expand each occurrence of the code fragment with an XHTML code fragment of the form `<!-- KUREF id="uid" --><!-- /KUREF -->` where `uid` is the unique id generated for this specific instance of a KU anchor.

2.3.3 The Front-End Navigation Diagrams

The main goal of the SmartGov platform is to provide a smart environment to develop and deliver transactional services. In order to do that, an integrated and unified environment that allows service development and knowledge creation and

retrieval is needed. The SmartGov Front-End is an environment that is going to support several phases of the Transaction Service Life-cycle, and some other issues as well:

- Work Group management
- SmartGov User management and Service Roles management stored in an RDBMS as explained in 2.8.13.1
- Taxonomy editing tool.
- Transaction services components development (TSs, Forms, TSEs, TSE Groups, Validation Checks, definition of statistics).
- Knowledge acquisition (KU editing tool).
- Retrieval facilities (via Taxonomies).
- Knowledge and TS life-cycle tool.

The following figures give a general view of navigation diagrams regarding the SmartGov Front-End:

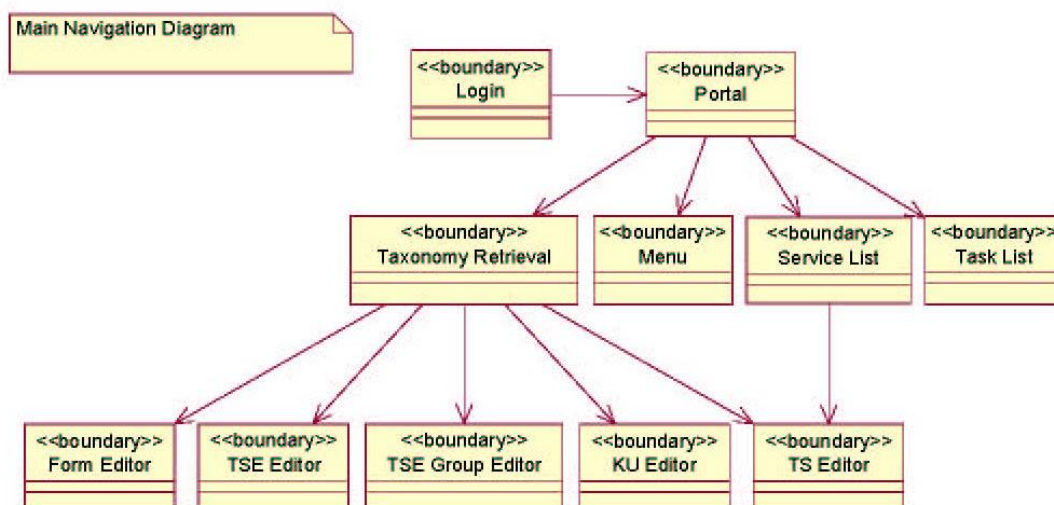


Figure 9 – Main SmartGov Front-End Navigation Diagram

As shown in the previous figure, after a valid SmartGov user is logged in the system, a Portal page is shown. Depending on the user profile (their role in the system, as Manager, Domain Expert, IT Staff, and Service Workers) different information will be shown. From this portal there are several ways to act:

- Selecting menu options from the Menu
- Performing task from the Task List
- Selecting a TS to edit from the Service List
- Retrieving a SmartGov object from the Taxonomy Retrieval tool.

- The Work Group selector will allow to change the current Work Group in wich the logged user is working in. This change drives to update the list of available services (related with the work group).
- From the List of Services, the TS editor can be accessed
- From the Taxonomy Retrieval tool, navigation to all SmartGov categorized objects is provide.
- From the Task List, access to the life-cycle of TSs and KUs is available.

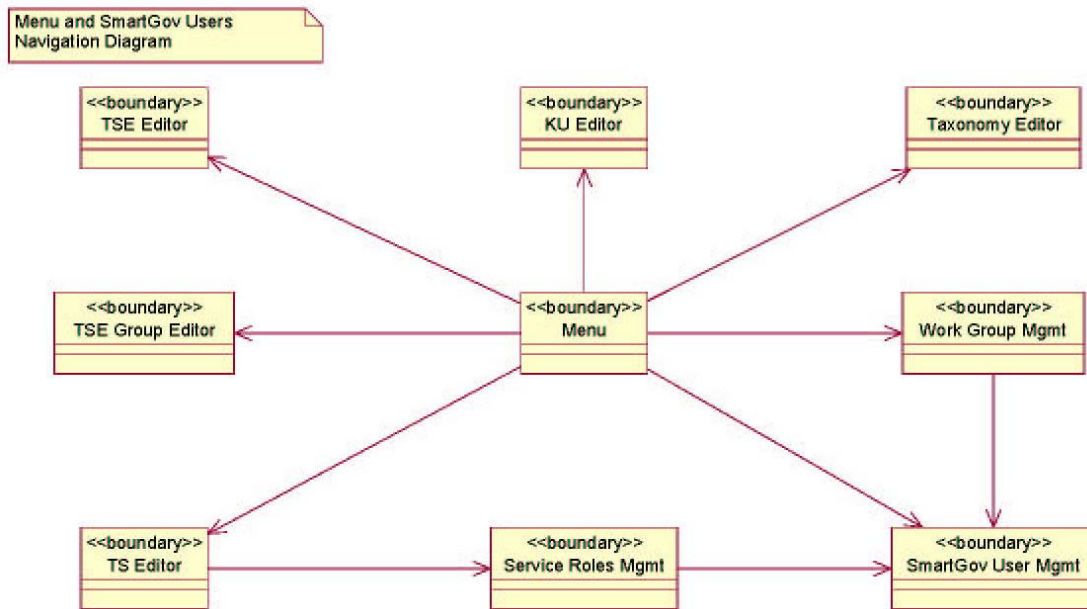


Figure 11 – SmartGov Menu Front-End Navigation Diagram

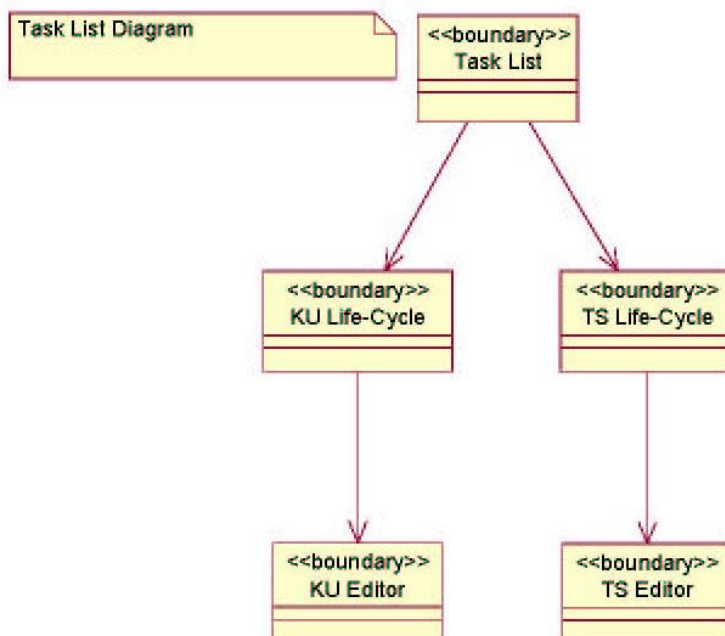


Figure 12 – SmartGov Task List Front-End Navigation Diagram

Several options will be available from the Portal Menu as it is shown in Figure 12. These options are role sensitive. Thus depending on the role of the user (Manager, Domain Expert, IT Staff, or Service Workers) some options will be unavailable. For instance, a Service Worker will not be able to edit the Work Group, User, and Service Roles Management, or modify the taxonomy using the Taxonomy Editor.

The Task List (Figure 12) gives access to the KU and TS approval cycle. Before making the knowledge or the Transaction Service ready to be deployed to the production environment, KUs and TSs must complete its life-cycle as is shown in Figure 13 (KU Life-Cycle). After this cycle the quality, accuracy, and content of the KU or TS should be sufficient to be deployed.

The approval cycle will be performed following the work group hierarchy, according to the specified permits (modifiers, reviewers, approvers) stored in the SmartGov User-Roles system. The life-cycle business rules will be stored in a configuration XML file, because the rules of approval could change in the future.

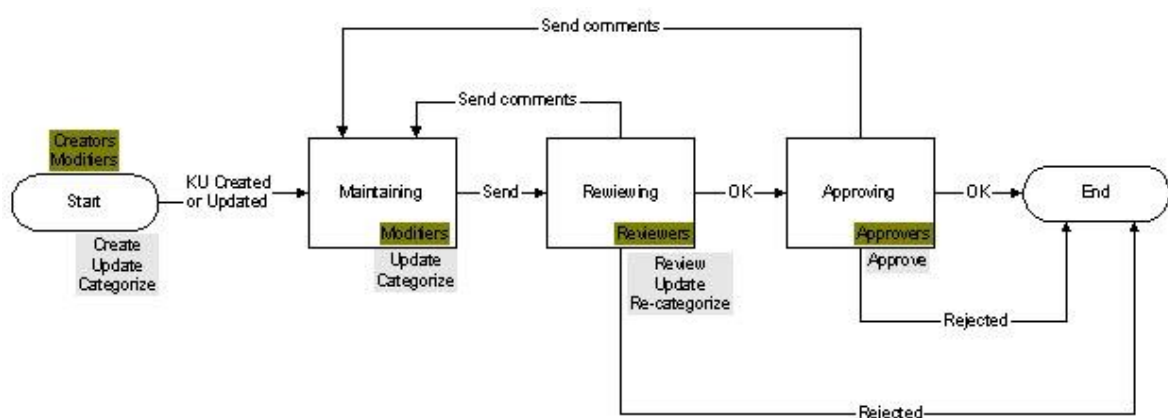


Figure 13 – KU Life-Cycle

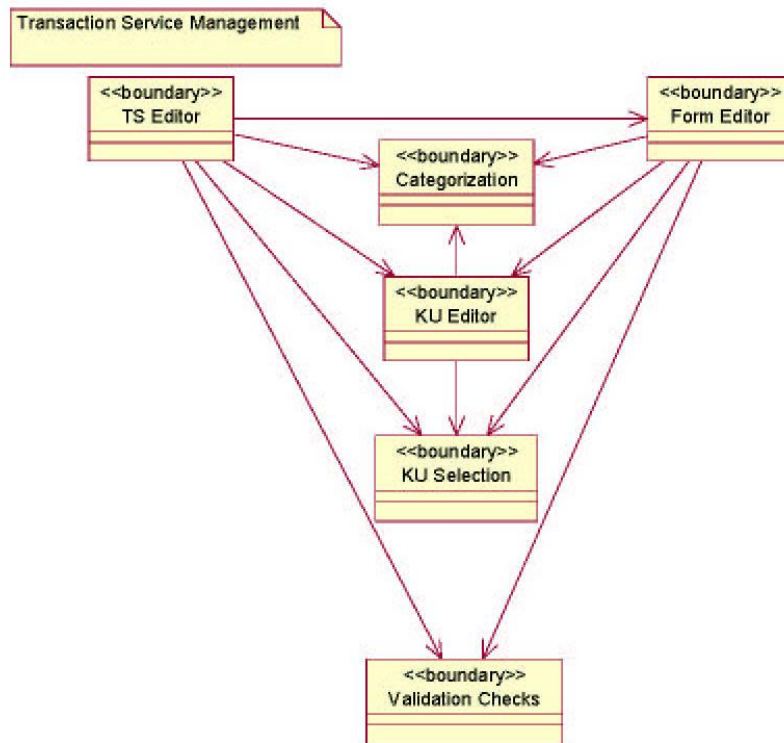


Figure 14 – SmartGov TS Editor Front-End Navigation Diagram

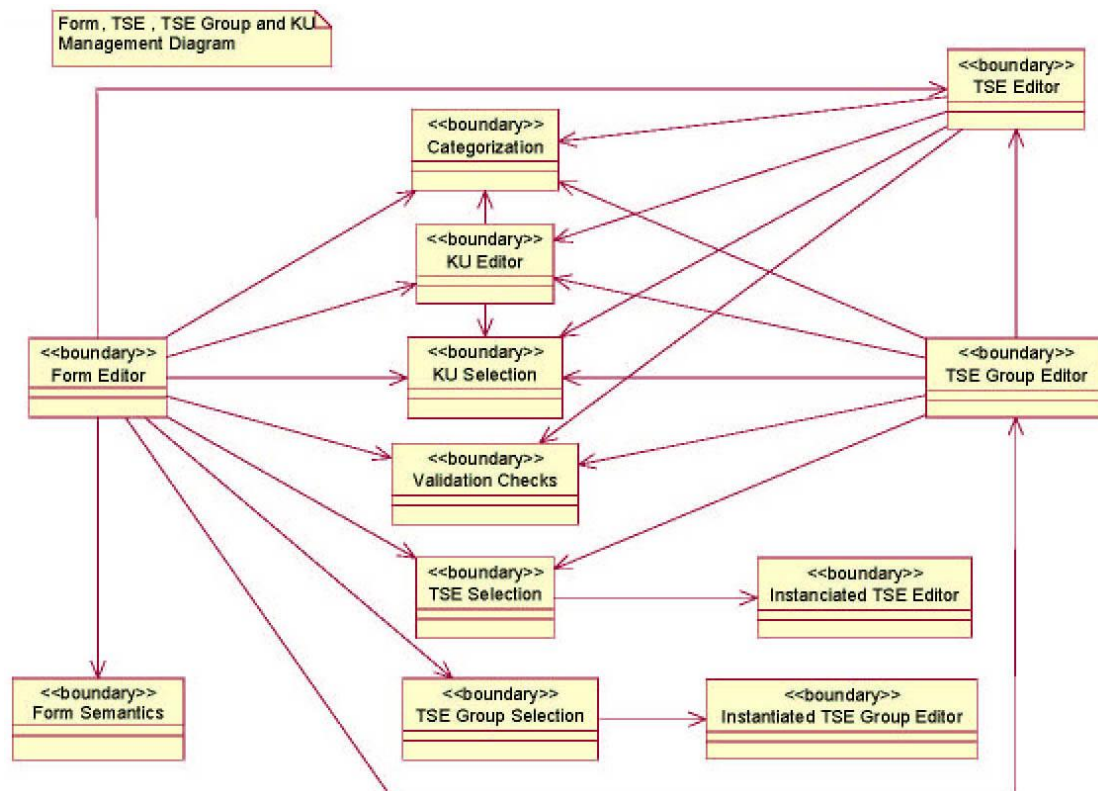


Figure 15 – SmartGov Form Editor Front-End Navigation Diagram

The two previous figures showed all the capabilities of the Transaction Service Editor. All links between pages are shown (except perhaps some connections between instantiated TSE and TSE Groups with elements like KUs or Validation Checks). In the following figures a closer view of this editor is shown.

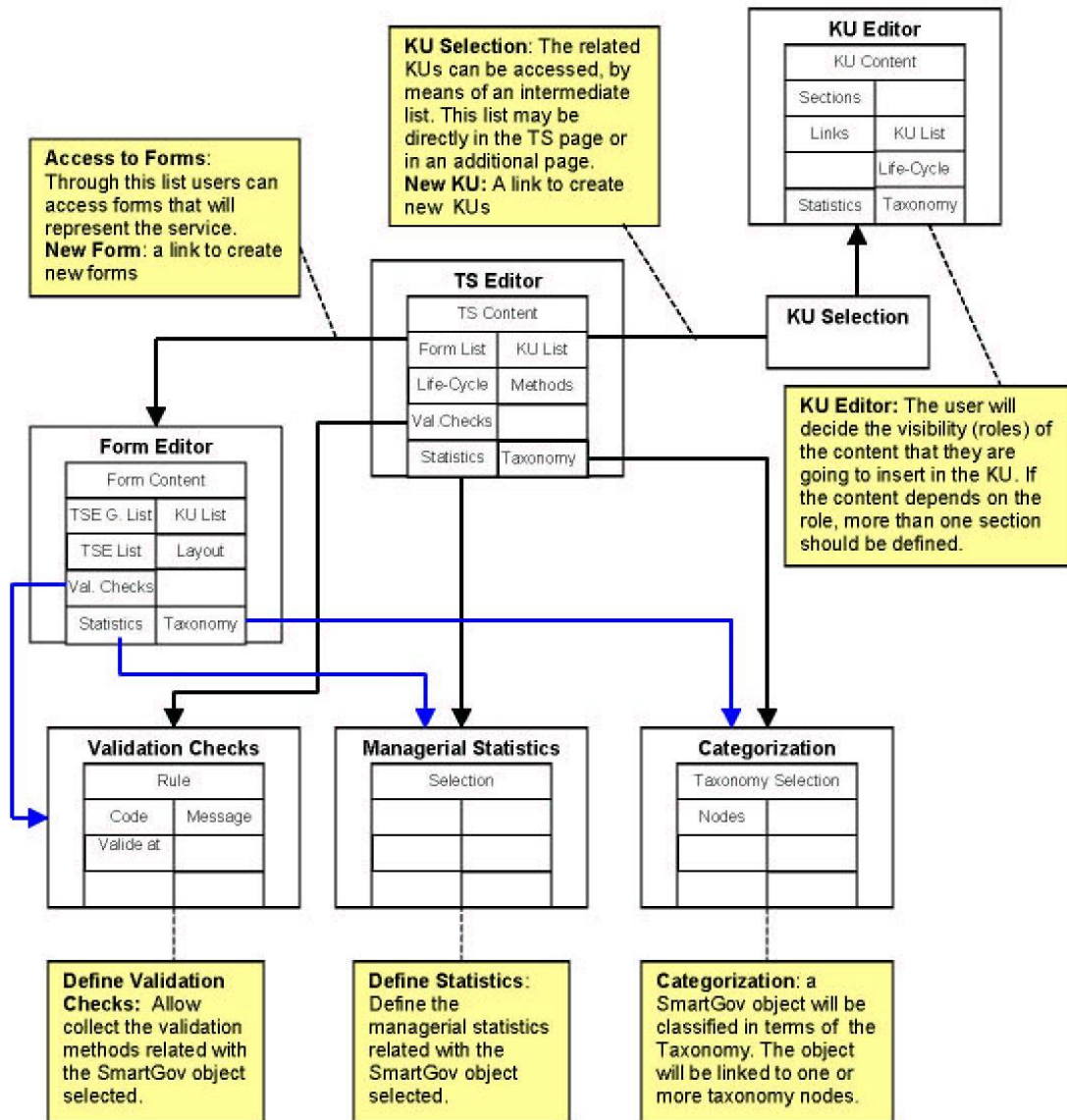


Figure 16 – TS Editor Diagram Explanation

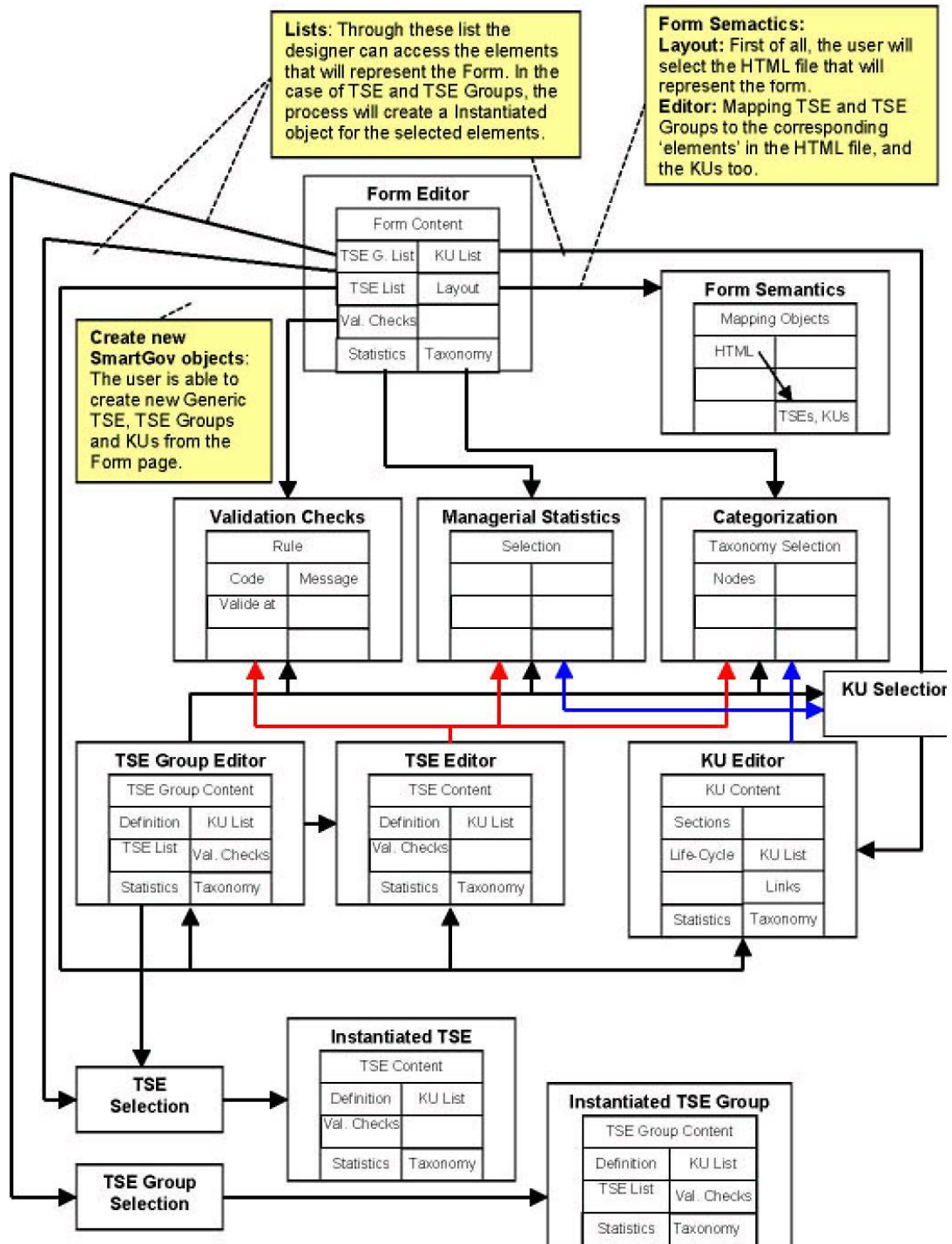


Figure 17 – Form Editor Diagram Explanation

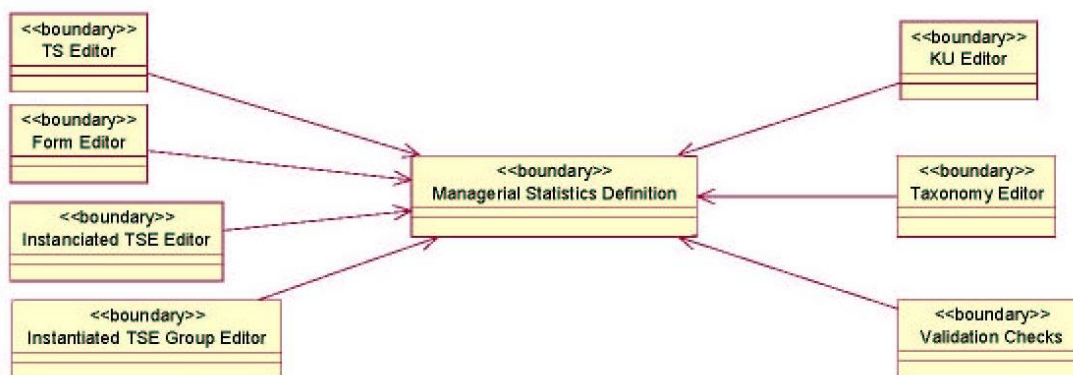


Figure 18 – SmartGov Managerial Statistics Front-End Navigation Diagram

The definition of managerial statistics is shown in the previous figure. Notice that this definition implies just a formal gathering of statistics criteria. The integrator will be responsible for implementing the collection of the statistics based on the descriptions defined here.

2.4 Logical view

2.4.1 Introduction

This section describes the architecturally significant parts of the design model, and their decomposition into subsystems and packages. For some packages, their decomposition into classes and class utilities are also given. Some architecturally significant classes are introduced here and their responsibilities, as well as a few very important relationships, operations, and attributes are given. The first architecturally significant package is the Jakarta STRUTS MVC Framework [Struts]. Due to its extensive usage and important role within many of the design elements of the platform Struts is presented in section 2.4.2.1. After that the XML repository, which will store all the XML documents needed by the platform, is presented. The XML repository will be accessed by the SmartGov front-end tool, presented right after, which will provide a user-friendly interface allowing the management of the aforementioned XML documents. After that the agents allowing inter-communication with installed IT legacy systems are presented. Finally the presentation of the integrator follows. This component will be responsible for the generation of the final e-forms service accessible by the citizens.

2.4.2 Architecturally Significant Design Packages

2.4.2.1 The STRUTS Framework

The Struts framework provides the invisible underpinnings that a web application needs in order to be extensible, re-usable and eliminates several dangers faced in the development of such applications. Struts helps in creating an extensible development environment for an application, based on published standards and proven design patterns. Its basic principle is the support of the Model-View-Controller design pattern and that is why in the next section this pattern is introduced.

2.4.2.1.1 *The Model View Controller Framework*

The Model-View-Controller architecture is a widely-used architectural approach for interactive applications. It divides functionality among objects involved in maintaining and presenting data to minimize the degree of coupling between the objects. The architecture maps traditional application tasks--input, processing, and output--to the graphical user interaction model. They also map into the domain of multitier Web-based enterprise applications.

The MVC architecture divides applications into three layers--model, view, and controller--and decouples their respective responsibilities. Each layer handles specific tasks and has specific responsibilities to the other areas.

- A model represents business data and business logic or operations that govern access and modification of this business data. Often the model serves as a software approximation to real-world functionality. The model notifies views when it changes and provides the ability for the view to query the model about its state. It also provides the ability for the controller to access application functionality encapsulated by the model.
- A view renders the contents of a model. It accesses data from the model and specifies how that data should be presented. It updates data presentation when the model changes. A view also forwards user input to a controller.
- A controller defines application behavior. It dispatches user requests and selects views for presentation. It interprets user inputs and maps them into actions to be performed by the model. In a stand-alone GUI client, user inputs include button clicks and menu selections. In a Web application, they are HTTP GET and POST requests to the Web tier. A controller selects the next view to display based on the user interactions

and the outcome of the model operations. An application typically has one controller for each set of related functionality. Some applications use a separate controller for each client type, because view interaction and selection often vary between client types.

Figure 19 depicts the relationships between the model, view, and controller layers of an MVC application.

Separating responsibilities among model, view, and controller objects reduces code duplication and makes applications easier to maintain. It also makes handling data easier, whether adding new data sources or changing data presentation, because business logic is kept separate from data. It is easier to support new client types, because it is not necessary to change the business logic with the addition of each new type of client.

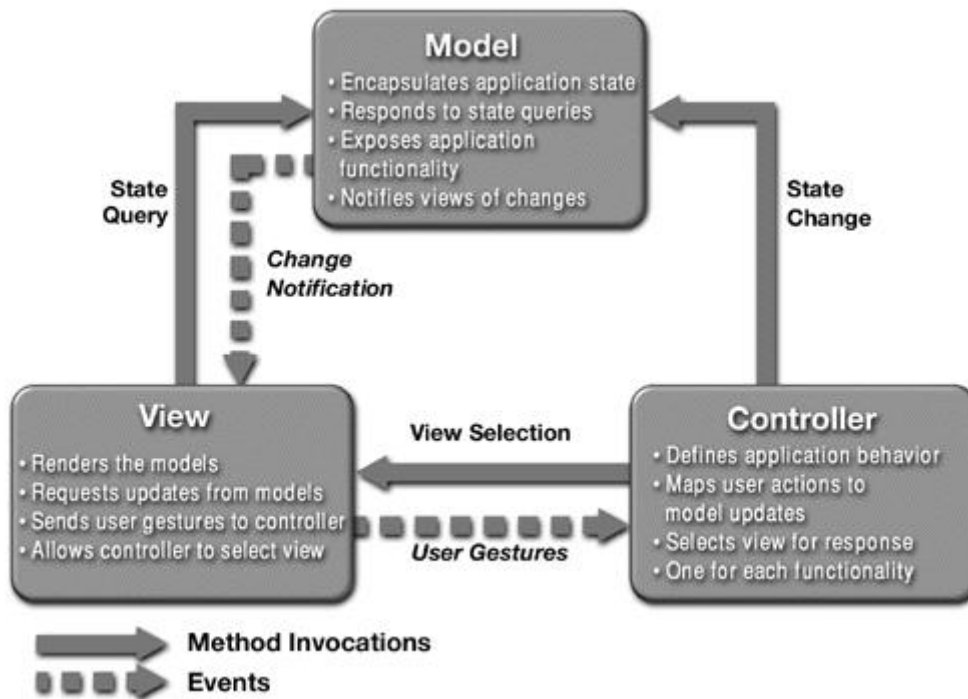


Figure 19. The Model-View-Controller Architecture

2.4.2.1.2 An Introduction to Struts

The core of the Struts framework is a flexible control layer based on standard technologies like Java Servlets, JavaBeans, ResourceBundle, [J2EE] and Extensible Markup Language (XML).

Struts encourages application architectures based on the Model 2 approach, a variation of the classic Model-View-Controller (MVC) design paradigm. Struts provides its own Controller component and integrates with other technologies to

provide the Model and the View. For the Model, Struts can interact with any standard data access technology, including Enterprise Java Beans, JDBC, and Object Relational Bridge. For the View, Struts works well with JavaServer Pages, Velocity Templates, XSLT, and other presentation systems.

Figure 20 illustrates the Struts architecture.

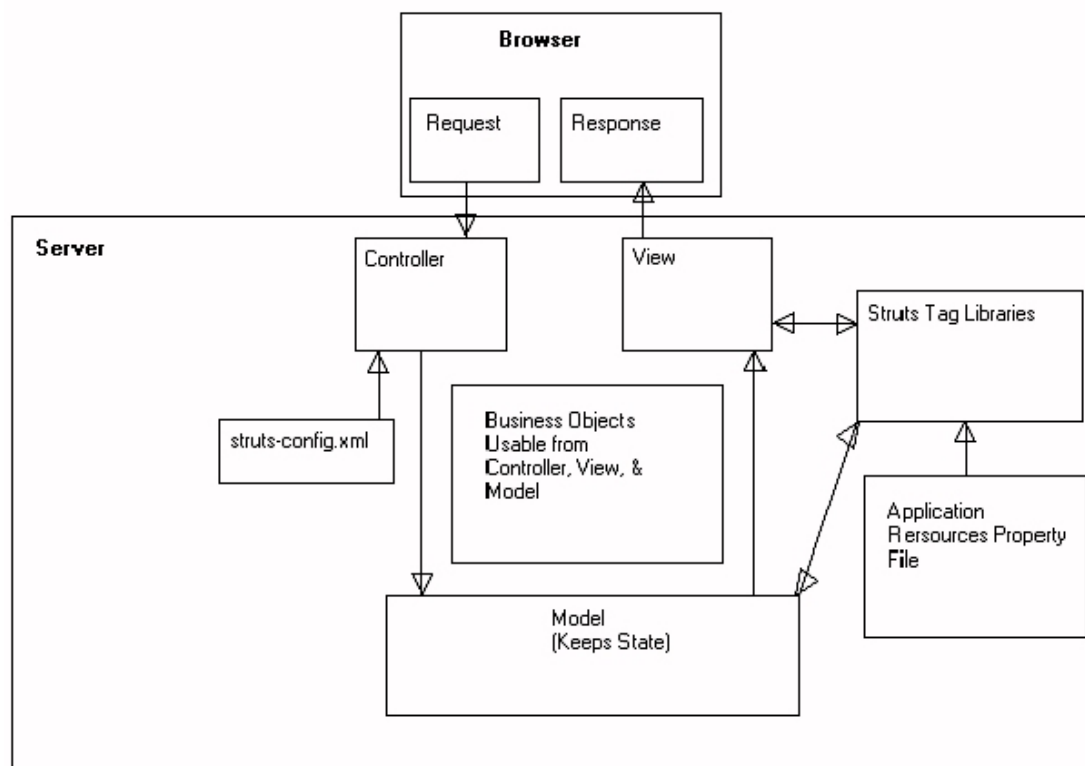


Figure 20 The Struts Architecture

There are three major components in Struts, the servlet controller (controller), the JSPs (View), and the application business logic (Model). The controller bundles and routes HTTP requests to other objects in the framework. It also parses the configuration file (Struts-config.xml), which contains action mappings that are used to turn HTTP requests into application actions.

On the other hand the model of the system is divided into concepts that keep the internal state of the system and actions that can change that state. These are represented by Java Beans. In the specific case that a Forms application is developed these beans are ActionForm Beans, which extend the ActionForm Class within the framework implementation. In conjunction to the aforementioned beans the business logic beans also exist. These encapsulate the functional logic of the application as method calls on JavaBeans designed for this purpose. For accessing relational Databases Struts allows the definition of datasources from

within its standard configuration file. A simple JDBC connection pool is also provided.

As far as the view part of the application, Struts provides the means for building internationalized applications. In fact it builds upon the Java platform using the following components:

- Locale - fundamental Java class that supports internationalization
- ResourceBundle - supports messages in multiple languages
- PropertyResourceBundle - standard implementation of ResourceBundle that allows you to define resources using the same "name=value" syntax used to initialize properties files
- MessageFormat - allows you to replace portions of a message string with arguments specified at run time
- MessageResources - lets you treat a set of resource bundles like a database, and allows you to request a particular message string for a particular Locale

The Struts framework also provides a number of Tag libraries to assist in coding JSPs efficiently. These are:

- HTML Tags: The tags in the Struts HTML library form a bridge between a JSP view and the other components of a Web application. Since a dynamic Web application often depends on gathering data from a user, input forms play an important role in the Struts framework. Consequently, the majority of the HTML tags involve HTML forms. Other important issues addressed by the Struts-HTML tags are messages, error messages, hyperlinking and internationalization.
- Bean Tags: The "struts-bean" tag library provides substantial enhancements to the basic capability provided by <jsp:useBean>.
- Logic Tags: The Logic tag library contains tags that are useful in managing conditional generation of output text, looping over object collections for repetitive generation of output text, and application flow management.
- Template Tags: The Template tag library contains three tags: put, get, and insert. Put tags put content into request scope, which is retrieved by a get tag in a different JSP page (the template). That template is included with the insert tag
- Custom Tags: Allow the definition of custom actions performed by the JSP that contains them

More information on the Struts Framework can be found at [Struts].

2.4.2.2 The XML Doc Repository

The XML doc repository is the component responsible for storing the XML documents mentioned previously in this document. It provides an Application Program(ming) Interface (API) to the SmartGov Front end tool. This API aims to facilitate creation, deletion, retrieval and alteration of the specified XML documents. The SmartGov Front-end is a web application addressed to the Domain Experts and Managers of the platform and basically constitutes the GUI (Graphical User Interface) of the repository. It also facilitates additional functionality, which is described in detail in section 2.4.2.3.

Following the API calls made by the SmartGov Front-end, the XML Doc Repository mechanism subsequently takes over the responsibility to handle with skill the XML documents in a transparent way, applying an inherent efficient storage mechanism based on indexes.

The overall architecture of the SmartGov XML doc Repository is shown in the following figure.

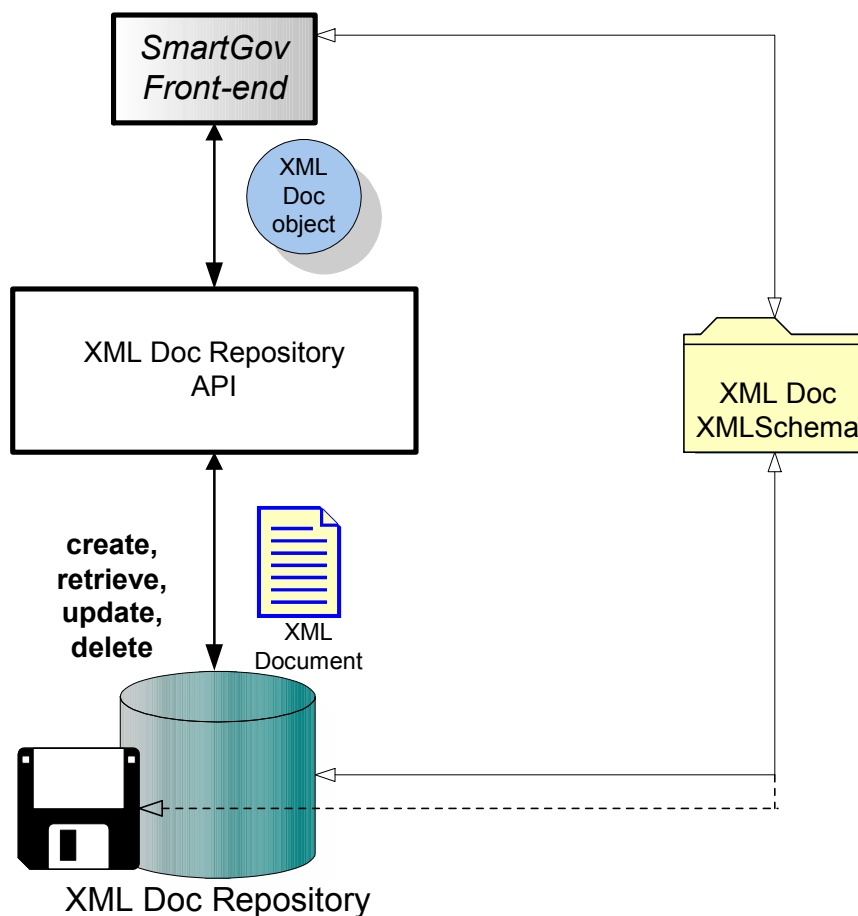


Figure 21. The XML Doc Repository Architecture

The SmartGov Front-end passes XML doc java objects to the XML Doc Repository API. The structure of the XML Doc objects is dependent on the specific XML doc structure that is each time created or retrieved and is generated by the Castor tool. In other words XML Doc objects are the representation of the XML documents in Java language. These objects comply with the object model generated by the castor framework. Castor is an open source data binding framework, which maps a Java object model to and from XML, according to a given XML schema. For further information about Castor, the reader should refer to the [Castor] After receiving an XML Doc object, the XML Doc Repository objects construct on the fly the respective XML document and perform the action requested by the front-end (insertion, deletion or modification).

In the following figure a class diagram of the XML Doc Repository API implementation is given.

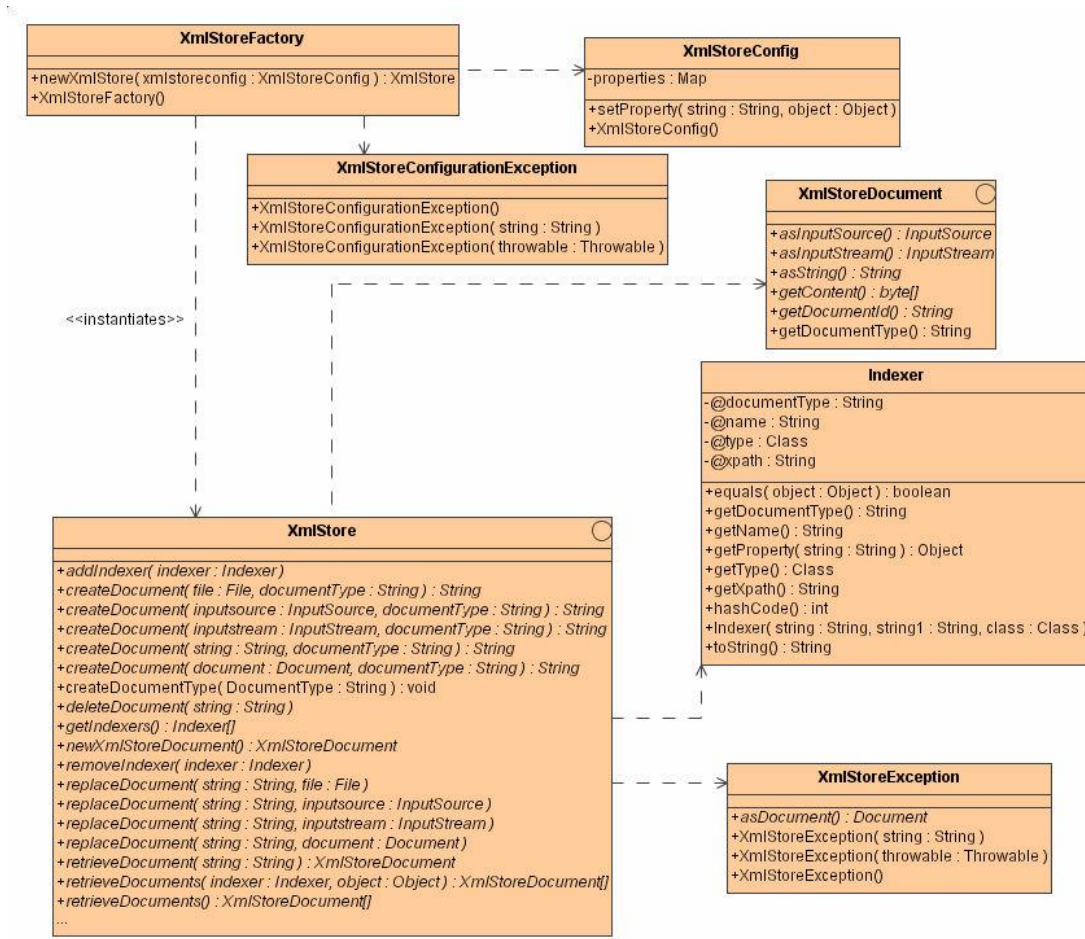


Figure 22 XML Doc Repository API class diagram

2.4.2.2.1 The XMLStoreFactory Class

The XMLStoreFactory class is a convenience class that provides access to implementations of the XMLStore Interface and configuration data through the class XMLStoreConfig. XMLStoreFactory instantiates an XMLStore object through the method newXMLStore(XMLStoreConfig).

2.4.2.2.2 The XMLStoreConfig Class

The XMLStoreConfig Object associated with the XMLStoreConfig Class is a generic object that can store whatever configuration data (String name, Object value) properties required by the underlying implementation of the XMLStore Interface. For example it may store configuration properties such as the full name of the class that implements the XMLStore Interface and in the case that the storage is done in a database XMLStoreConfig stores database configuration data such as the Class name of the JDBC Driver, the Class name of the DataSource and other configuration parameters.

2.4.2.2.3 The XMLStore Interface

The XMLStore public interface provides the API for persisting XML documents in a Database (RDBMS).

2.4.2.2.4 The Indexer Class

The Indexer class provides an index on the stored XML documents

2.4.2.2.5 The XMLStoreDocument Interface

Represents a document handled by an implementation of the XmlStore interface.

2.4.2.2.6 The XMLStoreException Class

The XMLStoreException class represents a general exception type that the XMLStore can throw.

2.4.2.2.7 The XMLStoreConfigurationException Class

The XMLStoreConfigurationException class extends the XMLStoreException class. When thrown, indicates inconsistency in the underlying XmlStore's configuration.

2.4.2.3 The SmartGov Front-End

The SmartGov Front-End, as it is stated in 2.4.2.2, is an integrated and unified environment that provides to the SmartGov users a light HTML interface that will allow service development and knowledge creation and retrieval. The following tools related with the service development will be supported in the SmartGov platform:

- Work Group management.
- SmartGov User management and Service Roles management.
- Taxonomy editing tool.
- Transaction services components development (TSs, Forms, TSEs, TSE Groups, Validation Checks, definition of statistics).
- Knowledge acquisition (KU editing tool).
- Retrieval facilities (via Taxonomies).
- Knowledge and TS life-cycle tool.

The logical architecture of the front-end is shown in Figure 23 and Figure 24:

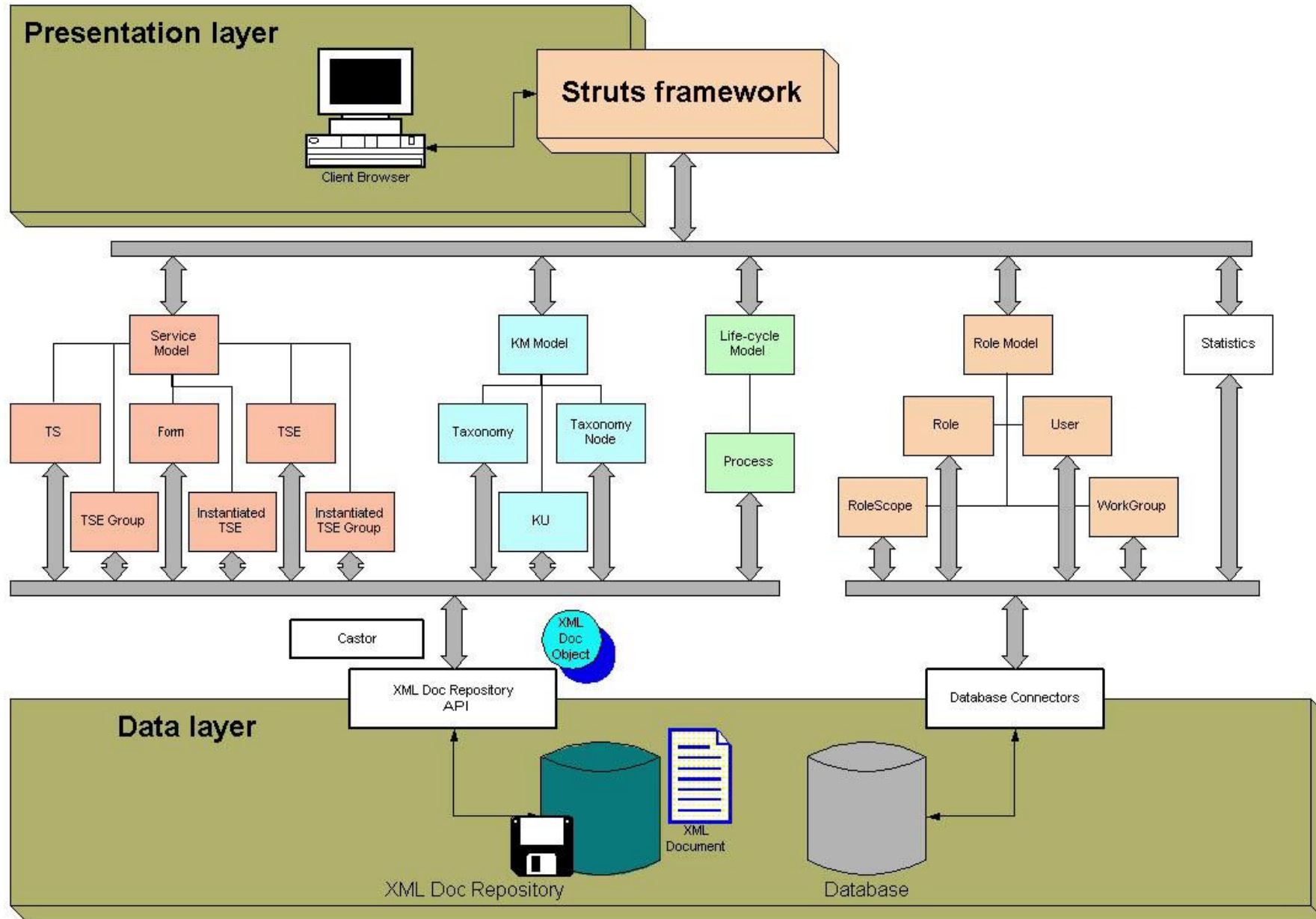


Figure 23 – SmartGov Front-End logical architecture components

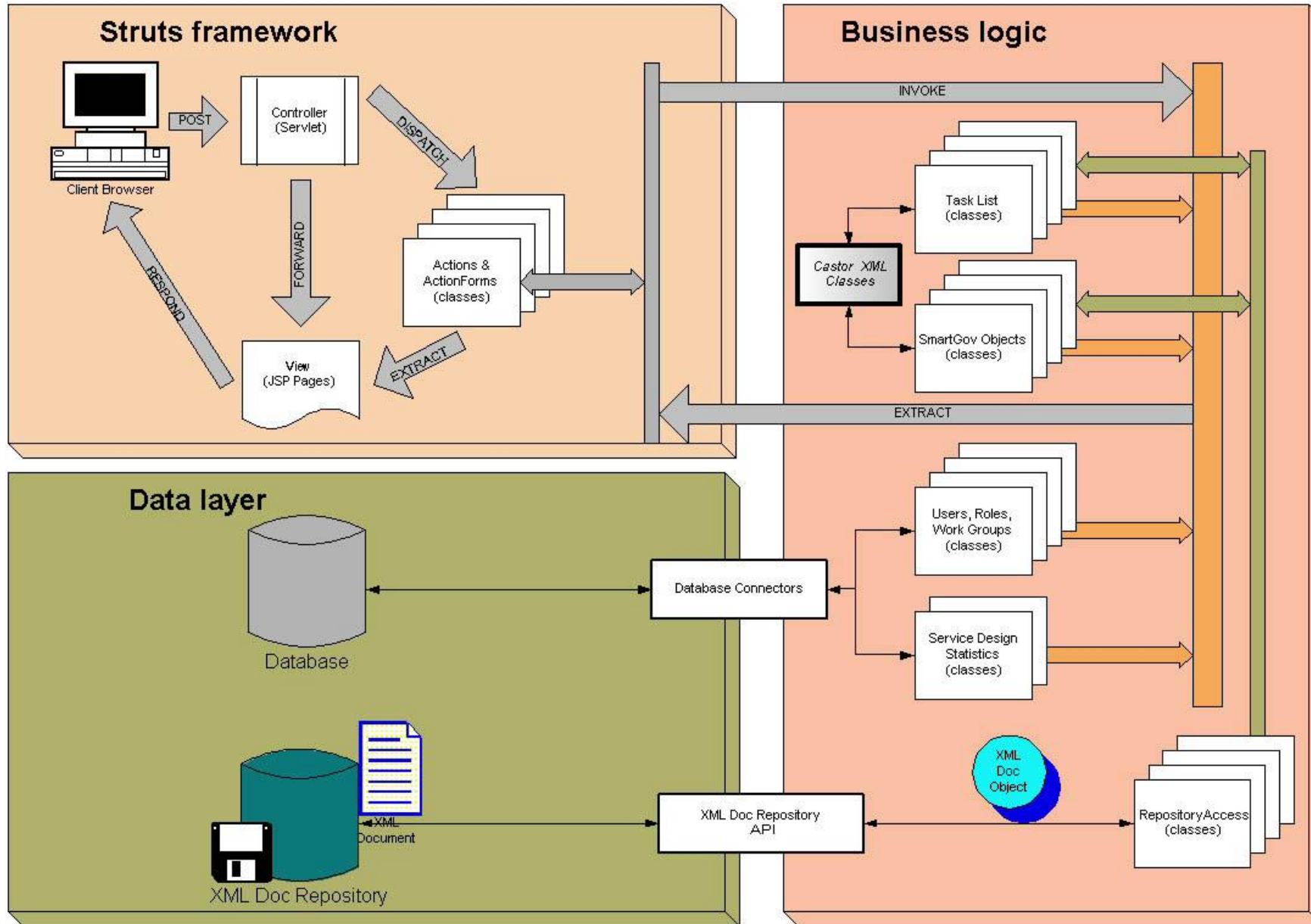


Figure 24 – SmartGov Front-End interaction between components architecture

2.4.2.3.1 Presentation Layer (Struts)

The SmartGov Front-end will be developed according to the MVC (Model-View-Controller) architecture, a standard approach for developing applications. This architecture will also be used in the Delivery Environment. Even the same MVC framework will be used in both environments: Struts, by the Jakarta Apache project (see [Struts])

Though the general lines about this architecture and framework are explained in the aforementioned paragraph, a deeper introduction to the framework will be carried out here.

2.4.2.3.1.1 Using Struts in the Front-End tool

Struts framework provides the central element in the MVC architecture, the controller, which is responsible for distributing requests and responses to the appropriate components. This controller can be customized with an XML file, configuring the actions that it is able to carry out, and the corresponding views to show the results.

Thus, developing a set of classes is required to add new actions to the application. These classes are:

- **Action class:** When the controller decides which action must be performed, it calls the perform method in the class that extends `org.apache.struts.action.Action`. In this method, the corresponding checks and calls to business logic classes will be done. Considering the return values from the business logic and the configuration established in the struts XML file, the next view to be shown to the user is decided.
- **ActionForm class:** This is a bean that encapsulates the access to the different fields in a Web form. Using Struts JSP Tags in collaboration with a class that extends `org.apache.struts.action.ActionForm` allows to automate the communication from the action to the form and vice versa. This class allows set/get the fields in the form with the corresponding methods, and adds a method to validate the modified / inserted data.

Therefore, for each action that has to be implemented in the front-end, we will require these two classes, besides the required business logic classes. Using this approach, we will keep our business logic completely isolated from the Web

environment, and it will be possible reusing it in the future in other non-Web environment (Desktop application, wap...).

2.4.2.3.1.2 Example of use

In order to show how Struts works in the SmartGov Front-End tool, an example, in this case Logging on the application, is explained.

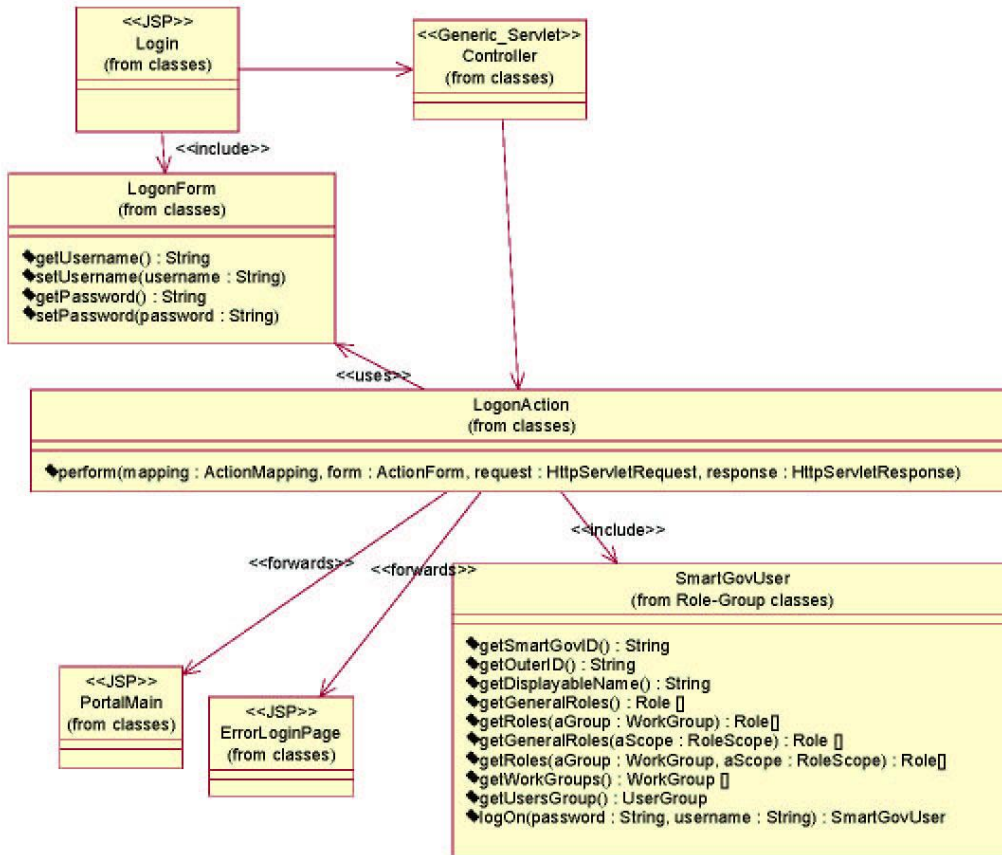


Figure 25 – Example of use of Struts in the Front-end tool

As it was aforementioned, two classes are required: LogonAction and LogonForm. It also requires a JSP page with a logon form including two fields: username and password. Once developed the JSP page, using Struts JSP tags specially developed to create forms, mapping these fields in the LogonForm class is needed, so the corresponding methods get/setUsername and get/setPassword must be added.

Now in the LogonAction class, in its perform method, the appropriated validation methods must be included and also the corresponding calls to the business logic

classes. For example, it must be checked that the user and the password fields are not empty, and, afterwards, a call to the User class (to check if the password is correct) is needed. Finally, according to the return from this call, the action class will direct the response to the Portal main page or to an error page.

It is important to point out that the Action class will contain client-related validations (in this example, checking that the fields are not empty) and that these validations may be implemented also in the Action Form class. However, the logic-related validations –those imposed by the business rules like, for example, that the password has to coincide with the database one, or even that the password must be at least six characters...- have to be included in business logic classes, because in this way these validations are applied always independently of the Presentation layer used (Web, wap, console application...)

A sequence diagram showing this approach is shown in Figure 26:

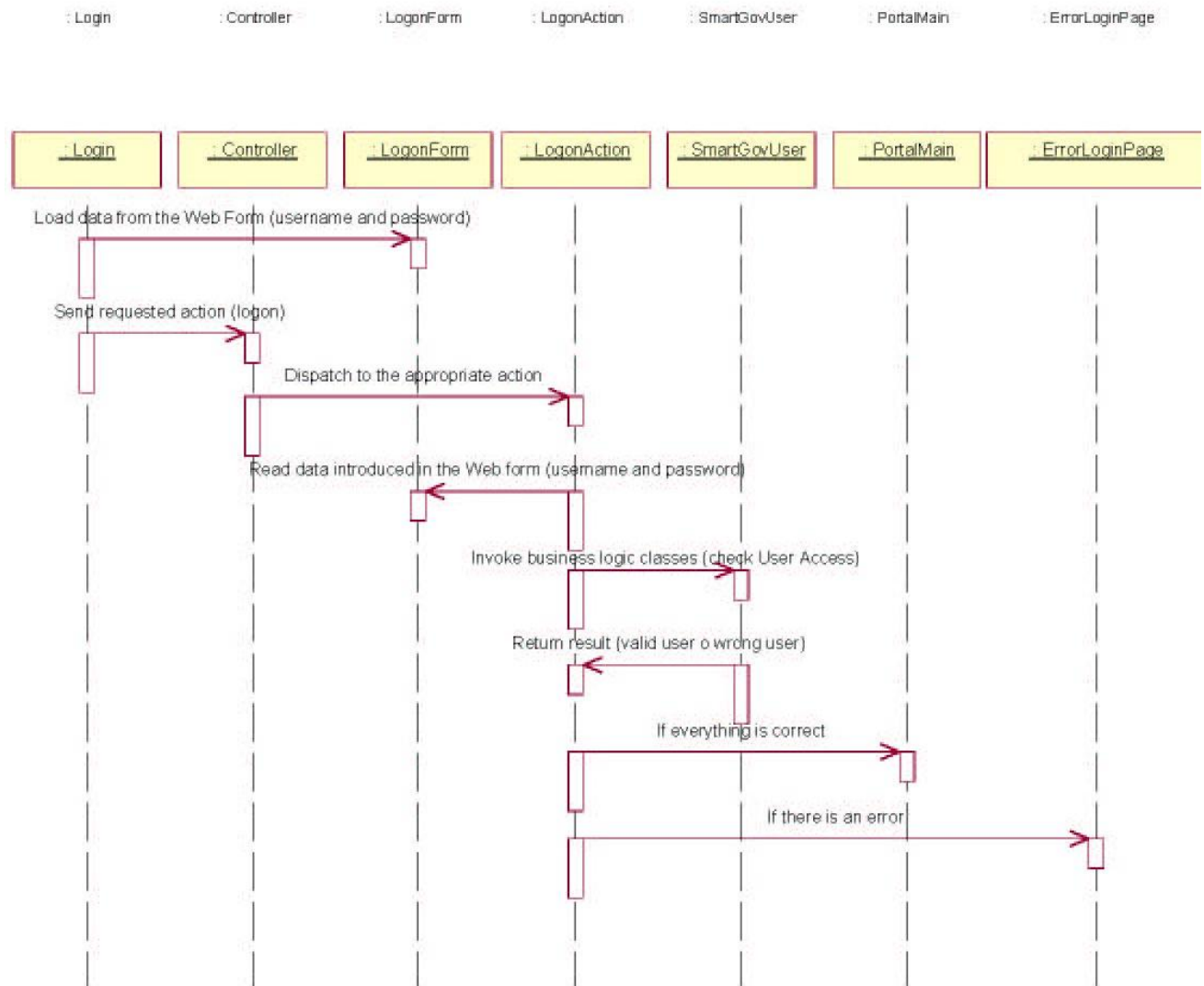


Figure 26 –Sequence diagram using Struts in the Front-end tool

2.4.2.3.2 Business Layer

2.4.2.3.2.1 SmartGov Service Design Model

The SmartGov front-end deals with the definition and design of several components of the electronic service that are going to take a concrete form within the SmartGov platform. These elements are the TSs, Forms, TSEs, TSE Groups, Instantiated TSE, and Instantiated TSE Groups, as it was aforementioned.

In the following figure a class diagram of the service design model regarding just the more important methods, and relationships between the service components is shown.

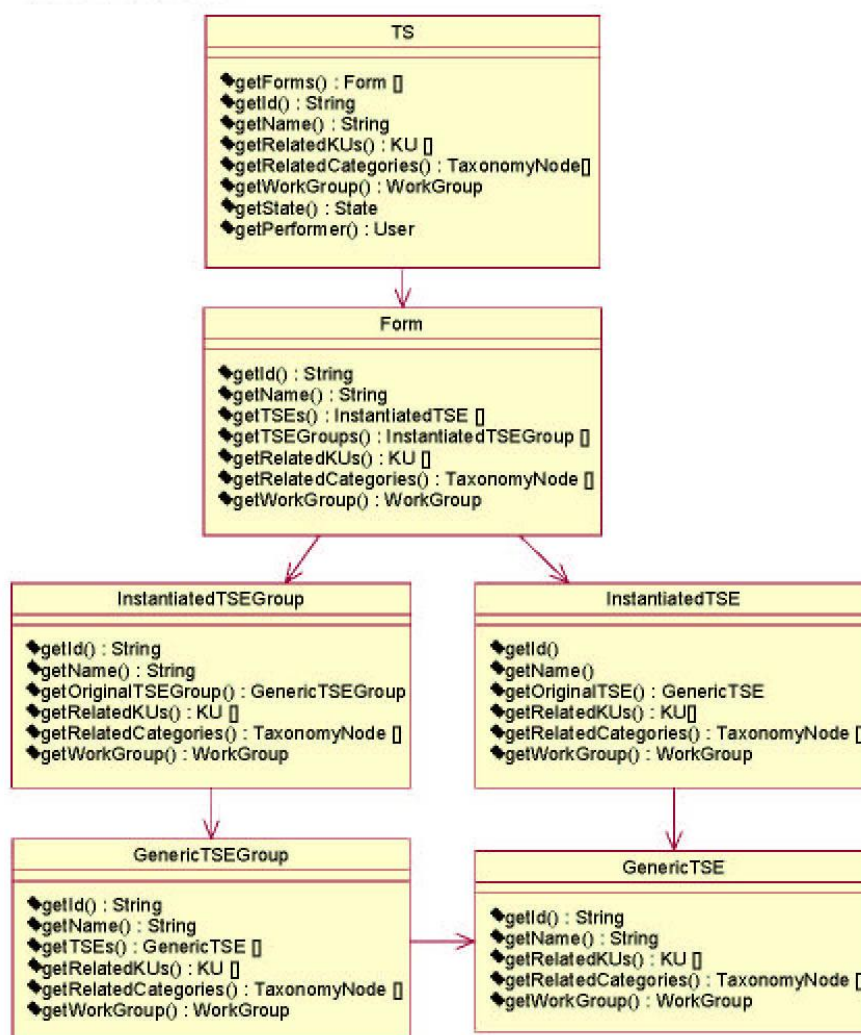


Figure 27 – SmartGov service design objects logical class diagram

2.4.2.3.2.2 SmartGov Knowledge Model

Knowledge related with e-services, or with the SmartGov platform, and SmartGov objects is going to be stored through Knowledge Units (KUs). Those KUs could be related with several SmartGov objects:

- Knowledge related with the SmartGov platform: those KUs will be a kind of system “help” that will be related with every SmartGov page, field or design object that needs this kind of help. They will be deployed with the SmartGov development environment as a part of the platform.
- Knowledge related with service objects: TS, Forms, TSE, TSE Groups or service KUs. This knowledge will be created by the users that are going to develop a service within the SmartGov platform. Some of these KUs can

be mapped of the help anchors to a form. These mapped KUs should be deployed with the service.

To categorize and recover knowledge and other SmartGov objects, a Taxonomy Editor will be developed. The SmartGov taxonomies will have a tree-like appearance. This editor will deal with several issues:

- Create, modify, and delete a taxonomy. Add and remove existing Nodes to the taxonomy.
- Create and delete Nodes.

The Taxonomy management should enforce some measures to assure the data integrity, like not deleting a node having objects related with it.

Other facilities related with taxonomies will be provided:

- Node selection tool to allow linking to SmartGov objects.
- Retrieval tool to choose which one among the SmartGov taxonomies the user wants to navigate.
- Retrieval tool to navigate through taxonomy nodes and allow retrieving of SmartGov objects.

In the following figures two classes diagrams of the KM model and the relationship with the service design model are given.

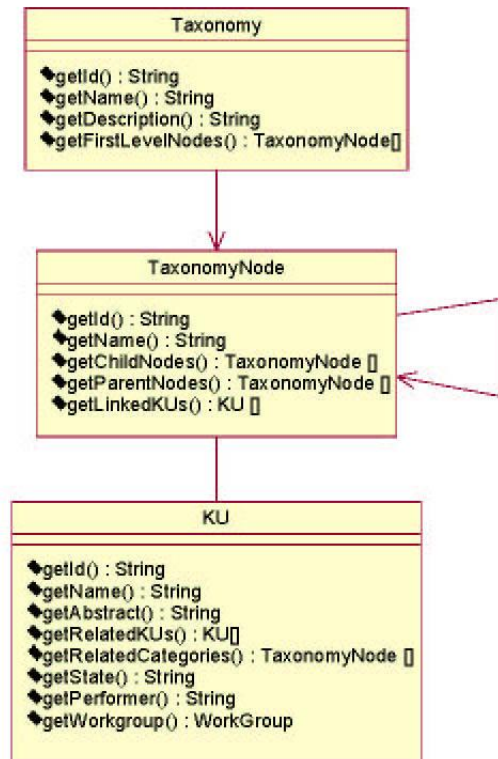


Figure 28 – SmartGov Knowledge Objects logical class diagram

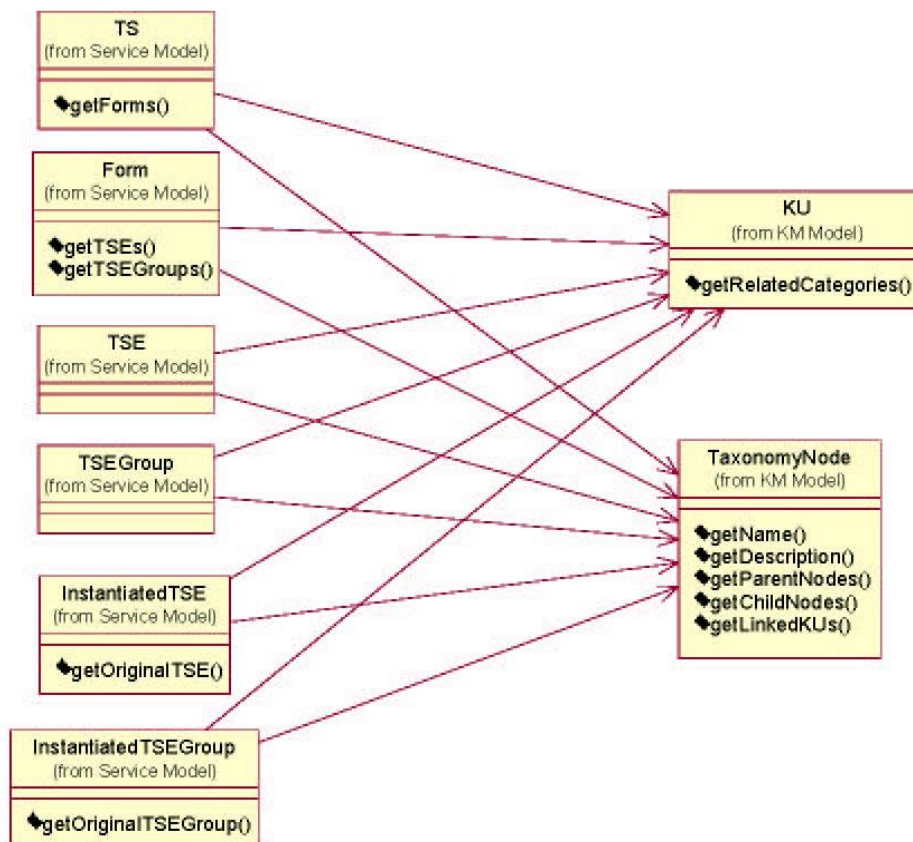


Figure 29 – SmartGov KM & Services relationships logical class diagram

2.4.2.3.2.3 SmartGov Life-cycle Model

To fulfil the required life-cycle for SmartGov objects (TSs and KUs), several workflow capabilities must be developed.

The life-cycle business rules will be stored in a configuration XML file, because the rules of approval could change in the future. These rules follow the next paradigm:

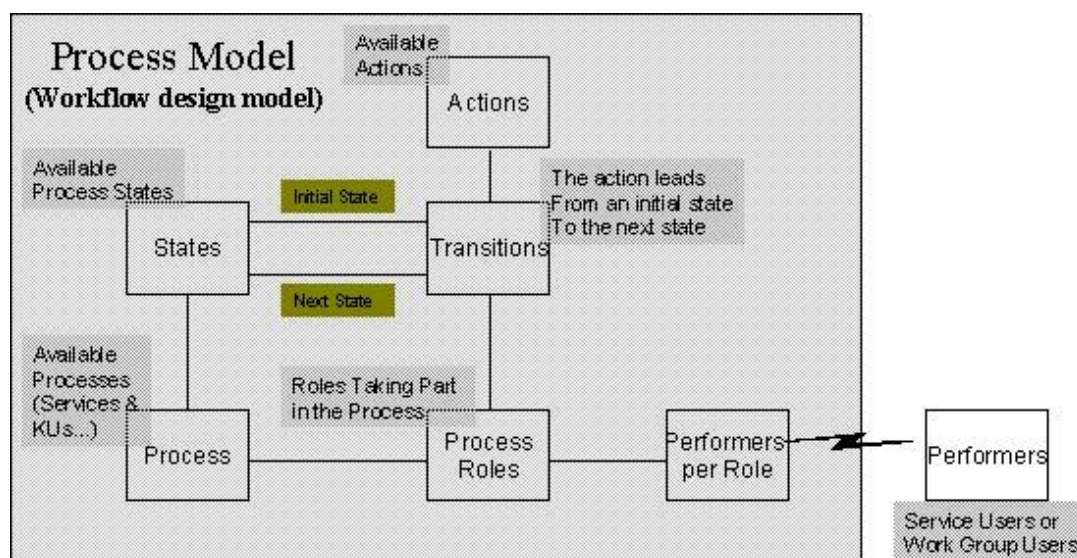


Figure 30 – SmartGov life-cycle rules

As it can be seen in Figure 30, there are two different Processes in SmartGov: TS, and KU life-cycle. Those processes will have defined States (Editing, Reviewing, Approving, Approved), and Process Roles (Editor, Reviewer, Approver). The changes between and Initial State to a Next State will be driven by Actions (Send to Review, Review, Approve). Those are the available Transitions, that could be performed by the users associated to the Process Roles. Users (Performers) and Roles should match with the users and roles defined in the RDBMS system (see 2.8.13.1).

From the SmartGov Portal, a Task List tool gives access to the KU and TS approval cycle. The task list is designed to retrieve and manage all tasks delivered to any kind of object, although in SmartGov just two objects (KUs and TSs) have a defined life-cycle. For this purpose a interface is defined to specify the characteristics that must be fulfilled by any object that needs to implement this functionality.

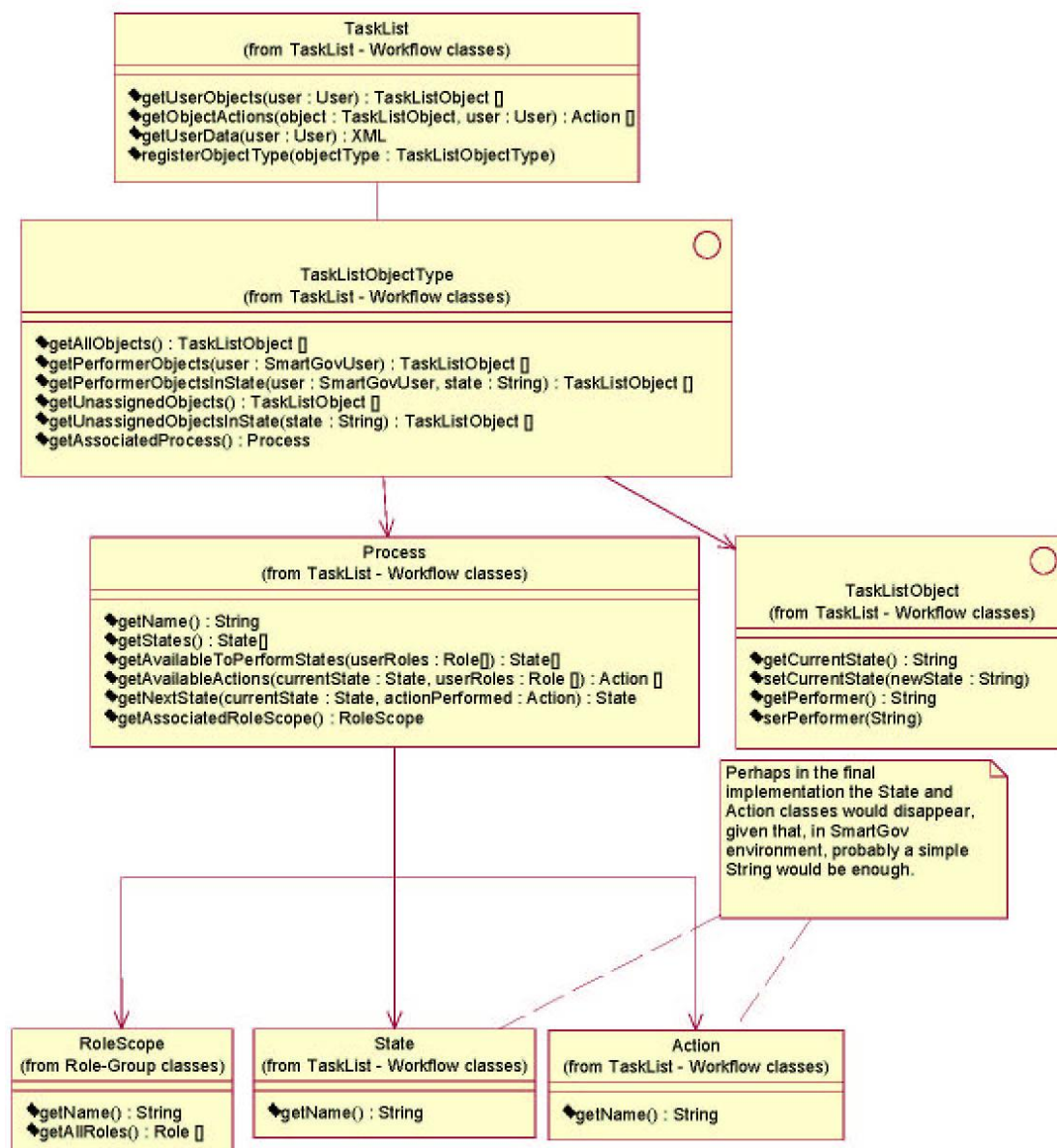


Figure 31 – SmartGov life-cycle logical class diagram

2.4.2.3.2.4 Users, Work Groups and Roles Model

SmartGov has to ensure a correct management of users and profiles to allow access to the service design environment to the users with SmartGov roles (Managers, Domain Experts, IT Staff, and Service Workers). Furthermore support for entity life-cycle and workflow capabilities have to be provided.

In the following figure a logical architecture diagram of the Role and User model is given.

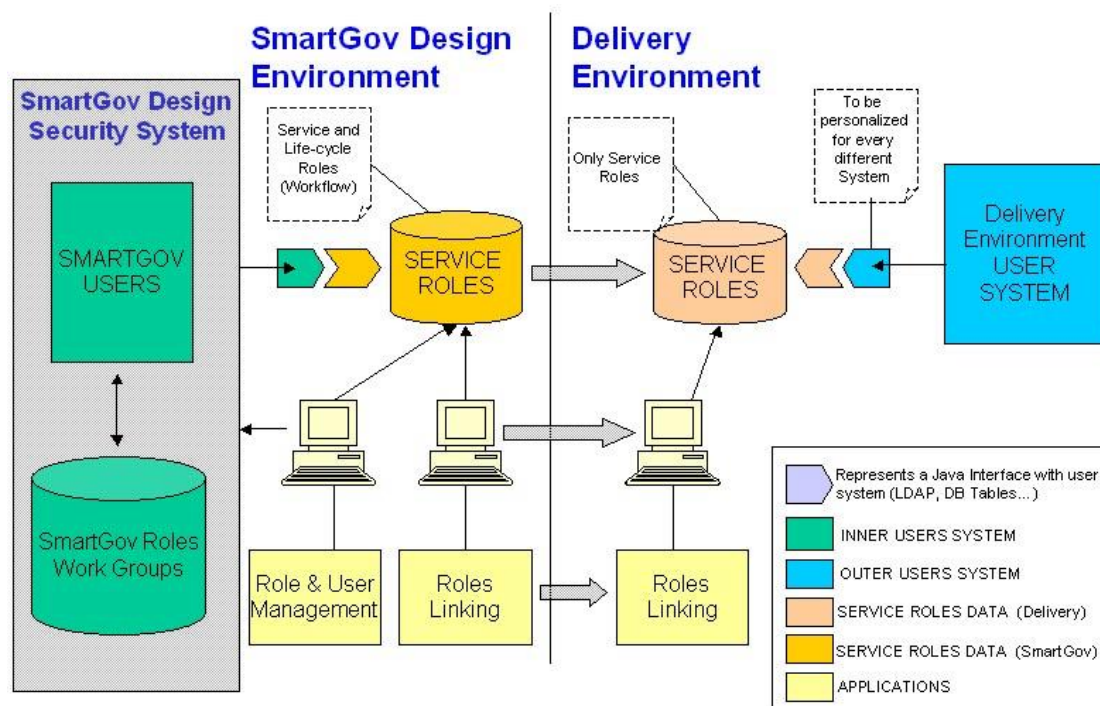


Figure 32 – SmartGov and Delivery Environment Role architecture

This proposed framework is based in the following premises:

- The SmartGov development environment user and roles model is independent of the user authentication system used in the service delivery environment.
- The SmartGov platform provides a role system for services and a mapping tool to link users to Service Roles. Thus SmartGov service designers will be able to define roles for the service and link these roles to the end-users. This will enable to management of service access control. Communication between Service Roles and the end-users of the implemented service in the Delivery Environment will require some development of an additional service (implementation of an interface) that will be hosted in the SmartGov Delivery platform.

This approach fulfils the required needs:

- SmartGov Roles, Work Group and User management.
- Facilitates life-cycle or basic workflow capabilities via Service Roles.
- Isolates and simultaneously connects the SmartGov Service Role with the User System in Delivery Environment. This provides roles functionality, but using the organization User System directly.

It is necessary to develop the following components:

- SmartGov Roles, Work Group, and SmartGov User Management.
- Service Roles Management.
- Connectors between roles:
 - With SmartGov User System
 - With the Delivery Environment User System (during deployment)

A class diagram showing the most important classes, methods, and properties of the user and role management is shown in Figure 33.

Regarding the user and Roles system, a mechanism to connect outer user systems has been developed. It consist on two interfaces, OuterUserSystem and OuterUser. The first one defines an outer user system in which the users of SmartGov platform are stored, and it can be asked to obtain a OuterUser object for an specific user, and so access the data for this user.

In order to integrate a new outer user system with the platform a implementation of these interfaces must be developed. Then the users will be imported into the SmartGov user system and they all will be included in the same group (UserGroup, not WorkGroup), with the corresponding reference to the class that implements OuterUserSystem interface.

The two different classes UserGroup and WorkGroup are related with the two types of groups that will coexist in the system:

- User Groups, used to associate users coming from the same outer system.
- Work Groups, joining all the users involved in the design of the same services, knowledge...

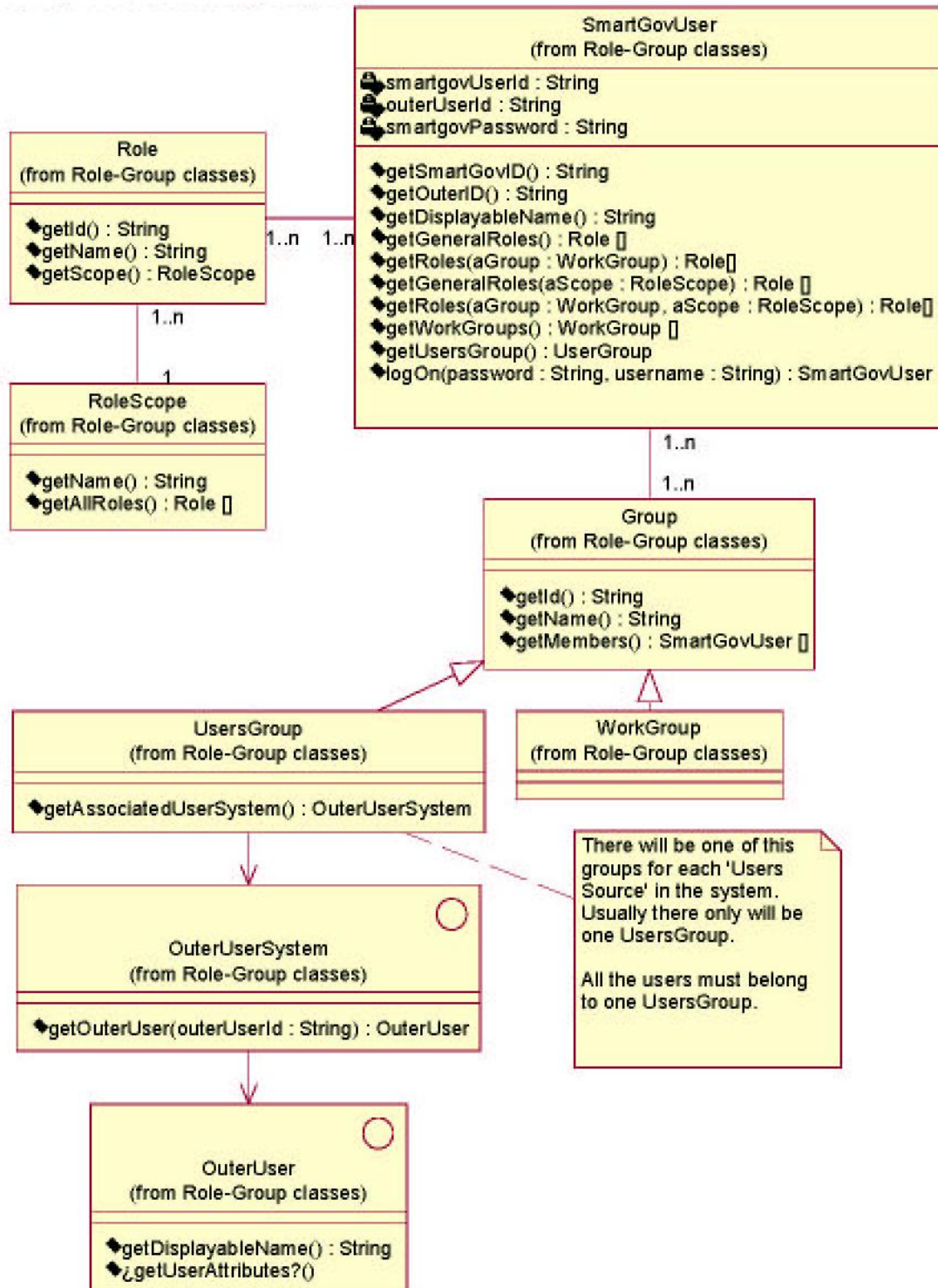


Figure 33 – User and Roles logical class diagram

2.4.2.3.2.5 Service Design environment Statistics

Statistics about KU, Taxonomies, and Taxonomy Nodes will be collected in the SmartGov Service Design environment. Those statistics will gather information about the use of Knowledge and taxonomies (categorization). This information will be useful to knowledge administrators (usually managers or domain experts) to refine, evaluate the quality, and the use of KUs, Taxonomies and Taxonomy Nodes, and will also lead to accurate estimations of the complexity, relevance, and richness of the KUs.

Regarding the SmartGov service design environment statistics, a specific storage in the SmartGov server will be provided. This information is going to be stored in a statistics database. The business-logic called from the Front-End tool will deal with the storage of this data. For example, when a Manager invokes a KU from a TS, a counter should be incremented. These statistics will be system-defined.

2.4.2.3.3 Storage Layer (Castor & XML Doc Repository or RDBMS)

2.4.2.3.3.1 Introduction

In order to assure the persistence of all the SmartGov data, a combined strategy will be used: some data will be stored in a XML Doc Repository (mainly SmartGov objects) and the rest in a RDBMS.

This split is due to the natural characteristics of the information that will be stored:

- The data that will be stored in the XML Doc Repository (see 2.4.2.2) fits exactly with the XML nature. For example, each SmartGov object is a document itself, and a XML file allows to describe all its characteristics, with the required flexibility. Most SmartGov data (TS, Forms, TSE, TSE Groups, KU and Taxonomy) is stored in the XML Doc Repository as it can be seen in the Data view section.

The XML Repository also will be used to save the life-cycle processes definition, because this information rarely will change, so a XML file is a perfect place to hold it. Using Castor, and the API provided by the XML Doc Repository (see 2.4.2.2), the XML data is accessible.

- A RDBMS, on the contrary, fits with a more strictly structured information, relationships, and stronger security, making also easier access and update each piece of data. Therefore, it will be used to store users, roles and work group data, because this information requires continuous accesses, and maintain strong referential integrity that a XML approach would not be so suitable. A JDBC driver will be used to give access to the RDBMS system.

Consequently, the SmartGov Front-end we will combine these two strategies, making the most of each of them.

2.4.2.3.3.2 XML Storage using Castor

The two key technologies that will be used in the XML Storage are the XML Repository and Castor framework. These two modules cover the two main stages required to access XML-stored data:

- Locating the particular XML file that contains the required data (XML Doc Repository).
- Accessing the different data inside the file, navigating the different nodes in the tree structure (Castor).

The XML Doc repository has been described earlier in this document (paragraph 2.4.2.2 so only Castor framework will be described deeply next.

Castor framework, as aforementioned, is an open source data-binding framework, which maps a Java object model to and from XML. Given a XML Schema, Castor is able to process it and generate all the code required to access and modify files that fit the schema. Thus, the XML Schemas defined in the SmartGov platform (see paragraph 2.8) can be automatically mapped to classes. And these classes can be updated whenever is required with every change in the XML Schemas.

Given that these classes provide data access, only adding business logic is required. With this aim, the automatically generated classes will be used, trying to keep business logic and data access loosely coupled. Using as starting point the Data Access Object and Factory Methods patterns, a mechanism to make independent business logic and objects retrieval will be developed. Thus, the data layer can be modified, extended or adapted when needed without requiring changes in the Business objects.

Therefore, with the generated Java classes and the ARC XML Doc Repository all the requirements for XML files storage and retrieval are solved.

Figure 34 shows the proposed architecture, with the complete implementation for the TS storage.

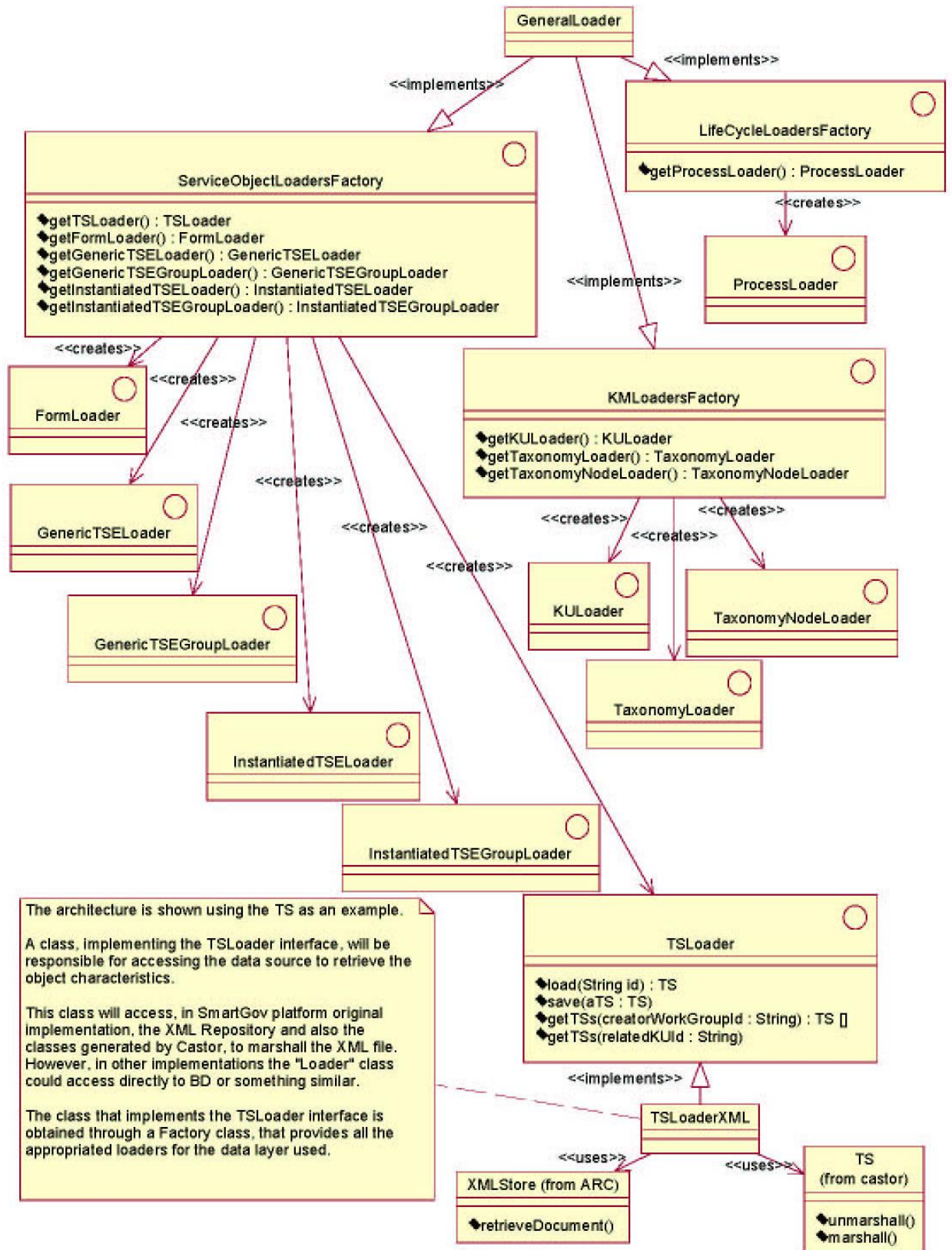


Figure 34 – SmartGov Front-End XML data access logical class diagram

2.4.2.3.3.3 RDBMS Storage

A Relational Database Management System (RDBMS) will be used to store the roles, groups and users data. We will have two separated models, though they work in deep relation.

- A general model that must be included in every SmartGov platform installation. It keeps all the data concerning roles, groups, and an auxiliary table to make available connections with outer user systems (LDAP, database...)
- A user's data model that keeps all the information related to SmartGov users. This table and the corresponding module may be used when using and outer user system is not necessary. Therefore, from the Front-end, the users will be able to manage this user information.

This module and table can be replaced by a LDAP or another similar system. An interface will be defined, and each new user system that will be incorporated to the SmartGov platform will require an additional development to implement this interface, adapted to the particular characteristics of the system.

These two models will be stored in a RDBMS because continuous accesses (and probably changes) will be required, and using XML would not be the best approach.

2.4.2.4 SmartGov Agent – Information Interchange Gateway

When a SmartGov installation is deployed, it is expected to exchange data with an organizational IT system. Usually this IT system will be the "back-end" system for the organization, from which citizen or enterprise registry data will be retrieved and to which transactional data will be stored. Different organizations have diverse back-end systems, with substantial differences or idiosyncrasies, which hinder the use of a common framework for communicating with them. Moreover, communication should take place in a high level of abstraction, without involving design and implementation details of the installed IT system.

One scheme for achieving the aforementioned goals is to employ two software modules, the *SmartGov Agent* and the *Information Interchange Gateway*. The *SmartGov Agent* is an integral part of the SmartGov platform, enabling the submission of *requests* to external systems and the retrieval of the respective results. The Information Interchange Gateway is attached to the installed IT system and arranges for interception of the requests originating from the

SmartGov agents, their execution and the returning of the appropriate results. Communication between the SmartGov agent and the Information Exchange Gateway may be performed using any standard data exchange protocol, such as WDDX, XML etc. The architecture of a SmartGov platform involving the SmartGov agent, the Information Exchange Gateway and an Installed IT system is depicted in Figure 35, while a more detailed positioning of these modules within the SmartGov platform is illustrated in Figure 36.

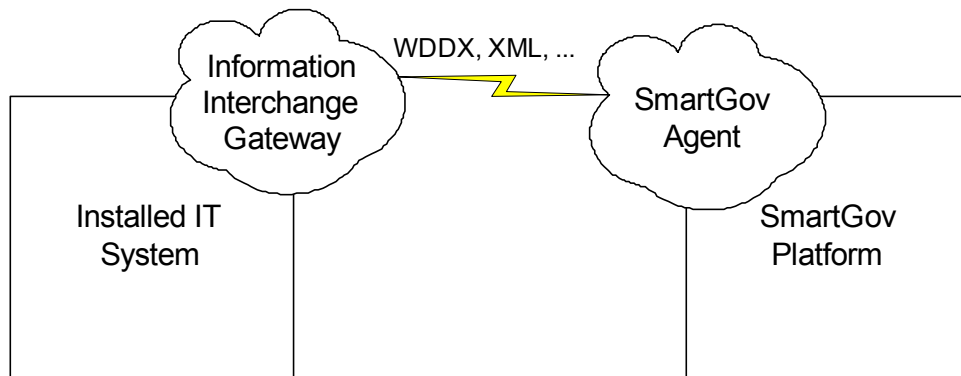


Figure 35: Communication with installed IT systems

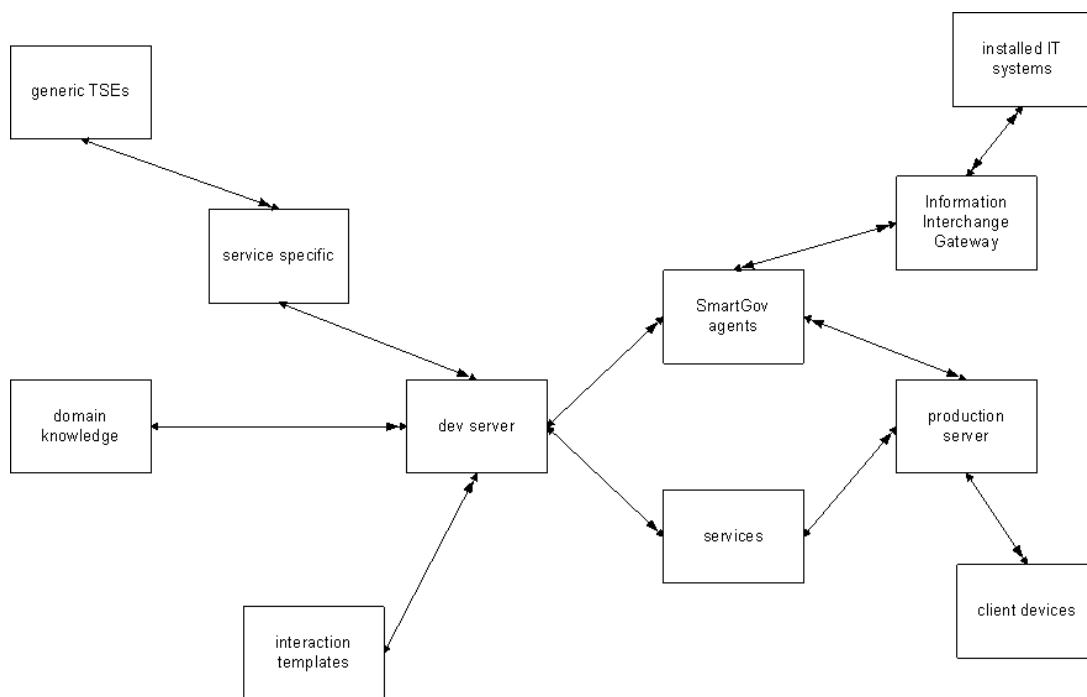


Figure 36: Placement of SmartGov Agents in the overall architecture

The Information Interchange Gateway publishes an *exported service list*, which defines the requests that it is willing to accept and serve. The exported service list is retrievable through a call to the IIG management interface. Each service is described by its name, a set of input parameters packed in an XML message and

a result, which is also returned as an XML document. Requests from the SmartGov agent to the Information Exchange Gateway include the request name and the (request-specific) set of input parameters (an XML document). Upon reception of such a request, the Information Exchange Gateway verifies that the invoked service is included within the exported service list and invokes a procedure, which performs the requested task and returns the results, which are then forwarded to the SmartGov agent. The procedures that actually execute the requested tasks will be coded by the organization's IT staff (or will be outsourced) and they practically encapsulate all the internal details and peculiarities of the installed IT system. This approach is well proven for interconnecting heterogeneous information systems (e.g. CORBA [Bolton2001], RPC [Sun2000]). In the following paragraphs the *SmartGov Agent* and the *Information Interchange Gateway* are described in more detail.

2.4.2.4.1 SmartGov agent

The *SmartGov agent* accepts requests from the SmartGov platform for communication with external IT systems and arranges for the communication to be performed. Each such request contains a *function name*, designating the service to be invoked and may define parameters to be passed to the service, while the service may return results that must be returned to the caller. Upon receipt of a request the *SmartGov agent* should perform the following functions:

1. *Locate the information system to which a request should be made.* The requested service may be offered by one or more IT systems. The SmartGov agent should determine which IT systems offer the requested service, select the most appropriate one and direct the request to it. Service-to-IT systems mappings are determined via configuration files.
2. *Collect the message provided in the context of the invocation.* The SmartGov agent should retrieve from the SmartGov platform the message that should be sent to the IT system, in order for the request to be fulfilled. The SmartGov agent requires that the message collected should be a valid XML message, but does not process or modify its contents in any way.
3. *Invoke the service on the IT system and collect the response.* The request is sent to the IT system and the reply is collected. Communication errors are also handled in this stage.
4. *Extract the results contained within the reply and return them to the caller.* The SmartGov agent receives the reply to the called method and

arranges for it to be delivered to the calling SmartGov application. Again, the SmartGov agent requires that the reply is a valid XML message but does not process or modify its contents in any way.

The SmartGov agent is also responsible for handling triggering events that may affect service operation. For instance, if a service is dependent on a specific law, the SmartGov agent may accept notifications from a legal document database indicating that a specific law is revised or complemented, and will arrange for appropriately reporting to the SmartGov platform administrators.

2.4.2.4.2 Information Interchange Gateway

The Information Interchange Gateway is attached to the IT system that offers services that may be invoked from the SmartGov platform. If multiple IT systems should offer services for SmartGov platforms, then each such system should run a separate instance of the Information Interchange Gateway. The Information Interchange Gateway should encompass the following functionalities:

1. *Service directory.* The Information Interchange Gateway offers through the service directory a list of the services offered by the specific IT system to SmartGov platforms. Each service within this list is described by its name, a set of input parameters packed in an XML message and a result, which is also returned as an XML document.
2. *Service execution.* Once a service invocation from a SmartGov platform has been received, the Information Interchange Gateway should arrange for the execution of the appropriate code fragment that implements the service. As a first step, the values of the parameters accompanying the request should be extracted, and the appropriate code fragment, its execution method and the parameter passing convention that must be employed should be determined. For instance, if the code fragment is an external program, parameters may be passed as command line arguments and results might be returned through the program's output; if the code fragment is compiled into a dynamically loadable library, it will be necessary to load the library and invoke the corresponding entry point within it, passing the parameters through the stack.
3. *Remote administration facility.* This facility enables the installation, de-installation and modification of services offered by the Information Interchange Gateway, without the need for other types of access to the IT system. In order to add a new service administrators should be able to provide a description of the new service, including its name, parameters,

results and invocation method, together with the code fragment that implements this service. The code fragment might be in source form, in which case an appropriate set of commands to transform it in an executable form should be provided. Service de-installation only requires the service name, whereas service modification may be implemented through de-installation followed by a new installation. In all cases, administration facilities are accessed after suitable authentication.

In the event of modifications to the installed IT system, it is expected that the administrators of the installed IT system will notify the administrators of the SmartGov platform, providing any necessary information for bringing the service implementations up to date.

2.4.2.4.3 Technical considerations

SmartGov agents and Information Interchange gateways act on XML-messages they receive or are themselves originators of such messages. These XML-messages will typically have a control and a data part. It is the responsibility of the SmartGov Agent and the Information Interchange Gateway to interpret the control part and act accordingly.

SGA and IIG communicate through XML messages, thus 7-bit clean plain ASCII messages are exchanged between these entities. Taking this into account, any error-free communication channel is sufficient for the exchange of these messages, such as TCP-IP sockets, however the use of higher-level abstractions such as RMI will be considered in the implementation phase.

Specialized or idiosyncratic data formats as well as access to data sources (such as databases or remote systems) are delegated to appropriate SmartGov Agents specifically crafted. Inevitably, this will result in specialized IT staff involvement. The Project's goal is to facilitate the work of IT staff by providing basic, general-purpose mechanisms as well as the XML-messaging infrastructure for communication with other parts of the SmartGov Platform. Frequently used and foreseeable actions could be represented in an abstract form to facilitate easier and speedier development. For example, a generalized method for accessing databases could be provided in the form of a DSN (Data-Source-Name).

As is usually the case in Public Administrations, there are already data centres operating under IT staff that closely guards access to resources for stability, performance, security and other reasons. Update of the SmartGov Agent that sits within these systems is not expected to be a frequent event. Therefore, SmartGov Agents should allow for remote updating of themselves through specific

messages in XML format. W3C specifications of XML-RPC and/or SOAP could be of help in this area.

So far, it has been implied that all necessary information for an e-service to operate is present locally within the SmartGov deployment platform, i.e. it has already been collected from 3rd party systems and stored in a readily accessible storage area by previous actions of appropriate SmartGov Agents. If communication with remote systems is needed, synchronous or asynchronous communication with these systems will be initiated upon receipt of the relevant request.

The amount of effort required for bridging an installed IT system to the IIG depends on the technologies employed by the installed IT system and the complexity of the requested operation. If the installed IT system is a database with JDBC capabilities, from which data must be retrieved (or into which data must be stored) the effort will be minimal since the task is reduced to adding some lines of Java code with JDBC invocations to a given template (which will be documented within the SmartGov platform). For more idiosyncratic systems, the process is expected to be slightly more complicated but templates and guidelines for the situations that are anticipated to occur frequently will be provided.

2.4.2.4.4 Invoking services offered by the SmartGov Agent and the Information Interchange Gateway

Programs executing either in the context of the SmartGov service delivery environment or in the context of the organisational information system may invoke services offered by the SmartGov Agent and the Information Interchange Gateway, in order to complete tasks specified by the implemented services business logic. These invocations are realised using an API offered by the SmartGov platform, which encompasses all the necessary classes and methods for performing the required actions. This API is subdivided into three packages, which offer the functionalities for submitting requests to the SmartGov agent, posting notifications to the SmartGov service delivery environment and arranging for recording messages to the platform log. These packages are discussed in the following paragraphs.

2.4.2.4.4.1 The SGA package

The SGA package enables applications running on the SmartGov service delivery environment to submit requests to the SmartGov agent. These requests will be forwarded for further processing to the appropriate organisational information

system and the results will be collected and returned to the submitting application. The class diagram for this package is illustrated in the following figure.

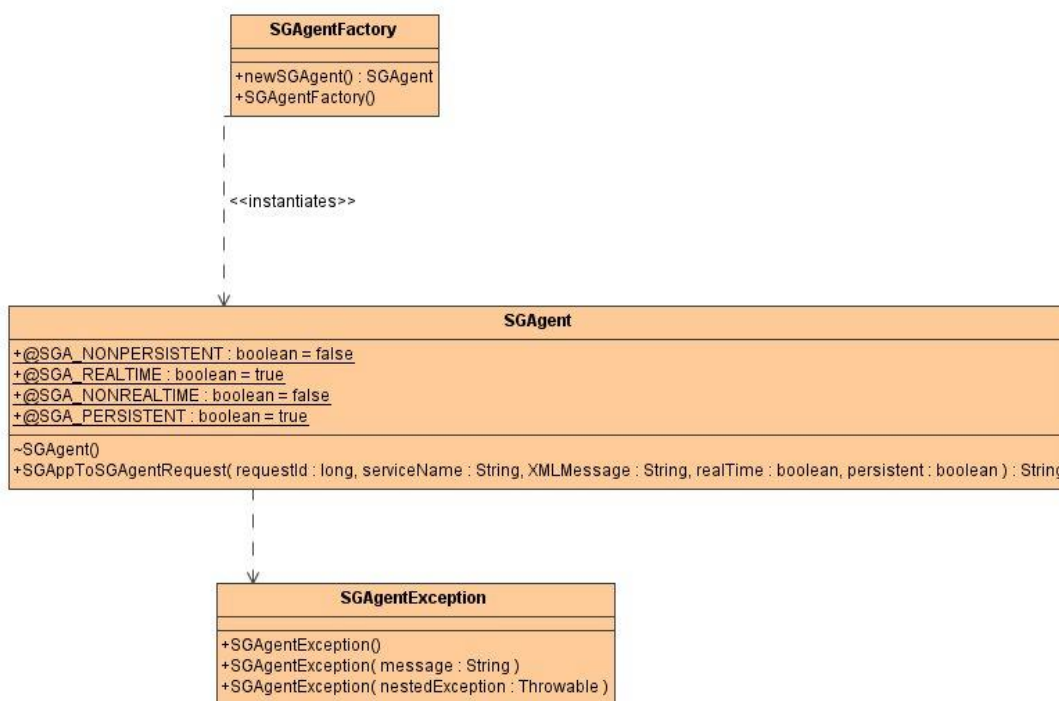


Figure 37 – Class diagram for the SGA package

The SGA package contains three classes, and more specifically:

- The *SGAgentFactory* class; this is a convenience class that provides access to implementations of the *SGAgent* interface. *SGAgentFactory* instantiates an *SGAgent* object through the *newSGAgent()* method.
- The *SGAgent* class. This class provides the *SGAppToSGAgentRequest()* method that arranges for the submission of the request and the returning of the result. Additionally, this class exports symbolic constants that may be used for designating the real time and persistence requirements for the submitted requests.
- The *SGAgentException* class, which models a generic exception type that may be thrown by methods in *SGAgentFactory* or *SGAgent*.

2.4.2.4.4.2 The IIG package

The IIG package enables applications running on the organisational information system environment to post notifications that will be finally received by the SmartGov service delivery environment. The class diagram for this package is depicted in the following figure.

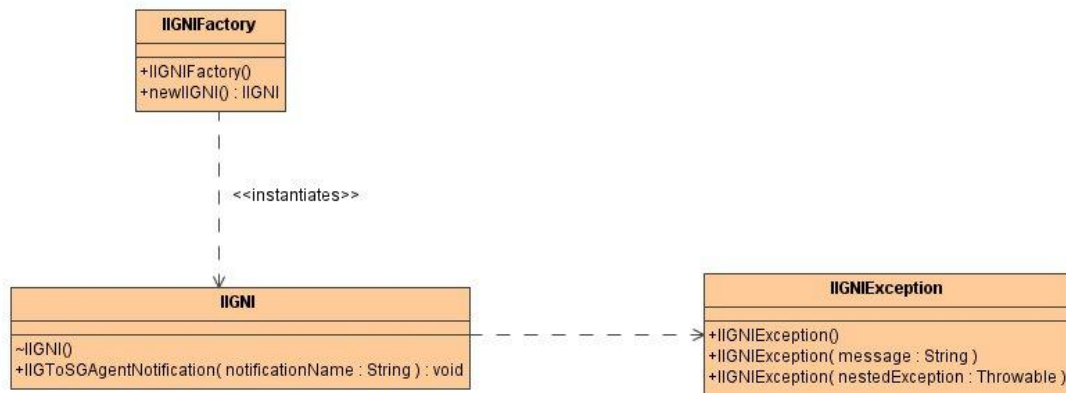


Figure 38 – The IIG package class diagram

The SGA package contains three classes, and more specifically:

- The *IIGNIFactory* class; this is a convenience class that provides access to implementations of the IIGNI interface. IIGNIFactory instantiates an IIGNI object through the newIIGNI() method.
- The *IIGNI* class. This class provides the IIGToSGAgentNotification() method that arranges for posting a specified notification to the SmartGov service delivery environment.
- The *IIGNIException* class, which models a generic exception type that may be thrown by methods in *IIGNIFactory* or *IIGNI*.

2.4.2.4.4.3 The SGLogging package

The SGLogging package enables applications running on the SmartGov service delivery environment or in the environment of the organisational information system to record messages in a platform log. The class diagram for this package is presented in the following figure.

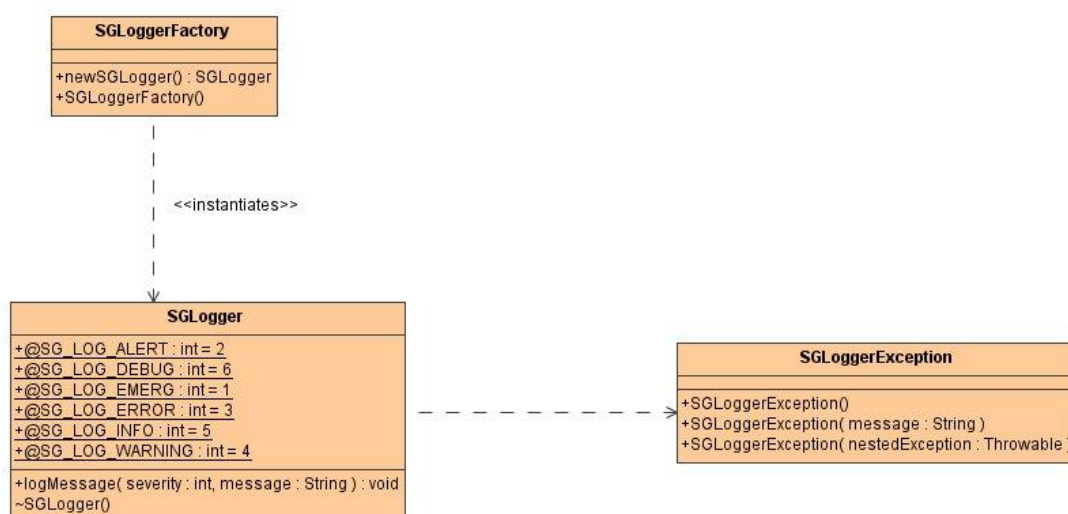


Figure 39 – Class diagram for the SGA package

The SGLogging package contains three classes, and more specifically:

- The *SGLoggerFactory* class; this is a convenience class that provides access to implementations of the SGLogger interface. SGLoggerFactory instantiates an SGLogger object through the `newSGLogger()` method.
- The *SGLogger* class. This class provides the `logMessage()` method that arranges for recording a specified message in the platform log. Additionally, this class exports symbolic constants that may be used for designating the event type and the criticality level associated with the event described by the message.
- The *SGLoggerException* class, which models a generic exception type that may be thrown by methods in *SGLoggerFactory* or *SGLogger*.

2.4.2.4.5A modelling example for the SmartGov agent and the Information Interchange Gateway

In SmartGov project D4.1 the VIES (VAT Information Exchange System) system has been analysed as a candidate for a pilot implementation within the SmartGov project. The requirements for such an application to be developed, regarding the communication between the SmartGov service delivery environment and the installed organisational IT system are as follows:

1. Users should be able to register to the electronic service, providing some personal identification data (which should be known only to them and the taxation authorities). The data from each registration request should be forwarded to the organisational IT system.

2. After processing the registration data, the organisational information system should return the results to the SmartGov service delivery environment, indicating which registration requests should be honoured and which should be rejected (due to incorrect data supplied, attempt to re-register an already registered user etc).
3. When a registered user requests to submit a new declaration, the user's personal details should be fetched from the organisational information system's registries, in order to appear pre-filled in the corresponding form fields.
4. When the user submits a new declaration, the content provided by the user should be sent to the organisational information system for storage.
5. A registered user should be able to retrieve the description of the submitted declarations.
6. A registered user should be able to retrieve the contents of a specific declaration that has been submitted.

In order to provide this functionality, the following services should be offered by the IIG attached to the organisational information system:

1. a service that accepts the user registration details and stores them. These details will be then processed by the organisation's back end, as specified by organisational policies. This service is invoked synchronously by the programs in the SmartGov service delivery environment, when the user "submits" the registration form.
2. a service that will return to the SmartGov service delivery environment the lists of correctly verified and rejected user registrations. This service is invoked by a specially designed program that runs on the SmartGov service delivery environment and whose execution is triggered by a notification event, as explained later in this section.
3. a service accepting a user's identification and returning the user's personal details. This service is invoked synchronously by the programs in the SmartGov service delivery environment, when the user chooses to start filling a new declaration.
4. a service accepting a user's identification and a declaration's data and storing these data in the organisational information system, tagged with the user's identification. This service is invoked synchronously by the programs in the SmartGov service delivery environment, when the user submits a new declaration.

5. a service accepting a user's identification and returning the description of the documents submitted by the specific user. This service is invoked synchronously by the programs in the SmartGov service delivery environment, when the user chooses to view a list of submitted declarations.
6. a service accepting a user's identification and a declaration's identification and returning the contents of the designated declaration. This service is invoked synchronously by the programs in the SmartGov service delivery environment, when the user chooses to view a specific declarations.

These functionalities offered by the IIG will be complemented by a notification event that is posted from the organisational information system's environment and signifies that the user registration requests have been processed and the corresponding "accepted" and "rejected" lists have been formulated. The SmartGov agent should be additionally configured so that upon the reception of such a notification event, a process will be initiated that will invoke service 2 (retrieval of the lists of correctly verified and rejected user registrations). This lists will be further processed by this program to produce usernames and passwords for the accepted users and communicate these credentials to the submitters of the corresponding requests, and notify the submitters of rejected requests that their attempt has failed.

2.4.2.5 The Integrator

The integrator is the heart of the SmartGov development platform. Its task is to automatically generate the necessary files, objects, and components in order to create a fully operational e-forms web application. The Integrator will have a very minimal interface integrated in the Front-end described in section 2.3.3 and will be loosely coupled with this application in the sense that the input of the integrator will be the output of the front-end tool. In more detail, the integrator will have to access the XML repository in order to retrieve the necessary documents that are stored there by the SmartGov front-end tool. As described in previous sections these documents define an e-forms web application in a structured way and include all the information that the integrator component needs. The aim of the integrator is then to generate all the necessary files-objects that the STRUTS framework requires in order to operate.

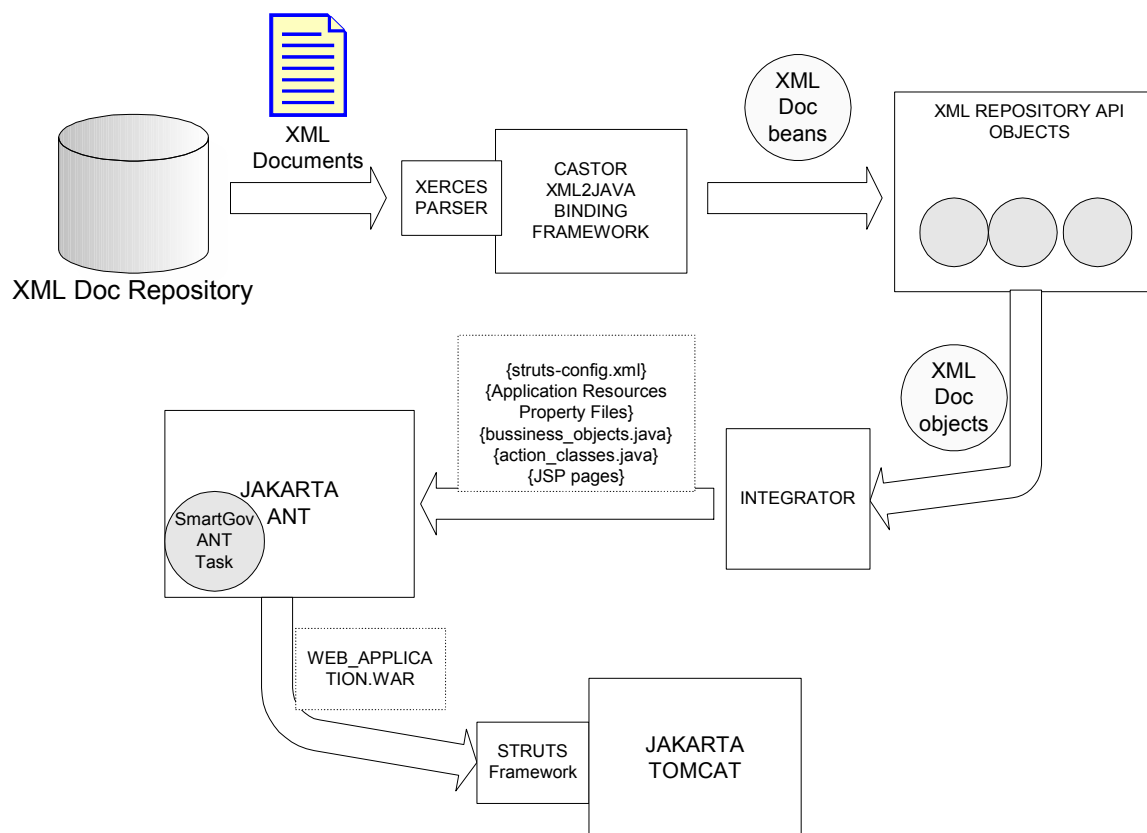


Figure 40. The software environment of the integrator

In Figure 40 the software environment of the integrator is shown along with the input and output that is expected of it. It should be clarified at this point that the integrator will output text or XML files and not compiled class files. This is the task of the following component as depicted in the figure logical flow, which is the Jakarta ANT. ANT is a Java-based built tool. It uses XML based configuration files calling out a target tree where various tasks get executed. Each task is run by an object that implements a particular Task interface. Ant includes a number of ready to run tasks like archiving tasks, compile tasks, deployment tasks and many more. However it also provides an API in order to write your own tasks (additional information about ANT can be found at [Ant]). In the context of SmartGov a new SmartGov ANT task will be developed in order to compile all the generated files of the integrator. This task will be started transparently by the Integrator and at the end will produce a WAR file and place it in the file structure of the Jakarta tomcat server. This way the web application will be straightly available to all end users.

2.4.2.5.1 Associating Layout with TSEs

The first task of the integrator is to associate the layout template, which is in XHTML format with all the TSEs relevant to it as explained in section 2.3.2.7. The procedure for achieving this is shown in the next sequence diagram (Figure 41).

At the start of the procedure the user requests the XHTML template associated with a particular form. The Front-end components of the integrators access then the XML repository API and obtain the XHTML template, which is then presented to the user. The user then initiates the procedure of mapping the form elements of the XHTML template with the relevant TSEs. For this purpose the XMLStore API is called again and the value of the Included TSE element is extracted. At this point a transformation of the XHTML file is needed in order for all form elements to be substituted by a list box presenting to the use all available TSEs. This transformation is accomplished with the use of an XSL engine, which in our case is XALAN of the Apache XML project. Xalan-Java is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. It implements the W3C Recommendations for XSL Transformations (XSLT) and the XML Path Language (XPath). It can be used from the command line, in an applet or a servlet, or as a module in other program(additional information about Xalan can be fount at [Xalan]).

Using the Xalan-API the integrator performs the aforementioned transformation and presents the resulted XHTML file to the user. The user is now capable of performing the required associations and submit them to the integrator front-end components. At this final stage the integrator invokes Xalan again in order to fill in the TSE mappings element of the form.xml file and after that updates the XML store with it.

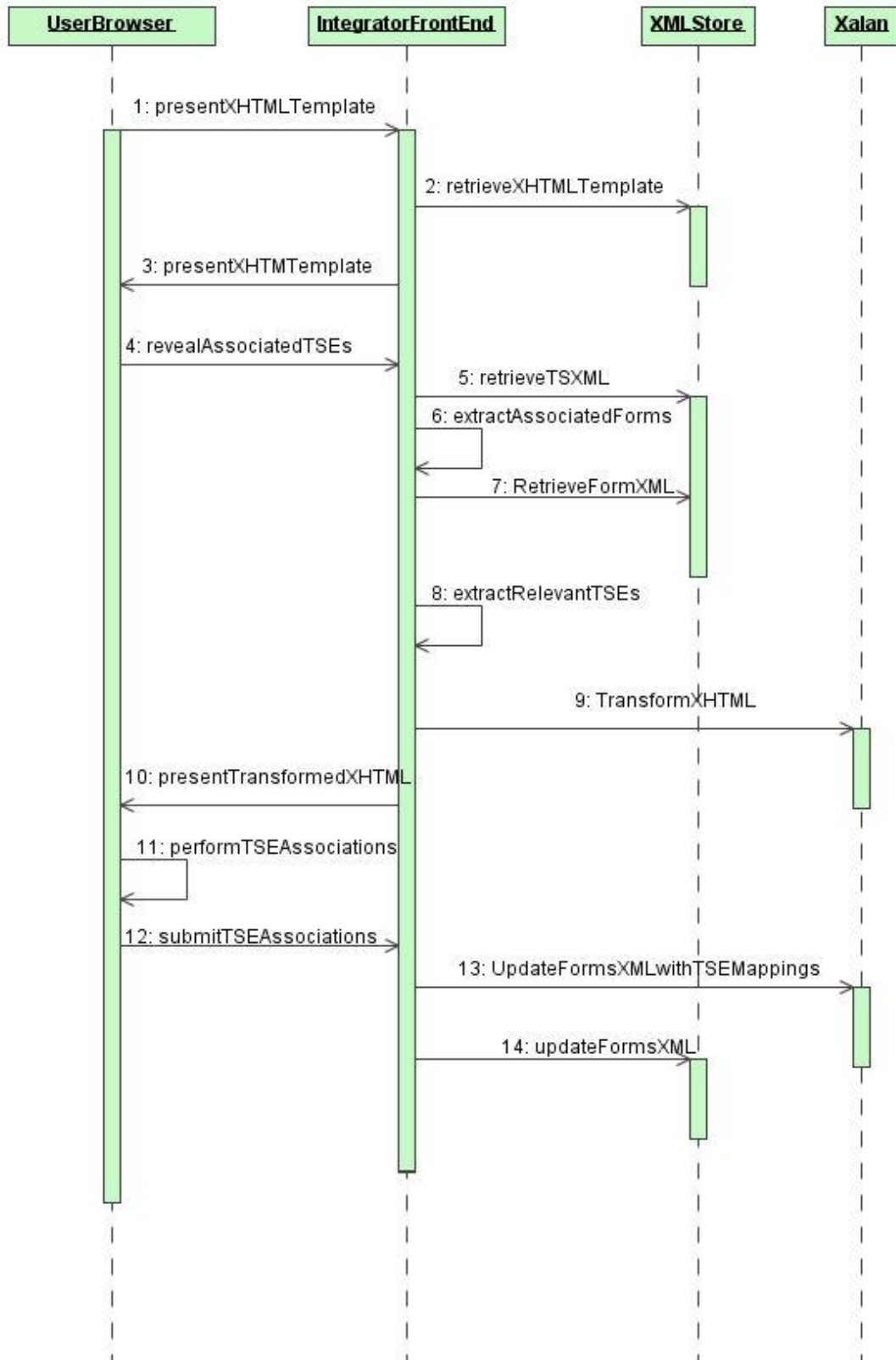


Figure 41 Performing TSE mappings

2.4.2.5.2 Generator's Logical Architecture

As mentioned in previous sections the integrator will undertake the task of generating all the files-objects that the Struts framework needs in order to operate. This implies that JSP files, property files as well as java files will have to be created and then passed to Ant in order for compilation packaging and deployment to occur as mentioned earlier. To achieve this the integrator core logic will have the logical structure depicted in the following figure.

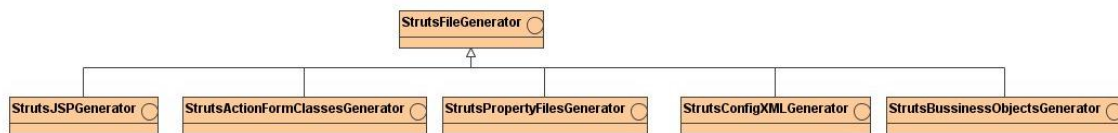


Figure 42 Struts Files Generator classes

Each interface of the above is responsible for controlling the generation of a distinct category of files that the Struts framework requires. Thus:

- The implementation of the StrutsJSPGenerator interface will undertake the task of handling the generation of JSPs, which will be part of the View components of the resulted Struts application.
- The implementation of the StrutsActionFormClassesGenerator interface will be responsible for the generation of the ActionForm and Action Java files part of the Struts model components.
- The implementation of the StrutsPropertyFilesGenerator will be responsible for the generation of the property files that will contain all the internationalized resources of the web application
- The implementation of the StrutsConfigXMLGenerator will be responsible for the generation of the Struts-config.xml file required for the configuration of the controller servlet of the Struts application
- The implementation of the StrutsBusinessObjectsGenerator will be responsible for generating the Java files that will constitute the the business objects of the final application

2.4.2.5.3 Generating JSP files

In order for the integrator to be able to generate the JSP files required, the procedure shown abstractly in the following sequence diagram will be followed.

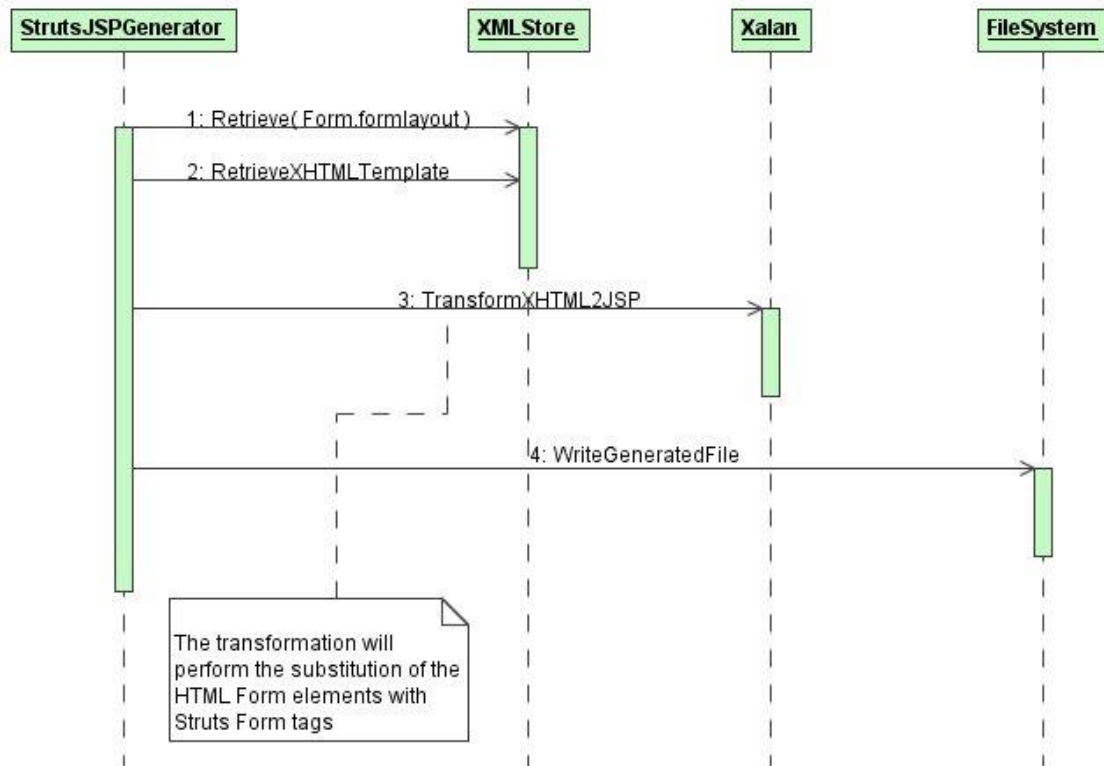


Figure 43 Transforming an XHTML template to a JSP page

The transformation required in this step also uses Xalan since the XHTML template is a valid XML document. The purpose of this procedure is to replace all the HTML form elements with the provided Struts Form tags. This will allow the use of the Struts tag libraries, which enhance the functionality of the JSP pages making them easier to integrate with the model components.

2.4.2.5.4 Generating the resources of the web application

Another important step in the configuration of the Struts Framework is the creation of the property files that will hold the internationalized resources of the web application. The procedure for achieving this is depicted in the following figure.

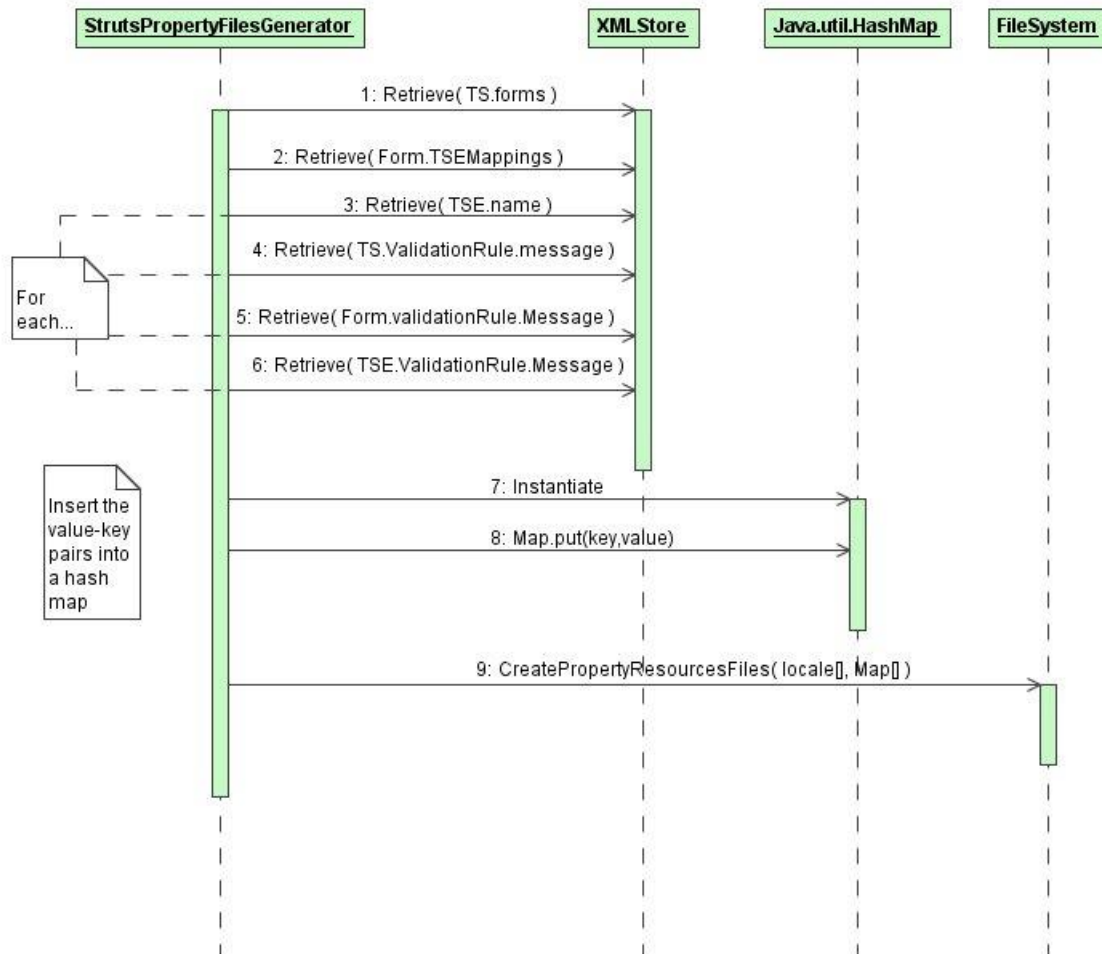


Figure 44 Creation of internationalized resources

When the process starts the StrutsPropertyFilesGenerator component accesses the XMLStore in order to retrieve all the resources that the web application will use in its view and require internationalization. It then uses a HashMap in order to store all the value-key pairs. Finally for its supported locale it creates a property file that will be used globally by the application. This property file will be available at run time to the JSPs with the use of the ResourceBundle class called through the relevant Struts tags (i.e. <error>, <message> etc..)

2.4.2.5.5 Generating the validation code

Another important part of the SmartGov platform will be the ability to provide to the public employees an easy and user-friendly way to express validation constraints. As described in section 2.3.2.3.4 the validation checks required by this type of application fall, in their great majority, under four specific categories. Based on this important conclusion a relatively simple validation language will be

developed to facilitate easy definition of validation checks. This language will be called the SmartGov ValLang. For its realization the components depicted in the following figure are needed.

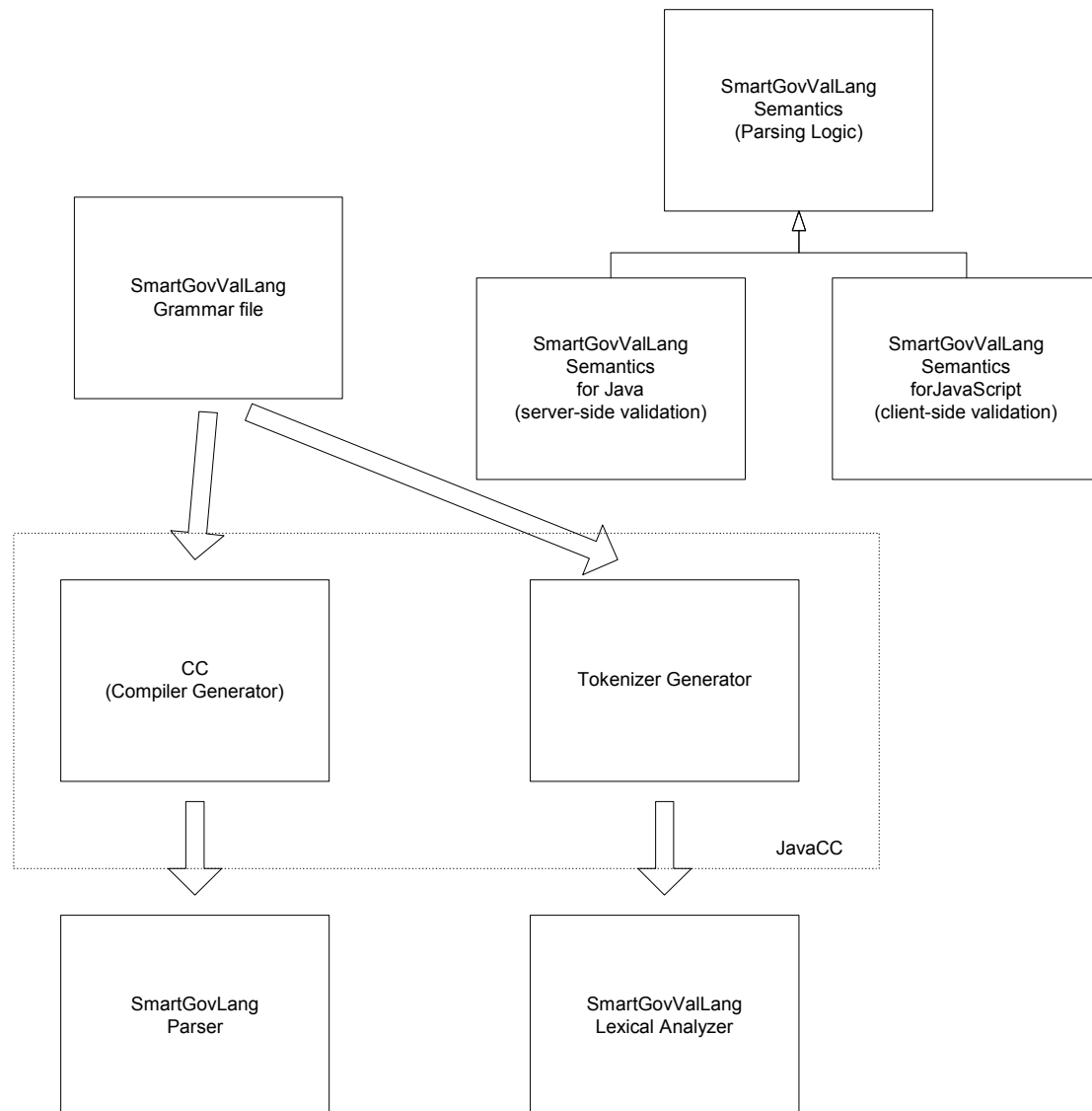


Figure 45 Development of a custom validation language

As shown in the figure the JavaCC tool will be used to assist in this task. Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java applications. It is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc (for more information on JavaCC consult [javaCC]).

Thus the first stem is to write the SmartGovValLang Grammar file. This file will then be fed into the JavaCC module. JavaCC includes both a compiler generator (Compiler Compiler) as well as Tokenizer Generator. The first module is equivalent to yacc (famous cc for use in C applications) and the second one to lex (also used for C based custom languages). At the end JavaCC will generate the SmartGovValLang Lexical Analyzer and the SmartGovValLang parser.

The final step then is to develop the SmartGovValLang Semantics which is a Java module that will contain the parsing logic for the custom language. In other words, in its specialization classes, is where the transformation of the SmartGovValLang into Java for server validation or JavaScript for client validation will take place.

The java code for server-side validation is always generated, whereas the Javascript code for client-side validation will only be generated if the validation check has been designated to be run both at the front-end and at the back-end.

In the following sequence diagram the procedure for ensuring server side validation is depicted.

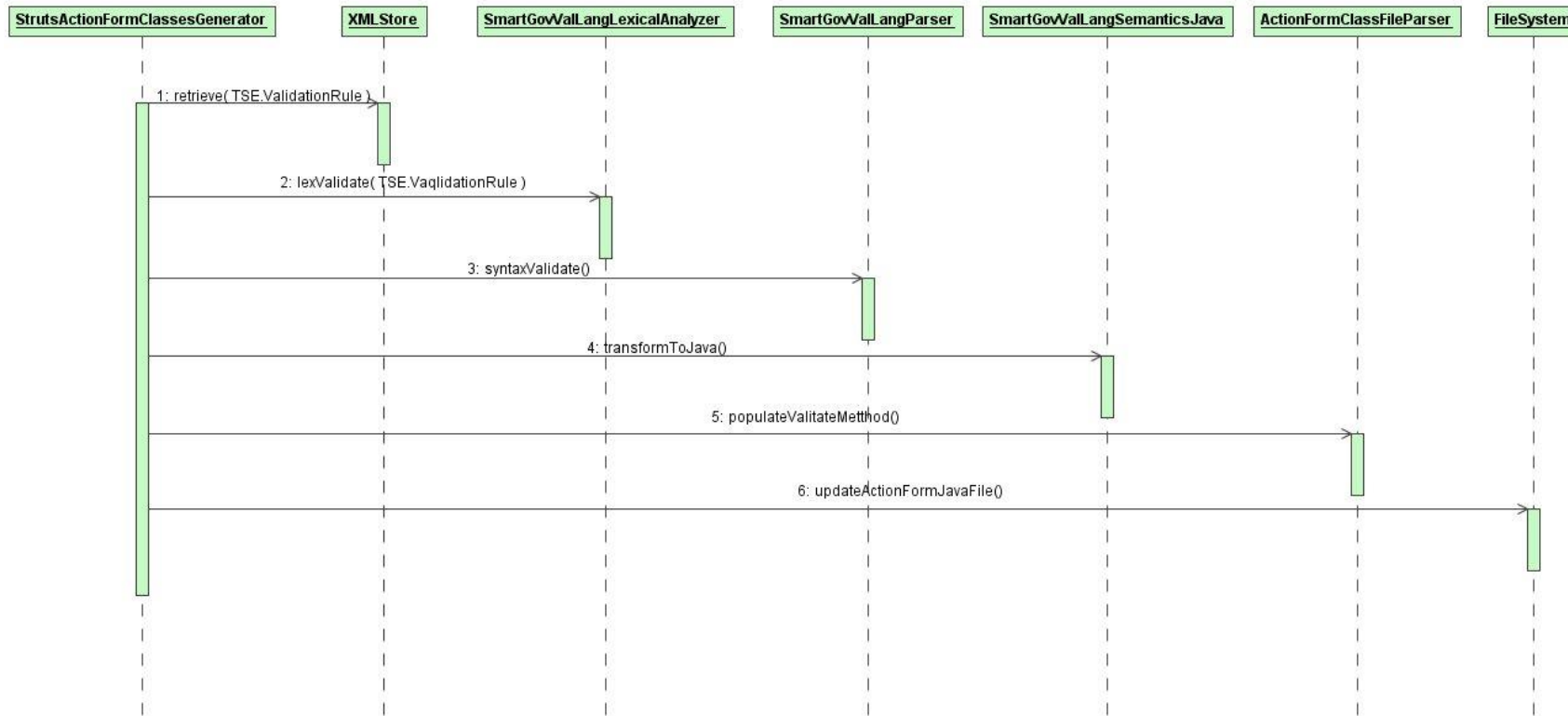


Figure 46 Generation of Java validation code from the custom SmartGovValLang

2.4.2.5.6 Generation of the ActionForm Class File

The ActionForm class is needed in the Struts framework in order to make available to the model and business objects all the parameters submitted by the user client. Thus this java file must be populated with all the required getter and setter methods for all the submitted parameters of the application. In relatively small form application the ActionForm bean will be one, in bigger applications this logic must be spanned in several bean classes. This procedure is depicted in the following sequence diagram.

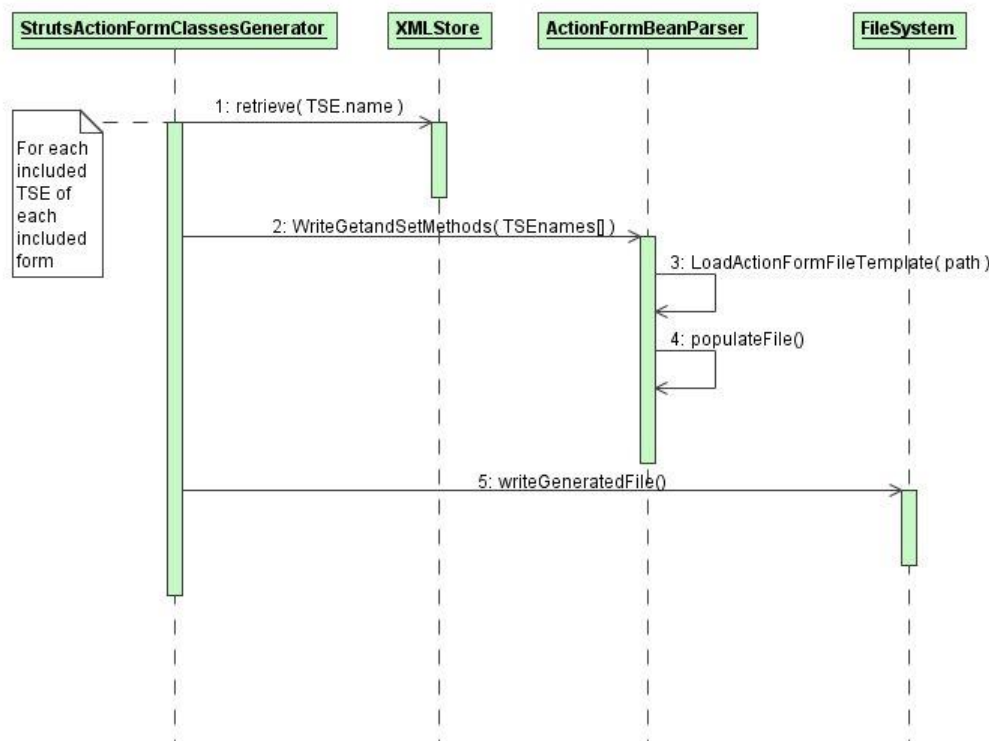


Figure 47 Action Form Java file generator

2.4.2.5.7 The Action Classes

The Action Classes required by Struts will not be generated in the manner that other files will. Since in our case the generated applications will be only forms applications a set of standard Action Classes will be developed and used accordingly to all applications. The StrutsActionFormClassesGenerator will request the appropriate classes from an Action Classes Handler and include them in the application components.

2.4.2.5.8 Generating DB Schema and Relevant DB Access Objects

Apart from the components that are explicitly required by Struts in order to operate, the final web application must have a data layer in order to support persistence and storage of data. For this purpose an RDBMS will be used in which data will be read, written and updated. Thus for each application that is generated a corresponding DB Schema must also be generated along with DB access beans that will provide JDBC access. This task will rely in the use of DDL scripts that will be fed in the RDBMS and will cause the generation of the database that will realize the data layer of the application. In order for this to be realized a generic DDL template will be elaborated that will be customized for the needs of each generated application. The aforementioned procedure is depicted in the following sequence diagram.

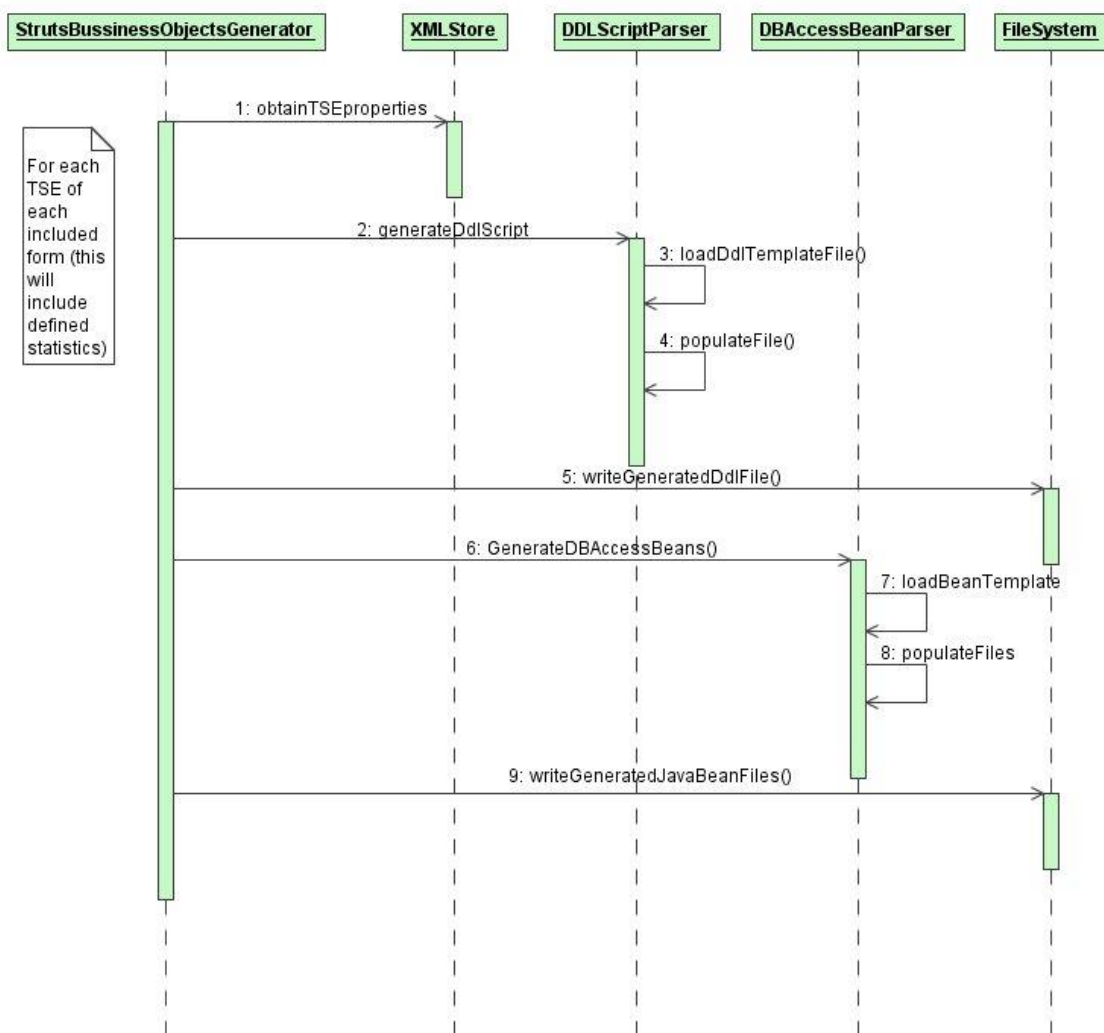


Figure 48 Generating a DB schema and relevant DB access objects

2.4.2.5.9 Generation of the Struts-config XML File

The Struts-config XML file is required for the configuration of Struts. This file must include all the ActionForm beans and Action classes implemented for the specific application in the format defined in its Schema file. It also requires the Database connections and datasources that the application will use for Database access.

The generation of this file will be handled by the StrutsConfigXMLGenerator component. This component will access a proxy object, which will store all the actions performed by the other generators and thus will hold all the relevant data required by the Struts Config file. Additionally a template of a generic Struts-config file will be used and populated with the required configuration info.

The proxy object mentioned before will follow the Proxy design pattern and will function as a mediator component for all the generators of the integrator. This way it will log all action performed and will have complete knowledge of every file generated.

2.5 Implementation View

2.5.1 The SmartGov Front-End

Taking as starting point the Logical view, the next diagrams show the basic structure of the system.

The implementation view describes how the classes and logic in the Front-end Logical View are physically implemented with source code. It also describes how the implementation source is physically contained in files and packages, and how these files combine to form components.

At the highest level, the Front-end application is comprised of three packages. This first schema shows the general structure of the Front-end based in the usual three layers structure.

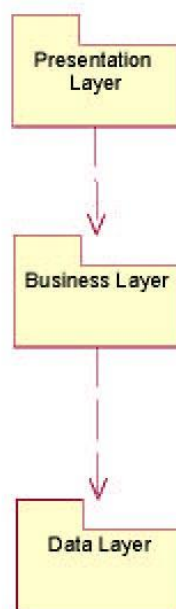


Figure 49 – Three layer Front-end structure implementation diagram

After this first introductory schema, each layer will be described deeply.

First of all, the presentation layer, whose structure is fixed by the use of Struts framework. Thus, this layer will be composed by three big types of components:

- Action classes.
- ActionForm classes.
- JSP pages.

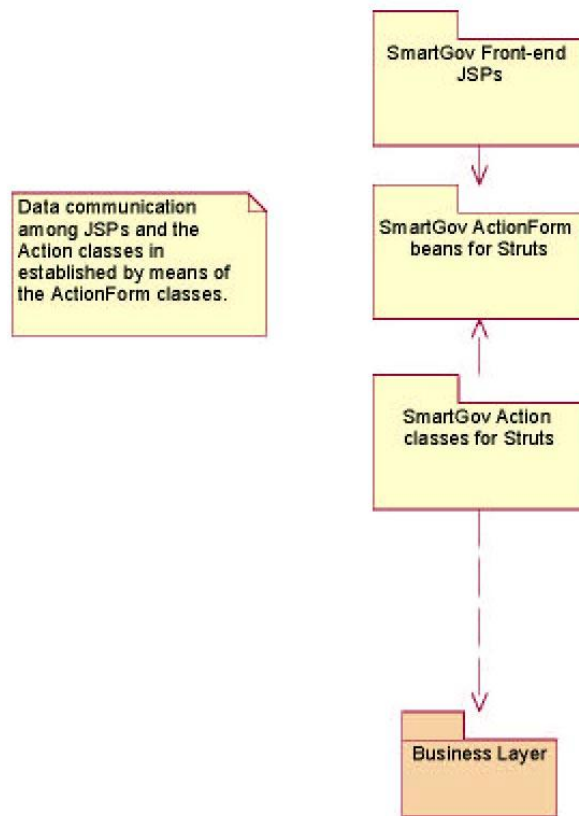


Figure 50 – Front-end Presentation Layer implementation diagram

As it can be seen in the previous diagram, the Action classes are responsible for connecting to the business layer, following the Struts paradigm.

SmartGov Front-end's business layer is implemented by combination of several packages. The following component diagram shows how the packages in the business layer are implemented and the connections between them.

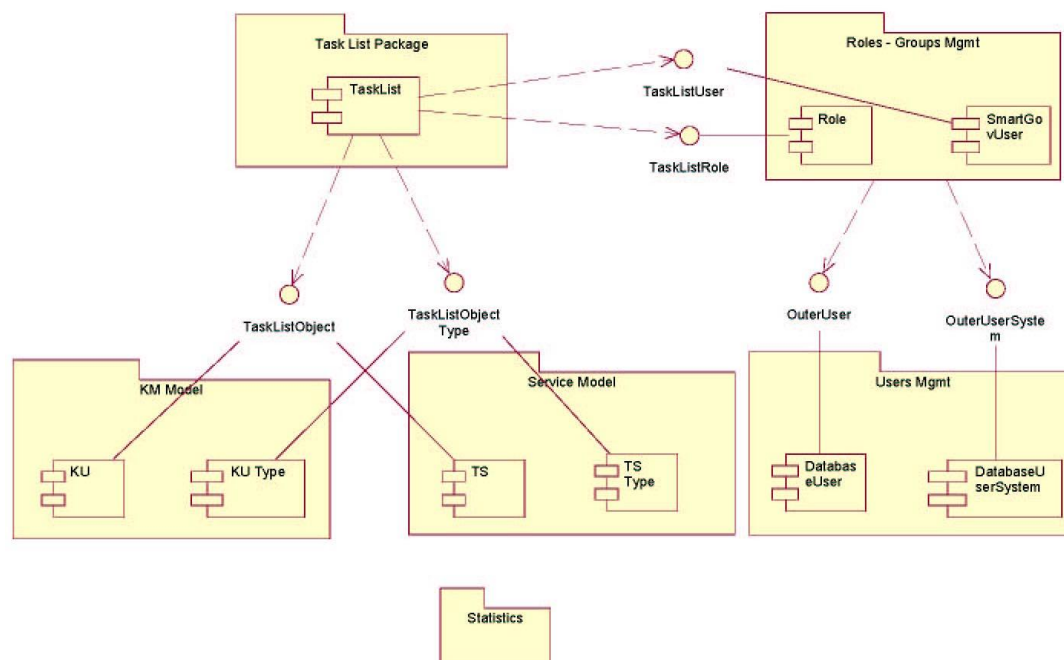


Figure 51 –Front-end Business Layer implementation diagram

In the diagram the five main packages can be seen: Task List, Roles-Group Mgmt, User Mgmt, KM Model and Service Model.

In the diagram, relationships, by means of interfaces, between the different packages are shown. These are the main relations:

- **TaskListObject and TaskListObjectType:**

The Task List package can be used to manage different types of objects (KUs and TSs in SmartGov platform). This package provides the required functionality to control the actions that can be performed over an object according to its current status, and also checks if a user has the required permits to perform those actions.

To include an object type in the task list, two classes must be defined to implement the mentioned interfaces. The first one must be implemented by the objects that will be managed by the Task List (KU and TS classes). The second interface, TaskListObjectType, will be implemented by auxiliary classes (KUType and TSType). These classes will define configuration data for the Task List, for instance, the process definition with the tasks, the states and roles; and also will provide methods to retrieve a user's task list.

- **TaskListUser and TaskListRole:**

In order to control user's permits and the tasks that this user can perform (defined by their roles) is necessary a User and Roles system. But, to keep the packages loosely coupled, instead of using directly SmartGov Roles user two interfaces has been defined. In SmartGov platform, the interfaces will be implemented by the Roles and Groups package, whose classes provide the required functionality.

- **OuterUser and OuterUserSystem:**

The Roles and Groups Management package is designed to make it independent of the used User System. In fact, it can use different User Systems simultaneously. To achieve this, the corresponding classes implementing this two interfaces have to be defined, to connect the outer system with the SmartGov platform.

The first interface, OuterUser, represents the user's data in the outer system. By means of the class that implements this interface, basic data like the first name, surname, email address... all stored in the outer system, may be accessed.

The second one, OuterUserSystem, follows the Factory pattern, and provides access to implementation of the OuterUser interface. By means of the classes implementing this interface, the system is able to create the instances of the classes that implement OuterUser, that is, to create the OuterUser objects.

The statistics package is responsible for collecting all the statistics in the design environment about KM objects. Therefore, it will be connected with the KM module. Probably a event-listener interaction will be defined.

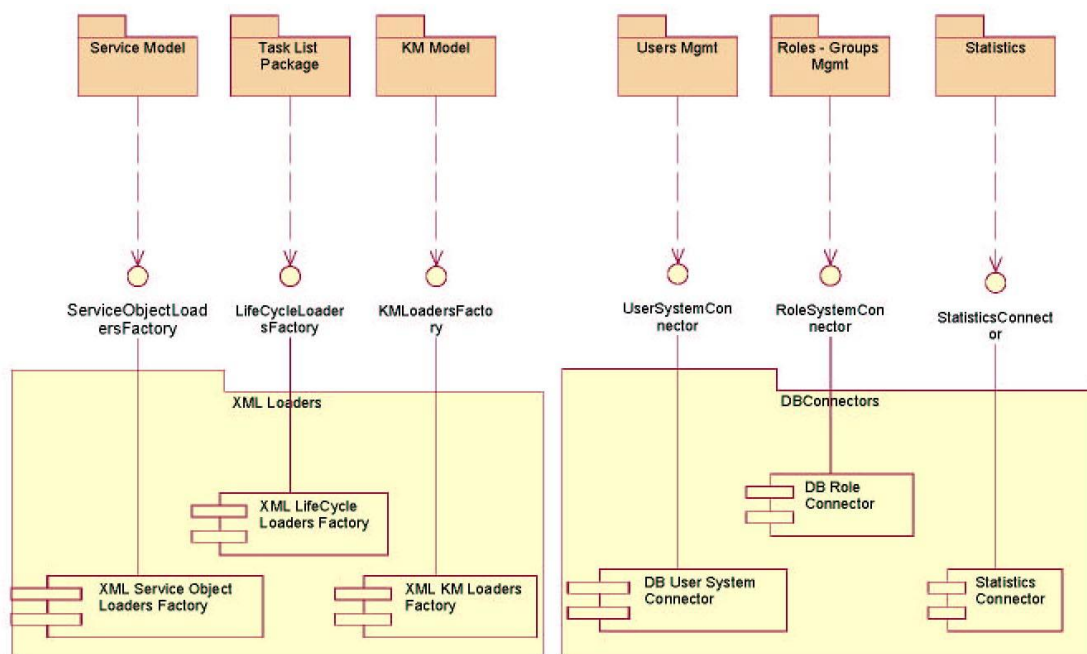


Figure 52 –Front-end Data Layer implementation diagram

To access to the data several interfaces between the business layer and the data layer has been defined as it can be seen in the previous diagram.

- **XMLLoaders (ServiceObjectLoadersFactory, LifeCycleLoadersFactory and KMLoadersFactory):**

The XMLLoaders follows the Factory pattern to make the business objects independent from the XML data source. These three interfaces are factories to obtain loaders classes for each type of object, that will implement the Loader interfaces defined in the logical view.

- **DB Connectors (UserSystemConnector, RoleSystemConnector and StatisticsConnector):**

The DBConnectors make the business objects independent from the RDBMS data source. All the required methods to access data layer will be defined in these three interfaces, that in SmartGov platform will be implemented by Database Connectors, but that can be replaced whenever is required.

2.5.2 Integrator

In the following figure the implementation diagram of the integrator component is shown.

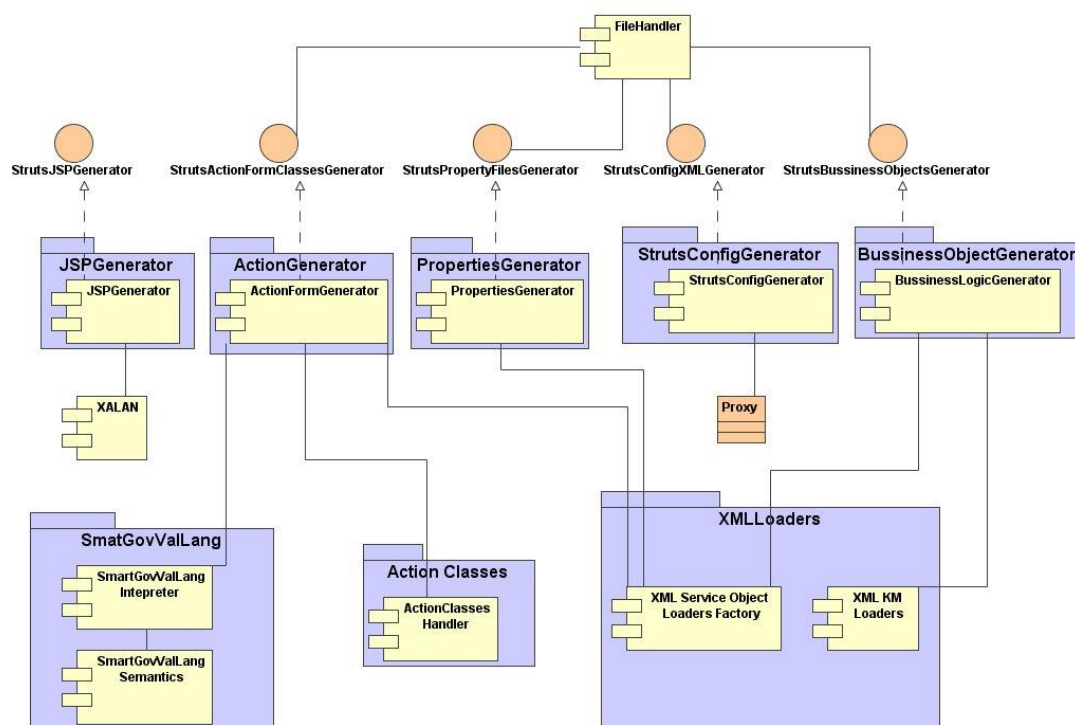


Figure 53 Integrator implementation diagram

The integrator is divided in five basic packages, which are responsible for generating the components that the Struts framework requires. The JSP generator package is responsible for the view components of the web application along with the Properties Generator package, which will handle all the internationalized resources. The Action Generator package will be handling the model part of the web application while the Struts Config generator package will generate the Struts-config.XML file required for the configuration of Struts. Finally the Business Objects Generator will handle the generation of the business objects that the application requires. All the aforementioned components will be the implementations of their corresponding interfaces and will use the XML Loaders package described in section 2.5.1.

The SmartGovValLang package is the package containing the classes and logic to interpret the custom validation language of the SmartGov platform to the JavaScript and Java necessary for client and server side validation correspondingly.

2.5.3 SmartGov Agent – Information Interchange Gateway

2.5.3.1 Software module structure

In this section we summarise the software components that comprise the SmartGov Agent and the Information Interchange Gateway, and give details on the operation of each one. We also list the software components of the SmartGov platform and external IT systems that interact with the SmartGov Agent and the Information Interchange Gateway.

1. *SmartGov application (SGoVApp)*. Any application (delivered service) within the SmartGov service delivery platform. This is implemented as programs in any language appropriate for the service delivery environment (e.g. java server pages, servlets etc), which are spawned by the server employed in the service delivery environment (e.g. web server, WAP server, etc).
2. *SmartGov Agent (SGA)*. A class library containing the methods that allow SmartGov applications to submit requests and retrieve results.
3. *SGA Pending Actions Queue Dispatcher (SGA-PAQUED)*. An autonomous program that periodically scrutinises the pending actions queue on the SmartGov service delivery platform, extracts actions that can be carried out, and initiates their execution.
4. *SGA Notifications Interceptor (SGA-NI)*. An autonomous program that continuously runs on the SmartGov service delivery platform and listens for notifications signifying that an external to the platform event has taken place. The SGA-NI responds to these notifications by placing a suitable entry in the SGA-PAQ, which will be handled by the SGA-PAQUED.
5. *IIG Minimal Yoking Processor (IIG-MYP)*. An autonomous program that continuously runs on the environment of the organisational IT system to enable the reception of requests emerging from the SmartGov applications and delivered through the SGA and the returning of the answers.
6. *IIG-Separate External Process (IIG-SEP)*. Heavyweight or lightweight process spawned by the IIG-MYP in order to fulfil a received request and produce the reply.
7. *IIG-Notification Initiator (IIG-NI)*. A process running in the environment of the organisation's information system and posts a notification event, whose final destination is the SGA-NI, in order to signify that some event has taken place. The IIG-NI sends the notification by calling an appropriate API to a library of methods that arrange for posting the notification to the IIG, which in turn forwards the notification to the SGA-NI.

8. *IIG Pending Actions Queue Dispatcher (IIG-PAQUED)*. An autonomous program that periodically scrutinises the pending actions queue in the environment of the organisation's information system, extracts actions that can be carried out, and initiates their execution.
9. *SGA Logger and IIG Logger*. Autonomous programs that continuously run on the SmartGov service delivery platform environment and the organisational information system environment, respectively, arranging for accepting requests for event logging and storing the log request contents to persistent storage.

In the following paragraphs these software modules and the communication between them are presented in more details.

2.5.3.2 SGA-IIG Communication Details

Applications developed within the SmartGov Framework (SGoVApps) delegate all communications with external IT systems to the SmartGov Agent (SGA). The SGA communicates with the Information Interchange Gateway (IIG) and returns results to the calling application, as shown in Figure 54.

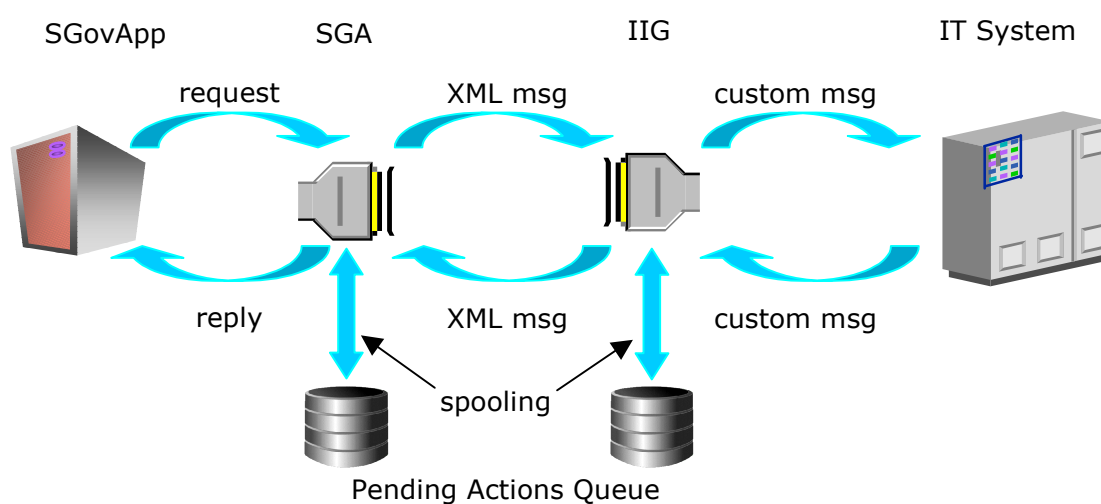


Figure 54 SGA-IIG: Sequence of messages needed for a generalized form of communication

A generic communication event is an event that spans the SmartGov Platform and reaches a 3rd party system. Initiation of communication may be initiated from the SGoVApp (SmartGov Application) or from the IT system and each receiving party has the responsibility of checking all necessary conditions that must hold for the event to complete. There are several cases that need to be considered depending on the initiator of the communication, the time-constraints that are put on it and

the periodicity of the event. In all these cases the same basic message format is used while explicit provisions are made to cover for exceptional cases.

2.5.3.2.1 SGoVApp requests

A SGoVApp initiates communication sending requests to an SGA using messages with the format shown in Table 1.

SGoVApp to SGA general message	<pre> < request_id, service_name, <XML_message>, real_time_indicator, persistence_indicator > </pre>
where,	
request_id	A unique request identifier that serves to characterize this request
service_name	A symbolic service name that the message refers to. The receiving SGA is expected to forward the encapsulated XML_message to the named service
<XML_message>	A message that contains all information that the named service_name requires. The SGA does not interpret this message, rather it is passed as is to the next step
real_time_indicator	Indicates whether the communication event is happening in real-time and consequently an immediate response is expected. When this flag is set, the SGA does not close the communication channel with the SGoVApp but it immediately forwards the message to the appropriate IIG and returns the result to the calling SGoVApp
persistence_indicator	Indicates whether the message should persist in case of communication errors or other abruptions and retransmitted later. If this flag is set, message is stored in the Pending Actions Queue.

A message in the above format basically means (to the SGA): "Using the request identified as `request_id`, forward `<XML_message>` to the service `service_name`"

Table 1 Message format that a SGoVApp sends to an SGA

2.5.3.2.SGA: Handling SGoVApp requests

Upon receiving of such a message the SGA decides what to do with it. Its decision is governed by auxiliary configuration data that bind the named `service_name` to actual form of communication and actual IIG. These auxiliary configuration data may be manifested through a variety of means. For example, they can be hard-coded in to the SGA; they could reside into an external configuration file or in a database, etc. The actual form will be selected in the development phase of the SmartGov platform. These auxiliary configuration data are in fact a set of lookup tables containing the information presented in Table 2, Table 3 Table 4.

SGA Services Configuration File Contains tuples of the form:	<pre> < service_name, IIG_name, ordered_list_of_methods > </pre>
where,	
<code>service_name</code>	A symbolic name. The SGA accepts requests for communicate only with services declared here.
<code>IIG_name</code>	A symbolic name referring to the actual IIG.

<p><code>order_list_of_methods</code></p>	<p>An ordered list of communication methods that can be used to transmit data to designated IIG. Methods are referred to with symbolic names.</p> <p>A default fallback method</p> <p><code><store_to_local_data_store></code> is defined for all services and if all else fails the data to be transmitted are stored in the local repository for later processing. We call this repository the Pending Actions Queue¹ (PAQ). Messages with <code>real_time_indicator</code> set are not stored to the PAQ unless the <code>persistence_indicator</code> is also set. When these messages are stored to the PAQ the <code>real_time_indicator</code> is cleared, since obviously the real-time restrictions does not hold anymore.</p>
<p>The semantics of these configuration tuples are: "The symbolic name <code>service_name</code> corresponds to the IIG specified by <code>IIG_name</code> and use one of the <code>order_list_of_methods</code> to communicate with it. Try the <code>ordered_list_of_methods</code> sequentially, until one succeeds."</p> <p>The symbolic name <code>service_name</code> reflects a certain business operation. There may be more than one tuple for the same <code>service_name</code>, with different symbolic <code>IIG_names</code>. The existence of such tuples means that the same <code>service_name</code> is offered by all of the listed IIGs. As before, the selection algorithm depends on the first found IIG that can complete the communication.</p>	

Table 2 Format of the SGA services configuration file

To fully resolve symbolic names referenced in the SGA Services Configuration Files, the SGA uses the following configuration tables:

<p>SGA Communication Methods Configuration File</p> <p>Contains tuples of the form:</p>	<pre>< method, <more_info> ></pre>
---	--

¹ Another Pending Actions Queue is associated with the IIG. Although the same name will be used, these constructs are different. The acronym PAQ will be used for referring to such a queue. Where there is a need to refer to a specific queue we will use the acronyms SGA-PAQ and IIG-PAQ accordingly.

where,	
method	A symbolic name referring to the actual communication method employed.
<more_info>	All physical level information required for implementing the said method. For example, a URL may be specified in case of a Web-Service, a filename in case of off-line transmission of data, required credentials for connecting to a database, etc.
<p>The semantics of these configuration tuples are: "To implement the communication method <code>method</code> use the <code><more_info></code> information." The actual data of the <code><more_info></code> part will be determined at the development phase.</p> <p>A special method <code>store_to_local_data_store</code> is always defined and contains all needed information to store data received to a local data repository. The actual implementation depends upon operating requirements. The SGA should support as a means of storage at least storing to a file in the local machine and storing in a database.</p> <p>Obviously, the SGA supports only the methods that it knows about. This knowledge is hard-coded in the software modules that implement it.</p>	

Table 3 Format of the SGA Communication Methods configuration file

SGA IIG Configuration File Contains tuples of the form:	< IIG_name, <more_info> >
where,	
IIG_name	A symbolic name referring to the actual IIG.
<more_info>	All physical level information required for initiating communication with designated IIG. For example, it may contain the DNS name, the IP address and port the IIG listens to.

The semantics of these configuration tuples are as follows: "The symbolic name IIG_name corresponds to the specific IIG specified by the <more_info> part". The actual data of the <more_info> part will be determined at the development phase.

This mapping between symbolic IIG names and actual manifestations is maintained by the administrator of the SmartGov Platform, in cooperation with the administrators of the IIG.

Table 4 The SGA IIG Configuration File

SGA Notification Interceptor Configuration file Contains tuples of the form:	< notification_name, < PAQ_entry > >
where,	
notification_name	A symbolic name referring to a notification event that may be received from an IIG.
< PAQ_entry >	The entry that will be inserted in the SGA PAQ as a response to the reception of the designated notification.
The semantics of these configuration tuples are as follows: "when the notification notification_name is received, the PAQ_entry will be inserted into the SGA-PAQ". This mapping between symbolic notification names and PAQ entries is maintained by the administrator of the SmartGov Platform.	

Table 5 The SGA Notification Interceptor Configuration File

A diagram depicting discussed modules of the SGA is shown in Figure 55.

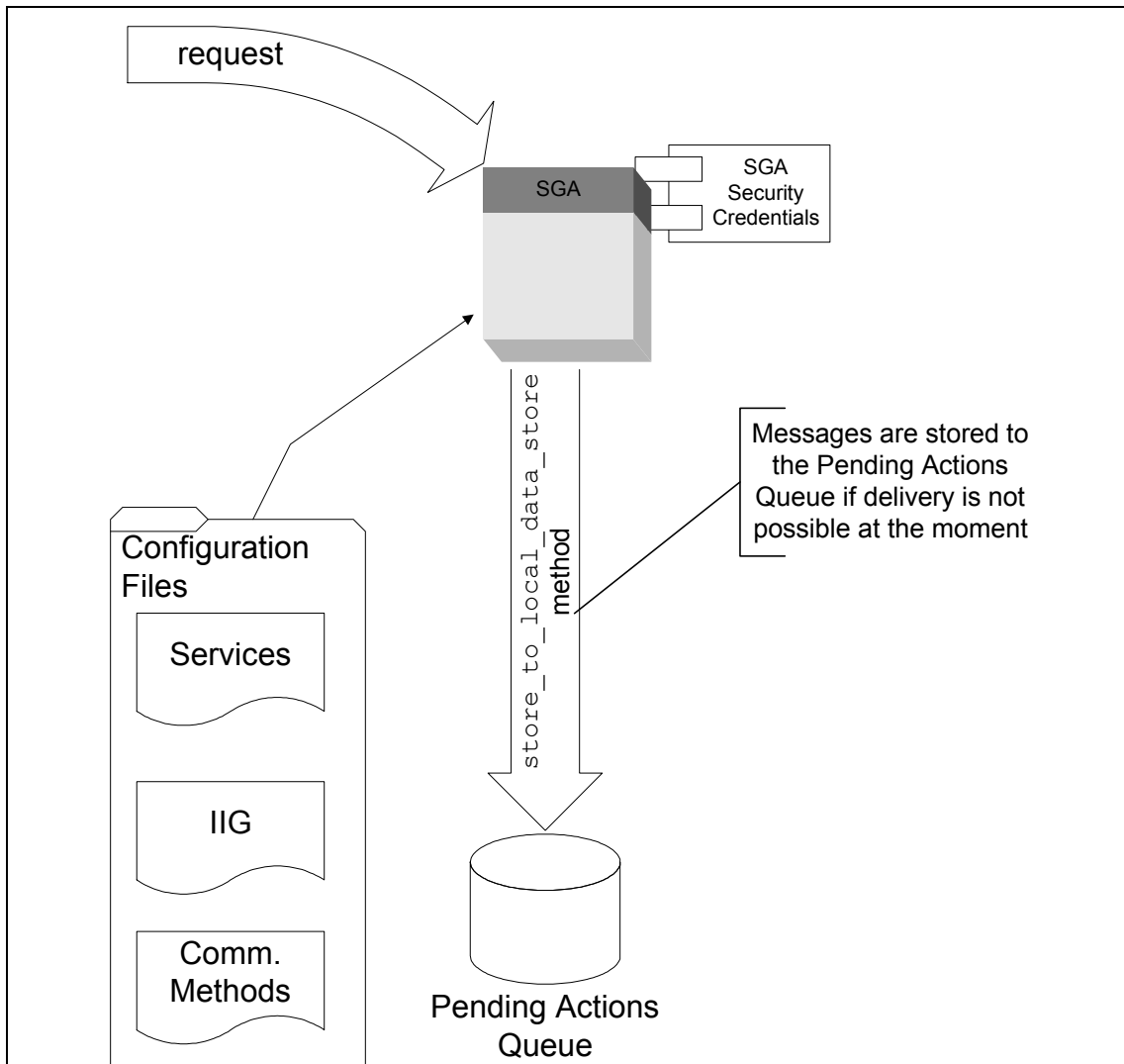


Figure 55 SGA modules

For each message that is received, an XML envelope is created comprising of the original <XML_message> and the SGA Security Credentials. This new XML message is forwarded to the IIG specified choosing one of the appropriate communication methods.

2.5.3.2.3 Messages forwarded by SGA to the IIG

When all relevant configuration data has been collected, the SGA is ready to transmit data to the IIG. For this purpose an XML message like the one specified in Table 6.

SGA to IIG messages Contains tuples of the form:	<pre> < service_name, <XML_message>, real_time_indicator <SGA_credentials> > </pre>
where,	
service_name	A symbolic name referring to a business operation. It should be the same name used in the initial request
<XML_message>	The same as the <XML_message> part of the original request.
real_time_indicator	The same meaning and value as in original request. The SGA expects to receive "immediate" answer from the IIG. The IIG should process the message as soon as it receives it and reply accordingly.
<SGA_credentials>	A set of credentials, e.g. the public key of the SGA, to prove the identity of the SGA that sends the message
The semantics of these tuples are perceived by the IIG as follows: "In <XML_message> are the data required to perform the operation implied by service_name. The proof of identity of the request originator may be found in <SGA_credentials>"	

Table 6 SGA to IIG messages

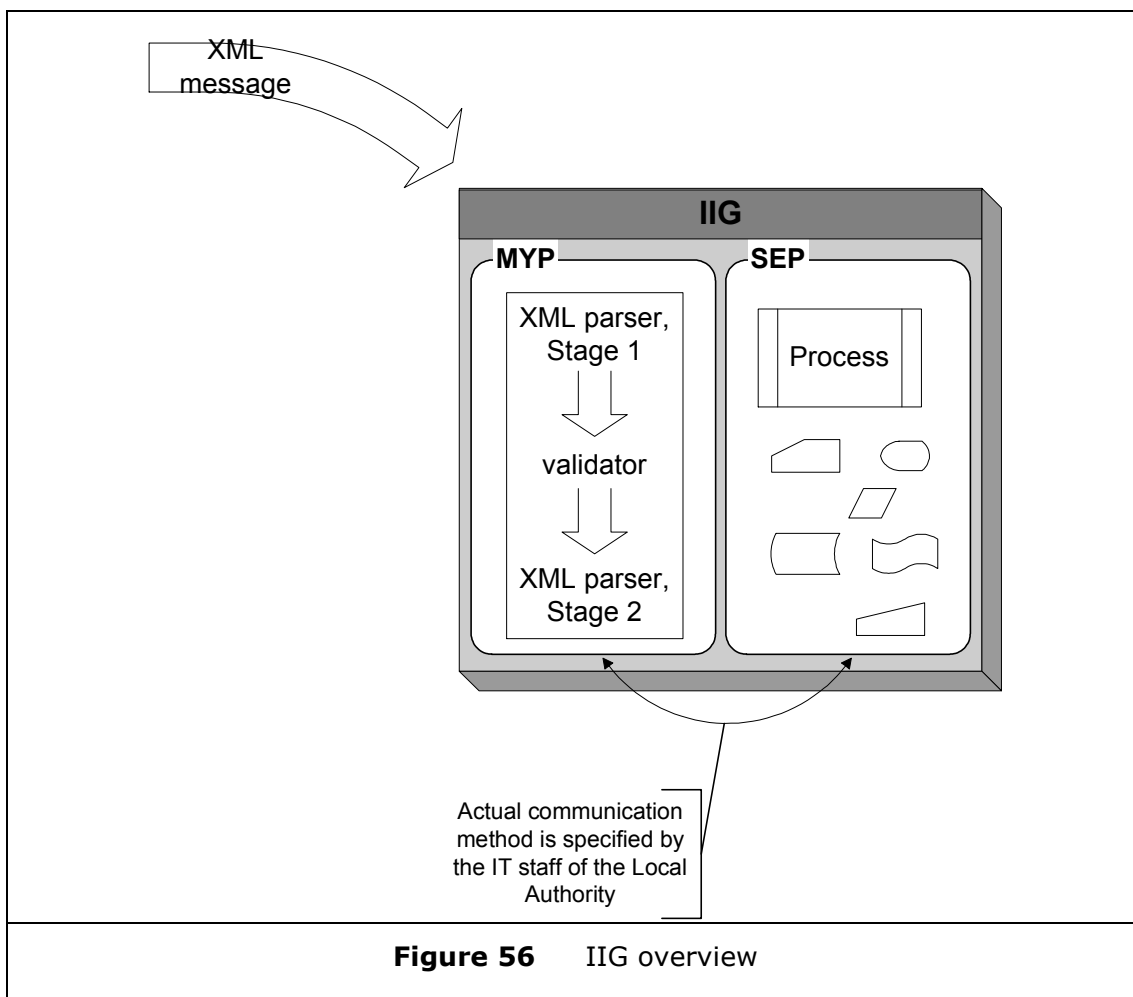
2.5.3.2.4IIG: Handling SGA messages

When the IIG receives a message the following actions are performed:

1. The message is assumed to be in XML format, so the IIG parses it and validates it against the XML schema used for this message exchange.
2. The <SGA_credentials> are checked to verify that the originator of the message is valid.
3. The symbolic service_name is checked to verify that this IIG supports the service specified.
4. If the above criteria are met, the <XML_message> part of the message is parsed and forwarded for further processing

In the above sequence of events two modules can be identified. The first module, which we will call MYP (Minimal Yoking Processor), is responsible for the parsing of the XML message the IIG receives, for checking the validity of the contained data and passing any data found to the second module. The second module, which we will call SEP (Separate External Process), receives the data that the IIG-MYP has prepared and performs any actual processing required. The SmartGov Project provides the IIG-MYP, either the actual implementation or a reference one², while the IIG-SEP is developed and maintained by the IT staff of the Local Authority.

The IIG, as specified so far, is depicted in the Figure 56.



² The SmartGov project will provide an actual working IIG-MYP implementation in accordance with the overall development environment that will be selected for the whole project. In the case that this implementation is not acceptable by the IT staff of the Local Authority, the SmartGov project will provide assistance in the form of detailed pseudo code. The transformation of this pseudo code to an actual software module is left to the IT staff of the LA.

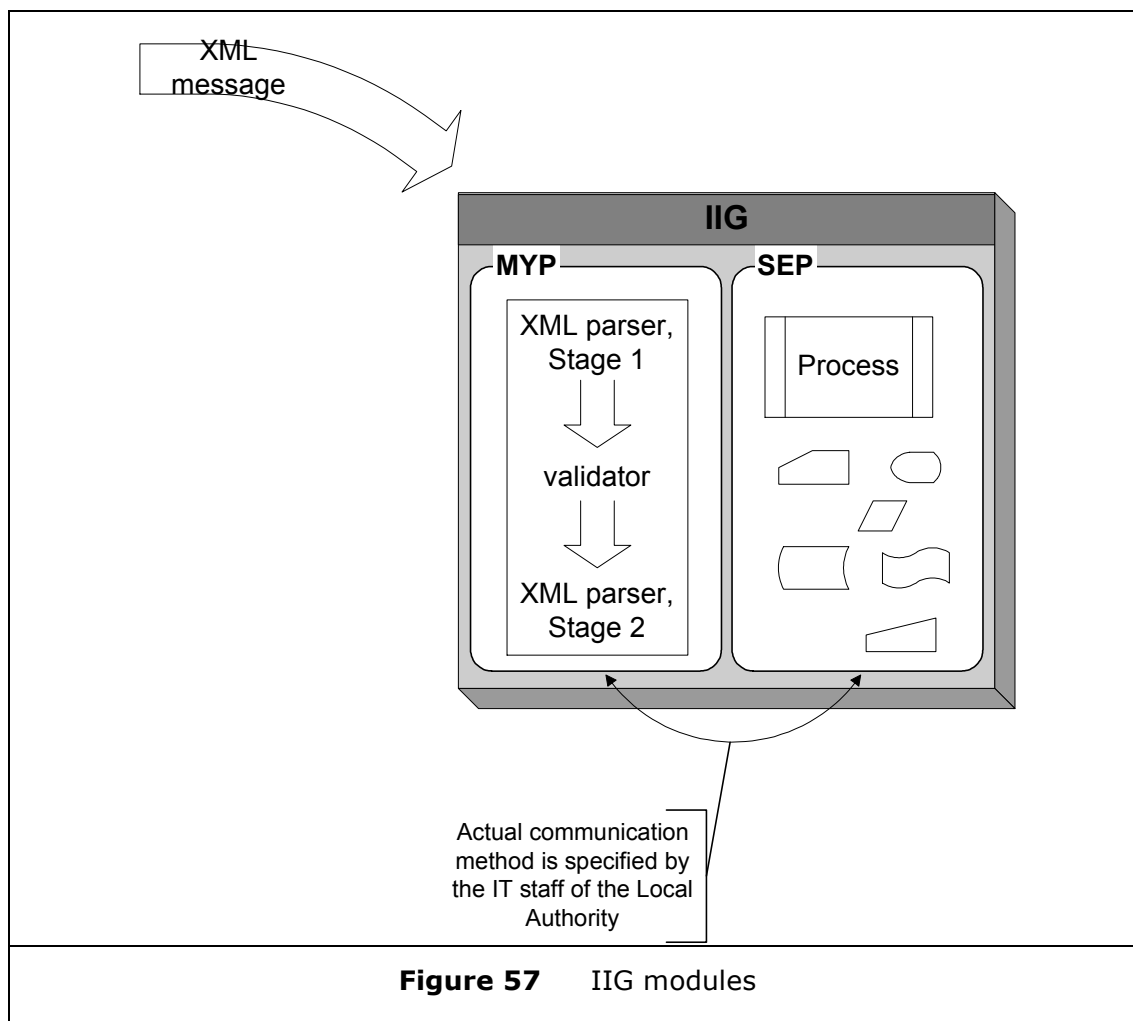
2.5.3.2.4.1 IIG: Handling Synchronous (real-time) requests

The IIG needs to reply in real-time in requests that have the `real_time_indicator` set. In the scope of the SmartGov Project "real time" has the meaning of "as soon as possible". Real-time events are expected when there is direct user interaction with a SGoVApp. For example, a user requesting a form provided by a SGoVApp that must be pre-filled with some data. In such a case the SGoVApp will initiate a real-time request to an SGA to fetch these data. The request must be serviced immediately as the user is waiting *now* for the completion of the operation to continue filling in the form. The SGA will forward the request to the appropriate IIG and the later should return the results as soon as possible. Furthermore, in scenarios where real-time requests appear it is assumed that there is an available network connection, preferably low-latency high-speed, between all involved parties, namely the SGoVApp, the SGA, the IIG-MYP and IIG-SEP. Some of these modules may run on the same machine, in which case the requirements for high speed and low latency are met. In all other cases, the `real_time_indicator` is not set and no specific timing restrictions are imposed upon the messages.

2.5.3.2.4.2 IIG: Handling Asynchronous (non-real-time) requests

When a non-real-time message is received, that is a message where the `real_time_indicator` is not set, it is stored in a Pending Actions Queue (PAQ). Storage takes place as soon as the message reaches the IIG and after the first Stage of the IIG-MYP has been completed and therefore it has been identified that it is not a real-time message. This way, the communication channel is freed as soon as possible and the IIG is free to process urgent, i.e. real-time, messages. Another advantage of this approach is that further processing of messages in the PAQ can be aligned with local IT system policies.

Figure 57 illustrates the IIG modules discussed so far.

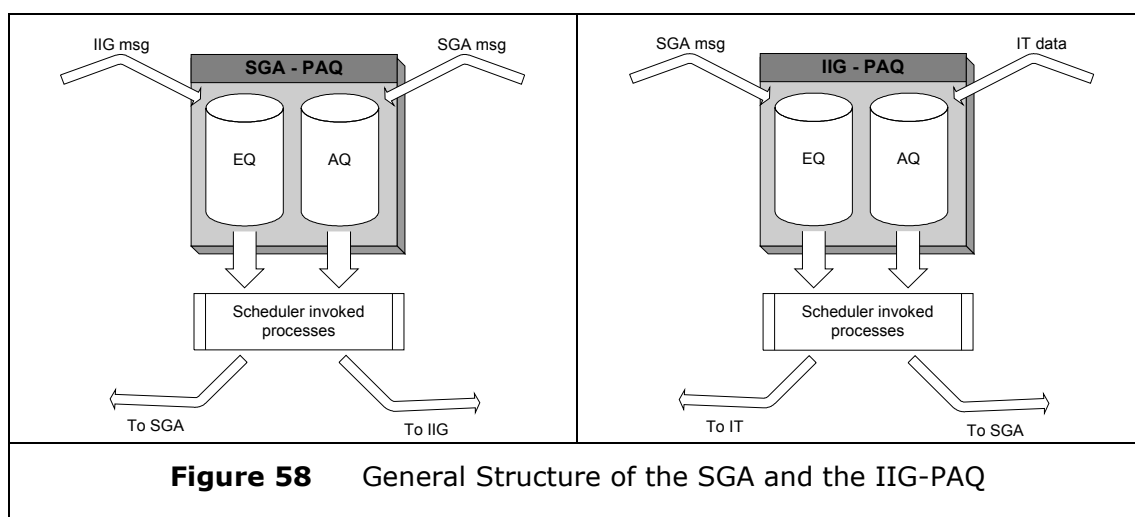


2.5.3.2.5 The Pending Actions Queue

The Pending Actions Queue (PAQ) serves as a temporary repository for storing messages in cases where immediate processing is not possible or desirable. Both the SGA-PAQ and the IIG-PAQ have similar structure and undergo similar processing but reside in different physical locations. The SGA-PAQ resides within the SmartGov Platform while the IIG-PAQ resides within the 3rd party IT system. Processing of the messages stored in the PAQ is done through a scheduler which is executed by the facilities provided by the underlying operating environment, e.g. `cron`-scripts in a Unix environment or `at` initiated processes in an Windows environment. Each PAQ is further partitioned into two separate queues. The Entra-Queue (EQ) and the Adelante-Queue (AQ)³. Each queue is characterized as

³ The Entra-Queue and the Adelante-Queue are the Incoming and Outgoing queues respectively.

entra or adelante regarding the PAQ⁴ it refers to. Consequently, the messages forwarded from the SGA-APAQ are inserted into the IIG-EPAQ and vice versa. In Figure 58 a general structure of a PAQ is depicted.



The periodicity of the scheduler invoked processes will be decided on the actual implementation environment taking into account any peculiarities and constraints placed by the actual working systems. Anyway, it is envisaged that the period of invocation will be variant, ranging from minutes to days, depending upon the processing requirements of each message class. For example, messages related to warehouse stock updating maybe processed in 10 minutes interval while messages related to certificate applications could be processed in a daily basis. The message formats for the SGA-APAQ and the IIG-EPAQ have already been discussed in paragraphs 2.5.3.2.1 and 2.5.3.2.3 respectively. The messages referring to the SGA-EPAQ and IIG-APAQ will be discussed in the following paragraphs.

2.5.3.2.6 IIG-The Pending Actions Queue

When the 3rd party IT system has finished processing of the messages forwarded by the IIG, the results are returned to the calling IIG. The format of the result is dependent on the 3rd party IT system and it is the responsibility of the IIG to re-format them into XML messages and present them to the next layer.

Depending on the nature of the initial request results are either forwarded immediately to the caller (real-time requests) or stored in the PAQ, in this case the IIG-APAQ. In the latter case, only a notification may be returned.

⁴ We will use the following notation to refer to various queues. The notation EPAQ refers to the EQ of the PAQ, while the notation APAQ refers to the AQ of the PAQ.

In 2.5.3.2.4.2 it is stated that the IT staff of the organisation dictates the actual communication method between the IIG and the 3rd party IT system. It is assumed that this communication method provides for an indication of the service on behalf of which the processing was done and thus the symbolic service name for which the processing was done is known to the IIG. This symbolic service name corresponds to the one used in the messages described in 2.5.3.2.3.

It must be noted that it is absolutely valid and very much possible for the 3rd party IT system to generate messages that do have not been triggered by a request from any SGoVApp. These kind of messages can, for example, happen every time there is a need of updating data residing in the SmartGov Platform with "fresh" data from the IT system. In these cases, notifications for the SmartGov platform will be produced, which, when delivered to the SmartGov platform, will trigger the appropriate action.

2.5.3.2.7IIG: Handling replies from 3rd party IT systems

The actual data that the 3rd party IT system returns are not interpreted by the IIG in anyway. They are just re-packaged into appropriate XML format and forwarded to the relevant SGA. When the results enter the IIG, two facts are known. The symbolic service name the results refer to and the actual results. These two facts form the basis of the XML message that is formulated according to Table 7.

<p>IIG to SGA message format Contains tuples of the form:</p>	<pre>< reply_id service_name, <results_in_XML>, real_time_indicator <IIG_credentials> ></pre>
<p>where,</p>	
<p>Reply_id</p>	<p>A unique reply identifier that serves to characterize this reply</p>
<p>Service_name</p>	<p>A symbolic name referring to a business operation. It should be the same name used in the initial request</p>
<p><results_in_XML></p>	<p>The results of the processing done by the IT system are encapsulated verbatim in an XML message. Only an XML wrapper is placed around them to facilitate communications</p>

Real_time_indicator	<p>The same meaning and value as in original request.</p> <p>The IIG signifies that this is a response to an initial real-time request.</p>
<p>A single tuple basically means: "In the <results_in_XML> are the results of the processing required by the service service_name. You, the IIG as verified by the <IIG_credentials>, can now forward this message as required by the service_name".</p> <p>A persistence indicator is not required here as it is assumed that all replies are either immediately forwarded or stored into the IIG-APAQ to be forwarded at a more convenient time.</p>	

Table 7 Format of the reply of the 3rd party IT system as re-formulated by the IIG-MYP

This message is formulated by the IIG-MYP and not by the 3rd party IT system processes.

The IIG, in order to forward this message, consults external configuration files that are provided by the same methods as the SGA configuration files. In analogous to the SGA configuration files, the IIG configuration files needed are:

- The IIG services configuration file
- The IIG communication methods configuration file
- The IIG SEP configuration file

The contents of these are presented in the following tables.

<p>IIG Services Configuration File</p> <p>Contains tuples of the form:</p>	<pre>< service_name, SEP_name, ordered_list_of_methods ></pre>
<p>where,</p>	
<p>service_name</p>	<p>A symbolic name. The IIG accepts requests to execute only services declared here.</p>
<p>SEP_name</p>	<p>A symbolic name referring to the actual SEP that will be executed as a response to the request.</p>

<p><code>order_list_of_methods</code></p>	<p>An ordered list of communication methods that can be used to transmit data to designated SEP. Methods are referred to with symbolic names.</p> <p>A default fallback method <code><store_to_local_data_store></code> is defined for all services and if all else fails the data to be transmitted are stored in the local repository for later processing. We call this repository the Pending Actions Queue (PAQ). Messages with <code>real_time_indicator</code> set are not stored to the PAQ unless the <code>persistence_indicator</code> is also set. When these messages are stored to the PAQ the <code>real_time_indicator</code> is cleared, since obviously the real-time restrictions does not hold anymore.</p>
<p>The semantics of these configuration tuples are: "The symbolic name <code>service_name</code> will be serviced by the SEP named <code>IIG_name</code>; this will be invoked using one of the <code>order_list_of_methods</code>, which will be tried sequentially, until one succeeds."</p> <p>The symbolic name <code>service_name</code> reflects a certain business operation. There may be more than one tuple for the same <code>service_name</code>, with different symbolic <code>SEP_names</code>. The existence of such tuples means that the same <code>service_name</code> is implemented by all of the listed IIGs. As before, the selection algorithm depends on the first found SEP that can be invoked.</p>	

Table 8 Format of the IIG Services configuration file

To fully resolve symbolic names referenced in the IIG Services Configuration Files, the IIG uses the following configuration tables:

<p>IIG Communication Methods Configuration File</p> <p>Contains tuples of the form:</p>	<pre>< method, <more_info> ></pre>
<p>where,</p>	
<p><code>method</code></p>	<p>A symbolic name referring to the actual communication method employed.</p>

<more_info>	All physical level information required for implementing the said method. For example, a URL may be specified in case of a Web-Service, a filename in case of off-line transmission of data, required credentials for connecting to a database, etc.
The semantics of these configuration tuples are: "To implement the communication method <code>method</code> use the <more_info> information." The actual data of the <more_info> part will be determined at the development phase. Obviously, the IIG supports only the methods that it knows about. This knowledge is hard-coded in the software modules that implement it.	

Table 9 Format of the IIG Communication Methods configuration file

IIG SEP Configuration File Contains tuples of the form:	< SEP_name, <more_info> >
where,	
IIG_name	A symbolic name referring to the actual IIG.
<more_info>	All physical level information required for initiating the designated SEP. For example, it may contain the path to the executable program and parameters that need to be passed to it.
The semantics of these configuration tuples are as follows: "The symbolic name <code>SEP_name</code> corresponds to the specific program specified by the <more_info> part" The actual data of the <more_info> part will be determined at the development phase. This mapping between symbolic SEP names and actual manifestations is maintained by the administrator of the SmartGov Platform in cooperation with the system administrators.	

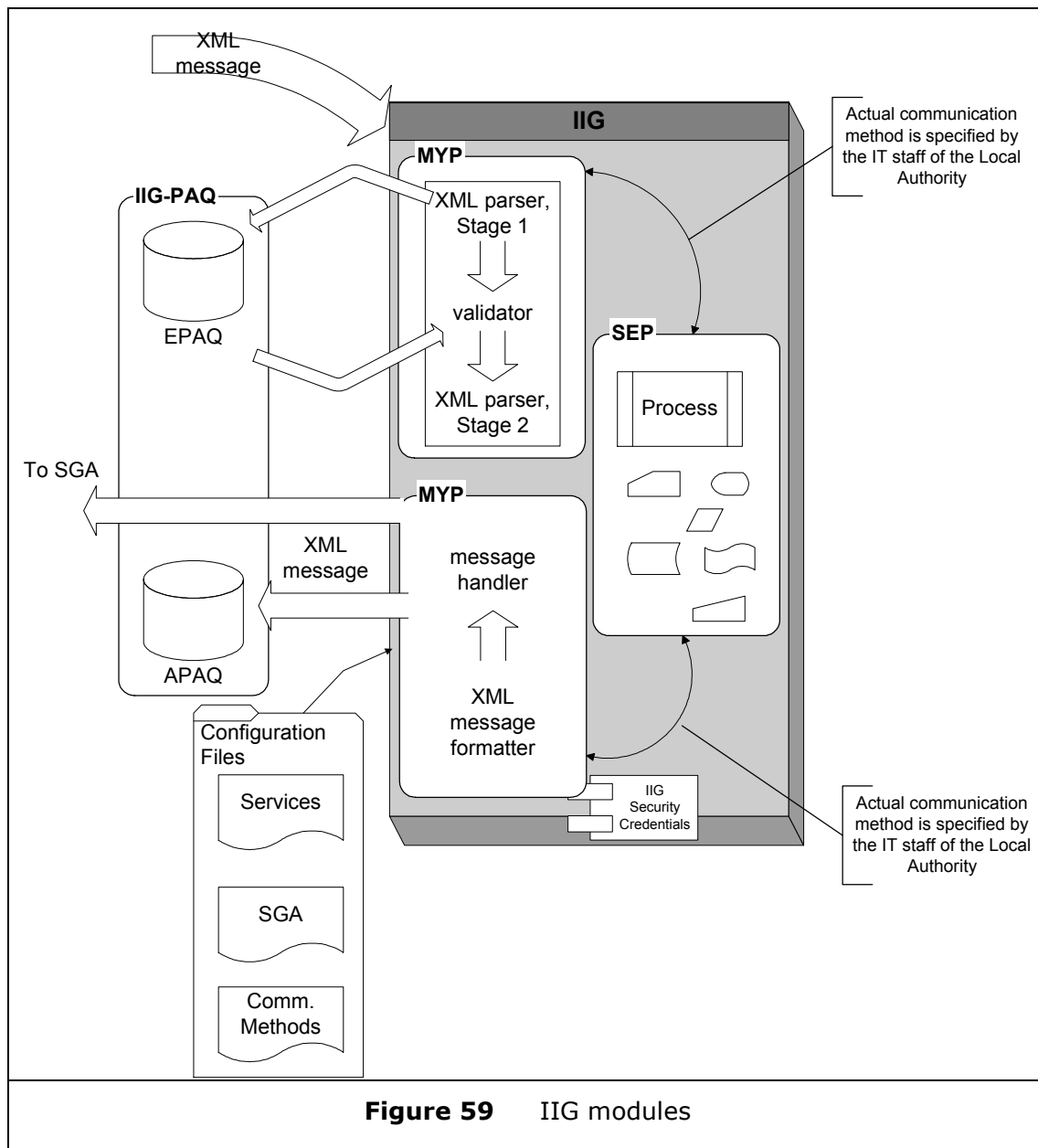
Table 10 The IIG SEP Configuration File

IIG Notification Initiator Configuration File Contains tuples of the form:	< notification_name, <SGA_NI_info> >
---	---

where,	
notification_name	A symbolic name for the notification event.
< SGA_NI_info >	All information required for connecting to the SGA notification interceptor and delivering the requested notification. For example, it may contain the IP address and the port to connect to.
<p>The semantics of these configuration tuples are as follows: "In order to deliver the notification notification_name, a connection to the SGA-NI designated in the SGA_NI_info should be established". The actual data of the < SGA_NI_info > part will be determined at the development phase.</p> <p>This mapping between symbolic notification names and the SGA NI information is maintained by the IIG administrators.</p>	

Table 11 The IIG Notification Initiator Configuration File

The IIG modules defined are depicted in Figure 59.



2.5.3.2.8SGA: Handling IIG messages

When an SGA receives a message from an IIG, the following actions are performed:

- The message is assumed to be in XML format, so the SGA parses it
- The `<IIG_credentials>` are checked to verify that the originator of the message is a known and valid IIG
- The symbolic name `service_name` is checked to verify that this SGA handles messages for this service.
- If the above criteria are met, the XML message received is parsed and the `<results_in_XML>` as well as the `service_name` are stored to the SGA-EPAQ.

Following the proposal of the IIG, a Minimal Yoking Processor (MYP) is also required here. This SGA-MYP is responsible for the initial parsing of the XML reply, the verification of the IIG credentials and the `service_name` suitability requirement.

Messages received by an SGA that originate from an IIG are always stored in the SGA-EPAQ. In the SmartGov Platform there is no provision for a specific process that listens to messages. Therefore the extraction of the messages stored in the EPAQ is the responsibility of processes invoked by the operating system's scheduler.

2.5.3.2.9 Delivery of messages to the appropriate SGoVApp

In the case of real-time responses the application that made the original request receives and handles the `<results_in_XML>`. Error conditions are specific to the application and may require further communication. This communication is handled in the same way as the normal one, exploiting the facilities offered by the SGA and the IIG.

In all other cases, non-real-time messages, scheduler controlled processes are periodically invoked to scan the SGA-EPAQ for messages. When such a message is found, the `service_name` is checked and the relevant SGoVApp is called to handle the `<results_in_XML>` part of the message. It is the responsibility of this SGoVApp to parse the `<results_in_XML>` and to make any necessary integrity checks. This application is not necessarily the same with the one that made the original request. It is left to the implementation phase to decide whether the producer and the consumer of the messages are one or separate applications. It is envisaged, that replies that are not characterized as real-time will eventually be stored in a local data repository for easy access to other SGoVApps.

2.5.3.2.10 Activity logging and statistics computation

For reasons of activity traceability and computation of statistical figures, the SmartGov Agent and the Information Interchange gateway modules maintain log files, in which information regarding the ongoing activities is recorded. The log files maintained and the information recorded in these, are presented in the following paragraphs.

2.5.3.2.10.1 The service delivery environment log files

Within the service delivery environment, the SGA, the SGA-PAQUED and the SGA-NI add entries to the log files, recording information regarding the following events:

- *Submission of a request.* For this event, the request id, the service name requested, the real time and persistence indicators and the timestamp of the request are recorded in the log file.
- *Method invocation.* When a request is received by the SGA, the corresponding methods specified in its configuration files are tried, in order to serve the request. For each such attempt, the id of the request being served, the name and address of the peer system, the invoked service name, the result status of the communication (success or failure) and the time taken to complete the attempt (begin and end timestamps) are recorded in the log file.
- *Initiation of the SGA-PAQUED execution.* When a fresh run of the SGA-PAQUED is initiated, the corresponding event and the current timestamp are recorded in the log file. For queue size profiling purposes, the number of entries in the SGA-PAQ is also written in the log file entry.
- *Process spawning.* As a result of processing an SGA-PAQ entry, the SGA-PAQUED may spawn a new lightweight or heavyweight process. For these events, the SGA-PAQUED records the request id (extracted from the SGA-PAQ entry), an identifier for the spawned process and the current timestamp. We note here that the SGA-PAQUED is not able to monitor the execution progress or the final termination status of these processes, thus it is not possible for the SGA-PAQUED to log information regarding the success, failure, or other details of the spawned process. However, the SmartGov platform provides a library and an associated API that enable the spawned processes to record any relevant information in the log file.
- *Termination of the SGA-PAQUED execution.* When the SGA-PAQUED has examined all entries in the SGA-PAQ it terminates its execution, recording in the log file the corresponding event and the current timestamp. For queue size profiling purposes, the number of entries in the SGA-PAQ is also written in the log file entry.
- *Reception of a notification.* Upon receipt of a notification, the SGA-NI records in the log file the notification identification, the peer system name and address and the current timestamp.

2.5.3.2.10.2 *The organisational information system environment log files*

Within the organizational information system environment, the IIG, the IIG-PAQUED and the IIG-NI add entries to the log files, recording information regarding the following events:

- *Reception of a request.* For this event, the request id, the service name requested, the real time indicator and the timestamp of the request are recorded in the log file.
- *Completion of a request execution.* When the IIG receives a request having the real time indicator set to *true*, it spawns an IIG-SEP and collects the reply. When the reply is collected, the request identifier and the current timestamp are recorded in the log file.
- *Initiation of the IIG-PAQUED execution.* When the IIG-PAQUED is run afresh, the corresponding event and the current timestamp are recorded in the log file. For queue size profiling purposes, the number of entries in the IIG-PAQ is also written in the log file entry.
- *Process spawning.* As a result of processing an IIG-PAQ entry, the IIG-PAQUED may spawn a new lightweight or heavyweight process. For these events, the IIG-PAQUED records the request id (extracted from the IIG-PAQ entry), an identifier for the spawned process and the current timestamp. We note here that the IIG-PAQUED is not able to monitor the execution progress or the final termination status of these processes, thus it is not possible for the IIG-PAQUED to log information regarding the success, failure, or other details of the spawned process. However, the SmartGov platform provides a library and an associated API that enable the spawned processes to record any relevant information in the log file.
- *Termination of the IIG-PAQUED execution.* When the IIG-PAQUED has examined all entries in the IIG-PAQ it terminates its execution, recording in the log file the corresponding event and the current timestamp. For queue size profiling purposes, the number of entries in the IIG-PAQ is also written in the log file entry.
- *Posting of a notification.* When a notification is posted, the IIG-NI records in the log file the notification identification, the peer system name and address and the current timestamp. This logging facility is integrated in the IIG-NI library, thus the invoking application need not take any extra actions for this purpose.

2.5.3.2.10.3 The logging requests

A logging request posted by any application to the SmartGov libraries responsible for implementing the logging process has the following format:

Logging message:	< eventCriticality message >
where,	
eventCriticality	This parameter characterises the associated message in terms of its severity. It may be set to the following values: SG_LOG_EMERG (an event requiring immediate attention), SG_LOG_ALERT (an event requiring attention), SG_LOG_ERROR (an error condition), SG_LOG_WARNING (a warning message) SG_LOG_INFO (an informational message) and SG_LOG_DEBUG (IT staff use for debugging purposes).
message	A string that describes the event
The entries in the log file are supplemented with the current timestamp (as perceived by the machine the logger process is run on) and the address of the machine the message has originated from.	

The libraries responsible for system logging must be able to locate the communication details for contacting the logger process. To this end, a configuration file is maintained on each system that logging requests may originate from, containing these communication details. This configuration file should reside on the directory pointed to by the `SG_LOG_CONFIG` environment variable.

2.5.3.2.11 Global and Service-Specific Context

While the SmartGov agent communicates with the third-party IT systems in a manner that is independent of (a) global parameters (e.g. the user that has logged in, the strength of the authentication method etc) and (b) service-specific parameters, such as the name of the service or the TSEs participating in it, it is clear that the contents of the messages exchanged with the organisational or third party IT systems will contain both global and service-specific context. For instance, a request to retrieve the service user's personal details from a registry should contain some form of identification for the user (user name, user id etc), which falls into the global context, whereas various elements of the reply should be assigned as values to service-specific TSEs. These details are transparent to

the SmartGov Agent and the Information Interchange Gateway, and are encapsulated in the XML messages that are transferred as request parameters or received as replies. The software entities that invoke the SmartGov Agent are responsible for:

1. formulating appropriately XML message that should be passed to the invoked service as a parameter, and embed in it any pertinent global or service-specific context.
2. extracting data from reply messages and modifying accordingly global or service specific context (e.g. assigning values to TSEs).

2.5.3.3 Remote administration of the IIG

The IIG should provide facilities for remote administration, since access to the premises and/or computer systems of the organisational IT systems may be restricted. Since the operation of the IIG is fully controllable by configuration files, it suffices to provide interfaces for the remote management of the content of these configuration files. The management operations are built in the IIG-MYP, which runs continuously on the environment of the organisational information system, thus they are always available to the administrators. Access to these operations is protected by authentication credentials and may be further restricted through the use of firewalls or any other appropriate technology. The administration interface offered by the IIG-MYP is detailed in the following paragraphs.

2.5.3.3.1 Administering the IIG services configuration file

The IIG provides the following management functionality for the IIG services configuration file:

- `IIG_query_services(void)`. This method accepts no parameters and returns a list of `<service_name, SEP_name, ordered_list_of_methods>` triples.
- `IIG_add_service(service_name, SEP_name, ordered_method_list)`. This method adds a new service to the IIG services configuration file, as designated by the method parameters. The service name must not exist in the IIG services configuration; additionally, the `SEP_name` and each of the methods listed in the `ordered_method_list` should be defined in the respective configuration file.
- `IIG_modify_service(service_name, SEP_name, ordered_method_list)`. This method modifies the service `service_name`

to map to the `SEP_name` which will be invoked using the methods dictated in the `ordered_method_list`. The `service_name` must exist in the IIG services configuration and the `SEP_name` and each of the methods listed in the `ordered_method_list` should be defined in the respective configuration file.

- `IIG_remove_service(service_name)`. This method deletes the designated service from the IIG services configuration file. The service name must be defined in the IIG services configuration file.

2.5.3.3.2 Administering the IIG SEP configuration file

The IIG provides the following management functionality for the IIG SEP configuration file:

- `IIG_query_SEP(void)`. This method accepts no parameters and returns a list of `<SEP_name, <more_info>>` pairs.
- `IIG_add_SEP(SEP_name, more_info)`. This method adds a new SEP name to the IIG SEP configuration file, as designated by the method parameters. The SEP name must not exist in the IIG SEP configuration.
- `IIG_modify_SEP(SEP_name, <more_info>)`. This method modifies the SEP `SEP_name` to map to the information specified in `<more_info>`. The `SEP_name` must exist in the IIG SEP configuration.
- `IIG_remove_SEP(SEP_name)`. This method deletes the designated SEP from the IIG SEP configuration file. The SEP name must be defined in the IIG SEP configuration file.

2.5.3.3.3 Administering the IIG Methods configuration file

The IIG provides the following management functionality for the IIG Methods configuration file:

- `IIG_query_methods(void)`. This service accepts no parameters and returns a list of `<method_name, <more_info>>` pairs.
- `IIG_add_method(method_name, more_info)`. This service adds a new method name to the IIG method configuration file, as designated by the service parameters. The method name must not exist in the IIG method configuration.
- `IIG_modify_method(method_name, <more_info>)`. This service modifies the method `method_name` to map to the information specified in `<more_info>`. The `method_name` must exist in the IIG methods configuration.

- `IIG_remove_method(method_name)`. This service deletes the designated method from the IIG method configuration file. The method name must be defined in the IIG methods configuration file.

2.5.3.3.4 Administering the IIG Notification Initiator configuration file

The IIG provides the following management functionality for the IIG Notification Initiator configuration file:

- `IIG_query_notifications(void)`. This service accepts no parameters and returns a list of `<notification_name, <SGA_NI_info>>` pairs.
- `IIG_add_notification(notification_name, SGA_NI_info)`. This service adds a new notification name to the IIG notification initiator configuration file, as designated by the service parameters. The notification name must not exist in the IIG notification initiator configuration.
- `IIG_modify_notification(notification_name, <SGA_NI_info>)`. This service modifies the notification `notification_name` to post the notification events to the SGA-NI designated by `<SGA_NI_info>`. The `notification_name` must exist in the IIG notification initiator configuration.
- `IIG_remove_notification(notification_name)`. This service deletes the designated notification from the IIG notification initiator configuration file. The notification name must be defined in the IIG notification initiator configuration file.

2.5.3.3.5 Supplementary remote administration functionalities

In addition to the remote administration functionality described in paragraphs 2.5.3.3.1 to 2.5.3.3.4, the IIG provides the following two remote administration interfaces:

- `IIG_reload_configuration(void)`. Via this service the IIG is instructed to re-read its configuration files and modify its behaviour according to the new contents of these files.
- `IIG_run_PAQUED(void)`. Via this service an immediate execution of the IIG PAQUED is requested, so that the IIG PAQ entries are examined. The IIG reserves the right to ignore the request if the IIG PAQUED is already running or if system load is excessively high. In these cases, an appropriate indication is returned.

2.5.3.4 Summary

A generic method of communication between application developed within the SmartGov Platform and 3rd party IT systems has been described. It has been assumed that these 3rd party IT systems operate under the control of a Local Authority (LA) and that a direct communication link (network) would not always be available. Thus provisions have been made to accommodate for alternative method of communications, such as delayed forwarding (spooling) of messages, sneaker-net type communications (floppy disks or tapes), etc. These alternate methods can also be employed in cases where the 3rd party IT system is not ready to receive messages from SGoVApps.

Communication relies upon two modules, namely the SmartGov Agent (SGA) and the Information Interchange Gateway (IIG). These modules have been analyzed and their components identified. The format of the messages exchanged has also been specified.

Security mechanisms haven been specified as an integral part of the communication. These mechanisms rely on the existence and verification of appropriate credentials. The actual format of these credentials has not been specified yet, but it has been assumed that the whole scheme will rely on existent and tested approaches such as those used by the SSL and the SSH protocols.

Required operation system support has been identified in the form of the scheduler. The scheduler is responsible for periodically invoking the relevant processes to handle messages in various stages of communication. The local administrators are responsible for proper scheduler configuration.

2.6 Process View

2.6.1 The SmartGov Front-End

In Figure 60, a schematic design of the basic process in a web application is shown, customized for our application.

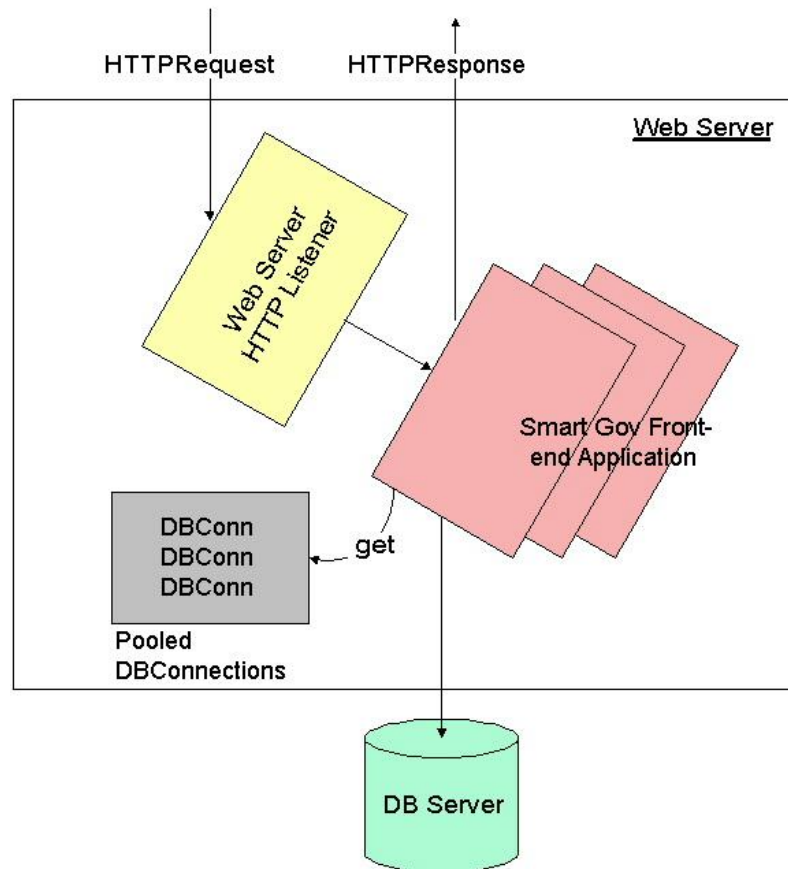


Figure 60 – SmartGov Front-end web application schema

The management of the processes required to reply the HTTP Request is responsibility of the web server. SmartGov front-end gives the 'logic', the intelligence of the application, but all the internal processes required are provided by the web server.

2.6.2 The Integrator

The integrator objects will live within the web container instance and thus all processes will be handled transparently by the container implementation. Accesses to the database server as well as the file system are also going to be managed by the container instance through the relevant objects (JDBC objects for the former and Java.io objects for the latter). The initiation of the process for the integrator will be realized through a relevant HTTP request as shown in Figure 60 for the case of the Front-end.

2.6.3 SmartGov Agent - Information Interchange Gateway

The SmartGov agent and the Information Interchange Gateway are the components of the SmartGov platform that cater for the communication between the service delivery environment and the organisation's installed IT systems or third-party IT systems external to the organisation. In the following paragraphs the process view for these two components is described, organised by the communication events. For each communication event, the involved software modules (as listed in section 2.5.3.1) are identified and the actions taken by each module are described.

2.6.3.1 Synchronous communication

The synchronous communication paradigm is employed when, in the context of delivering some service to a user, a request must be forwarded from the service delivery environment to an organisational or external IT system and this request returns results that are needed in order to continue with delivering the service. An example of such a request is the retrieval of the user's personal data from a registry, in order to be filled in the corresponding TSEs of a form. Process interaction in this case commences as follows:

1. SmartGovApps invoke the SGA, asking for the request to be carried out, by calling the appropriate Java class method of the SGA library. Parameters are passed using the Java language calling convention. The invocation is synchronous, therefore the execution of the invoking SmartGovApp is suspended until the method execution is finished. Method execution takes place within the control thread of the calling program.
2. The SGA assembles a message with all appropriate information needed for the completion of the request, including the parameters passed by the programs implementing service delivery and the pertinent configuration and authentication data. This message sent to the IIG-MYP component through a message passing mechanism. Execution of the SGA control thread (effectively, the execution of the service delivery instance) is suspended until the message is delivered and the reply collected, or a timeout expires with no reply having been returned, in which case an exception is raised.
3. The IIG-MYP receives the message and spawns a new control thread for handling the request. The new lightweight process will validate the received message and the authentication data, extract the parameters that are required for processing the request and spawn an IIG-SEP process that will actually run the code fulfilling the task at hand. This code may be Java code

(and thus be able to be called as a Java method), an operating system program (in which case a suitable system call must be performed etc). The IIG-SEP calling details should be specified by the organisation's IT staff. Communication between the IIG-MYP and the IIG-SEP spawner is performed through the Java language parameter passing mechanism in an asynchronous manner, i.e. once the IIG-SEP spawner is initialised it completely detaches from the IIG-MYP control thread and their execution continues independently thereafter.

4. The IIG-SEP process returns the reply to its IIG-SEP spawner using a suitable method; if the IIG-SEP process is a piece of Java code, results may be returned through the Java method calling mechanism; for external operating system processes pipes, sockets, intermediate files or any other appropriate programming technique may be used. The IIG-SEP spawner returns the results to the SGA through a message passing mechanism and terminates its execution.
5. The SGA, having received the reply, exits the suspended state and becomes runnable again. It extracts the reply from the received message and forwards it to the calling SGovApp service instance through the Java method calling mechanism, completing the synchronous communication cycle.

2.6.3.2 Asynchronous communication

The asynchronous communication paradigm is employed when, in the context of delivering some service to a user, a request must be forwarded from the service delivery environment to an organisational or external IT system but the results of this request –or even its actual execution– are not required in order to continue with the delivery of the service; the only requirement is that the request will be *eventually* carried out. Process interaction in this case commences as follows:

1. SmartGovApps invoke the SGA, asking for the request to be carried out, as described in item 1 of section 2.6.3.1.
2. The SGA assembles a message and passes it to the IIG-MYP as discussed in item 2 of section 2.6.3.1.
3. The IIG-MYP receives the message and arranges for inserting the appropriate information in the IIG-PAQ. Once this information has been inserted, a reply is returned to the SGA indicating that the message has been received.
4. The SGA, having received the reply, exits the suspended state and becomes runnable again, returning the control to the calling SGovApp service instance.

The actual execution of the request is, in this case, deferred until the corresponding IIG-PAQ entry is processed by the *IIG-PAQUED*.

2.6.3.3 Periodic events

The SmartGov architecture introduces two software modules that are run periodically in order to identify and execute tasks that have been deferred. These modules are the *SGA-Pending Action Queue Dispatcher (SGA-PAQUED)*, running within the service delivery environment and the *IIG-Pending Action Queue Dispatcher (IIG-PAQUED)*, running within the environment of the organisational information system. Each of these modules awakens periodically (periodicity is specified by the local IT staff) and scans through the corresponding action queue to locate entries for actions that may be executed. When such an action is found, a separate process is launched to actually execute the task at hand and the entry is removed from the PAQ. Standard operating system facilities are employed for parameter passing between the PAQUED and its child processes, while temporary files may be used in order to pass large volumes of information to the child processes. The child processes have no access to the internal structure or contents of the PAQ. Periodic activation of the PAQUED modules relies on the facilities offered by the underlying operating system for scheduled task execution.

2.6.3.4 Notifications

Processes running in the environment of the organisational information system may post *notification events* to the service delivery environments signifying that some action should be initiated (or may proceed) within the service delivery environment. Processes send the notification events by employing the *IIG-Notification Initiator (IIG-NI)*, a library that contains classes and methods that may be invoked by the processes and arrange for delivering the notifications to the SmartGov service delivery environment. In more detail, the notification posting cycle proceeds as follows:

1. Processes that need to post notification events invoke IIG-NI, asking for the notification to be delivered by calling the appropriate Java class method of the IIG-NI. Parameters (e.g. the name of the notification event) are passed using the Java language calling convention. The invocation is synchronous, therefore the execution of the invoking process is suspended until the method execution is finished. Method execution takes place within the control thread of the calling program.
2. The IIG-NI assembles a message with all appropriate information needed for the delivery of the notification, including the name of the notification passed by the invoking process and the pertinent configuration and authentication data. This message sent to the local IIG, through a message passing

mechanism. Execution of the IIG-NI control thread (effectively, the execution of the invoking process) is suspended until the message is delivered and the reply collected, or a timeout expires with no acknowledgement having been returned, in which case an exception is raised.

3. The IIG forwards the notification to the SGA Notification Interceptor (SGA-NI) component. The SGA-NI receives the message and arranges for the insertion of an entry into the SGA-PAQ. This entry will be later examined by the SGA-PAQUED, as described in section 2.6.3.3, leading to the execution of an appropriate action.

2.6.3.5 Logging facilities

Logging facilities are provided at both the SmartGov service delivery environment and the organisational information system environment, in order to record and document events that have taken place and relate to the SmartGov platform. A logging request may be submitted by any program within the service delivery environment and the organisational information system environment, in which case the following procedure commences:

1. the calling application provides the message to be logged together with a designation of each criticality to the API provided by the SmartGov implementation. The calling application is suspended.
2. the procedure implementing the API locates the appropriate logging facility (SGA logger or IIG logger) and communicates to it the criticality designation and the message.
3. the logging facility arranges for storing the current timestamp, the criticality designation and the message to some persistent storage.
4. an acknowledgement is returned to the calling application that the log request has been honoured.
5. the calling application resumes its execution.

2.7 Deployment View

In the following figure a proposed network topology to host the various components of the SmartGov platform is depicted. This topology ensures a high level of security, while with the proper choice of hardware it will provide a satisfactory performance. In the following sections of the deployment view the specific needs of the major SmartGov platform components are presented.



Figure 61 SmartGov platform network topology

2.7.1 SmartGov Front-End

The SmartGov Front-end is developed using the Struts MVC Framework as starting point so the requirements will be fixed mainly by this architecture.

To run this front-end, a J2EE-compliant web server is required and also a Database server. A dedicated database server is not required, so the Front-end can use the same server that the other modules of SmartGov Platform. This server will be accessed through a JDBC Driver.

The number of users of the system and the size of the organization will be the key factor to evaluate the needs of our system: more concurrent users will demand higher requirements in the web server, and also a bigger organization will imply more services being developed, more knowledge stored, and thus

higher requirements. These requirements will have to be adjusted once a specific web server platform has been decided (Tomcat-Apache, Weblogic...) and also may be necessary to make some changes in configuration files to adapt the front-end. Once the front-end has been developed, it will be necessary making load tests to tune the requirements of the platform, according to the number of users and services being developed. Anyway, the processes have, in the general case, small footprints and are short-lived, so requirements should not be too different from the usual ones for web applications specified for each web server platform.

For execution of components developed in the Java language, the Java 1.4 runtime environment should be available on the hosting machine. For database access, the JDBC 3.0 API, bundled with the Java 1.4 runtime environment will be used. Also Struts and Castor classes should be available.

During the development, the used platform is going to be Apache 2 + Tomcat 4.1 as web server, adding the Struts and Castor packages. Also will be used MySql as database server. And finally, as aforementioned, the Java 1.4 runtime environment, and the JDBC 3.0 API, bundled with the Java 1.4 runtime environment, will be used.

2.7.2 Integrator

The integrator components will live within the instance of the servlet container. All necessary objects will be loaded from there and will have access to the database server through a JDBC driver and to the XML repository through the XML Repository API objects. Also access will be provided to the dissemination server of the platform in order for the integrator to deploy the generated web application.

The integrator will make use of the Jakarta Ant module, the Jakarta Struts Framework and will also require the Java 1.4 runtime environment and the JDBC 3.0 API.

2.7.3 SmartGov Agent - Information Interchange Gateway

In this section the requirements for the installation and execution of the SmartGov Agent and the Information Interchange Gateway are described.

2.7.3.1 Service delivery platform

The SmartGov Agent class library component runs as an integral part of the services delivered to the end users, thus no special provisions need to be made for it, either in terms of hardware or in terms of software environment.

The SGA-PAQUED is a separate process running periodically and accessing the PAQ. As a result of its operation, additional processes may be spawned. These services will have, in the general case, small footprints and be short-lived, thus no significant overheads will be incurred; however, local IT staff associating CPU-intensive or large-footprint processes with SGA-PAQ entries should consider the processing, memory and storage requirements of these programs when sizing the hardware.

For the management of the SGA-PAQ itself, database services need to be available. To this end, a database server should be deployed providing JDBC connectivity, in order to be accessible to the SGA-PAQUED. The database server may be dedicated to the purpose of hosting the SGA-PAQ, or may offer database services for other purposes.

In terms of host requirements, the SGA-PAQUED may be run on the machine that offers the database services for storing the SGA-PAQ. In this case, the hardware requirements of the machine will be primarily dictated by the database server, plus a 15% increase in processor and memory requirements. Disk capacity will not be affected, in general. If the SGA-PAQUED is hosted in a separate machine, a minimum configuration for it would be a 1Ghz Pentium processor (or equivalent) with 128 Mbytes of memory (unless the operating system installation instructions suggest a larger amount of memory) and a 4 GB disk.

The SGA-NI is a daemon process listening for incoming notification events and arranging for inserting the corresponding entries to the SGA-PAQ; its memory footprint is expected to be small and its processing requirements minimal, thus it may be hosted on the same machine that the SGA-PAQUED is run on.

The SGA-logger is a daemon process listening for incoming event logging requests and arranges for storing the contents of these log requests to some persistent storage. It is expected that its memory footprint will be small and its processing requirements minimal, thus it may be hosted on the same machine that the SGA-PAQUED is run on.

For execution of components developed in the Java language, the Java 1.4.1 runtime environment should be available on the hosting machine. For database access, the JDBC 3.0 API, bundled with the Java 1.4.1 runtime environment will be used.

2.7.3.2 Organisational Information System Environment

The organisational information system should host the IIG-MYP in order to be able to receive and serve requests originating from the service delivery environment. The IIG-MYP is a daemon process listening for incoming requests

and arranging for spawning a separate external process (SEP) or placing an appropriate entry in the IIG-PAQ. The memory footprint and the processing requirements of the IIG-MYP are minimal, for host sizing, however, two issues should be considered:

- The invocation frequency, memory footprint and processing requirements of the separate external processes. Usually, these processes will simply access a registry for retrieval or update purposes, however batch maintenance jobs may also be run.
- The storage and management of the IIG-PAQ database services need to be available. To this end, a database server should be deployed providing JDBC connectivity, in order to be accessible to the IIG-MYP. The database server may be dedicated to the purpose of hosting the IIG-PAQ, or may offer database services for other purposes.

The IIG-PAQ is also accessed by the IIG-PAQUED, a separate process running periodically and accessing the PAQ. As a result of its operation, additional processes may be spawned. These processes will have, in the general case, small footprints and be short-lived, thus no significant overheads will be incurred; however, local IT staff associating CPU-intensive or large-footprint processes with IIG-PAQ entries (e.g. batch jobs) should consider the processing, memory and storage requirements of these programs when sizing the hardware.

Overall, the IIG-MYP, IIG-SEPs, IIG-PAQEUD and IIG-PAQ may be hosted on a single machine, whose hardware requirements will be largely dictated by the database system that hosts the IIG-PAQ, plus a 30% increase in processing and memory requirements, if IIG-SEPs are of medium frequency, small-footprint and short-lived. If IIG-SEPs are very frequent, have large footprints or high demands for processing power, a higher increase should be opted for, or tasks should be split in two machines, one providing the database services and one hosting the IIG-MYP, the IIG-SEP and the IIG-PAQUED.

The IIG-logger is a daemon process listening for incoming event logging requests and arranges for storing the contents of these log requests to some persistent storage. It is expected that its memory footprint will be small and its processing requirements minimal, thus it may be hosted on the same machine that the SGA-PAQUED is run on.

It should be noted here that some organisational information system platforms may be able, in terms of software requirements and hardware capacity, to host the IIG-MYP, IIG-SEP and the IIG-PAQUED; however, it is advised that especially the IIG-MYP module is placed on a different machine than the organisational

information system, in order to facilitate the enforcement of strict security measures and not jeopardise the integrity of the organisational information system.

For execution of components developed in the Java language, the Java 1.4.1 runtime environment should be available on the hosting machine. For database access, the JDBC 3.0 API, bundled with the Java 1.4.1 runtime environment will be used.

2.8 Data view

2.8.1 Introduction

The Data view documents the SmartGov data model. This involves mapping classes and elements defined in the Logical view to XML Schemas or relational tables in a relational database.

SmartGov use a database, a well known storage system, to store some its data (Users, Roles and WorkGroup). The rest of data is stored in XML, that is validate against XML Schemas, as it is stated in the XML Doc Repository chapter (2.4.2.2). XML Schemas express shared vocabularies and allow machines to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents. XML Schemas provides mechanisms for declaring, defining and refining data types.

In the following chapters the commented structure of the relevant SmartGov XML Schemas is explained

2.8.2 Modelling of Transaction Services

<p>diagram</p>																										
<p>children</p>	<p>TSId name description includedFormSets linkedKUNode linkedTaxonomyNode validationRule authenticationRequirements preaction postaction allowSave allowEdit allowDelete deadline lifeCycle statistics</p>																									
<p>identity constraints</p>	<table border="1"> <thead> <tr> <th></th> <th>Name</th> <th>Refer</th> <th>Selector</th> <th>Field(s)</th> </tr> </thead> <tbody> <tr> <td>key</td> <td>TSKey</td> <td></td> <td>../TS</td> <td>TSId</td> </tr> <tr> <td>keyref</td> <td>formKeyRef</td> <td>formKey</td> <td>../TS/includedFormS ets</td> <td>formId</td> </tr> <tr> <td>keyref</td> <td>TSLinkedKUNodeRef</td> <td>KUKey</td> <td>../TS</td> <td>linkedKUNode</td> </tr> <tr> <td>keyref</td> <td>TSLinkedTaxonomyNod eRef</td> <td>taxonomyNode Key</td> <td>../TS</td> <td>linkedTaxonomyNode</td> </tr> </tbody> </table>		Name	Refer	Selector	Field(s)	key	TSKey		../TS	TSId	keyref	formKeyRef	formKey	../TS/includedFormS ets	formId	keyref	TSLinkedKUNodeRef	KUKey	../TS	linkedKUNode	keyref	TSLinkedTaxonomyNod eRef	taxonomyNode Key	../TS	linkedTaxonomyNode
	Name	Refer	Selector	Field(s)																						
key	TSKey		../TS	TSId																						
keyref	formKeyRef	formKey	../TS/includedFormS ets	formId																						
keyref	TSLinkedKUNodeRef	KUKey	../TS	linkedKUNode																						
keyref	TSLinkedTaxonomyNod eRef	taxonomyNode Key	../TS	linkedTaxonomyNode																						
<p>source</p>	<pre><xs:element name="TS"> <xs:complexType> <xs:sequence></pre>																									

<pre> <xs:element name="TSId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="description" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="includedFormSets" type="formSet" maxOccurs="unbounded"/> <xs:element name="linkedKUNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="validationRule" type="validationMethod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="authenticationRequirements" type="xs:string"/> <xs:element name="preaction" type="method"/> <xs:element name="postaction" type="method"/> <xs:element name="allowSave" type="xs:boolean"/> <xs:element name="allowEdit" type="xs:boolean"/> <xs:element name="allowDelete" type="xs:boolean"/> <xs:element name="deadline" type="xs:date"/> <xs:element name="lifeCycle" type="lifeCycleType"/> <xs:element name="statistics" type="TSStatistics" minOccurs="0"/> </xs:sequence> </xs:complexType> <xs:key name="TSKey"> <xs:selector xpath="//TS"/> <xs:field xpath="TSId"/> </xs:key> <xs:keyref name="formKeyRef" refer="formKey"> <xs:selector xpath="//TS/includedFormSets"/> <xs:field xpath="formId"/> </xs:keyref> <xs:keyref name="TSLinkedKUNodeRef" refer="KUKey"> <xs:selector xpath="//TS"/> <xs:field xpath="linkedKUNode"/> </xs:keyref> <xs:keyref name="TSLinkedTaxonomyNodeRef" refer="taxonomyNodeKey"> <xs:selector xpath="//TS"/> <xs:field xpath="linkedTaxonomyNode"/> </xs:keyref> </xs:element> </pre>

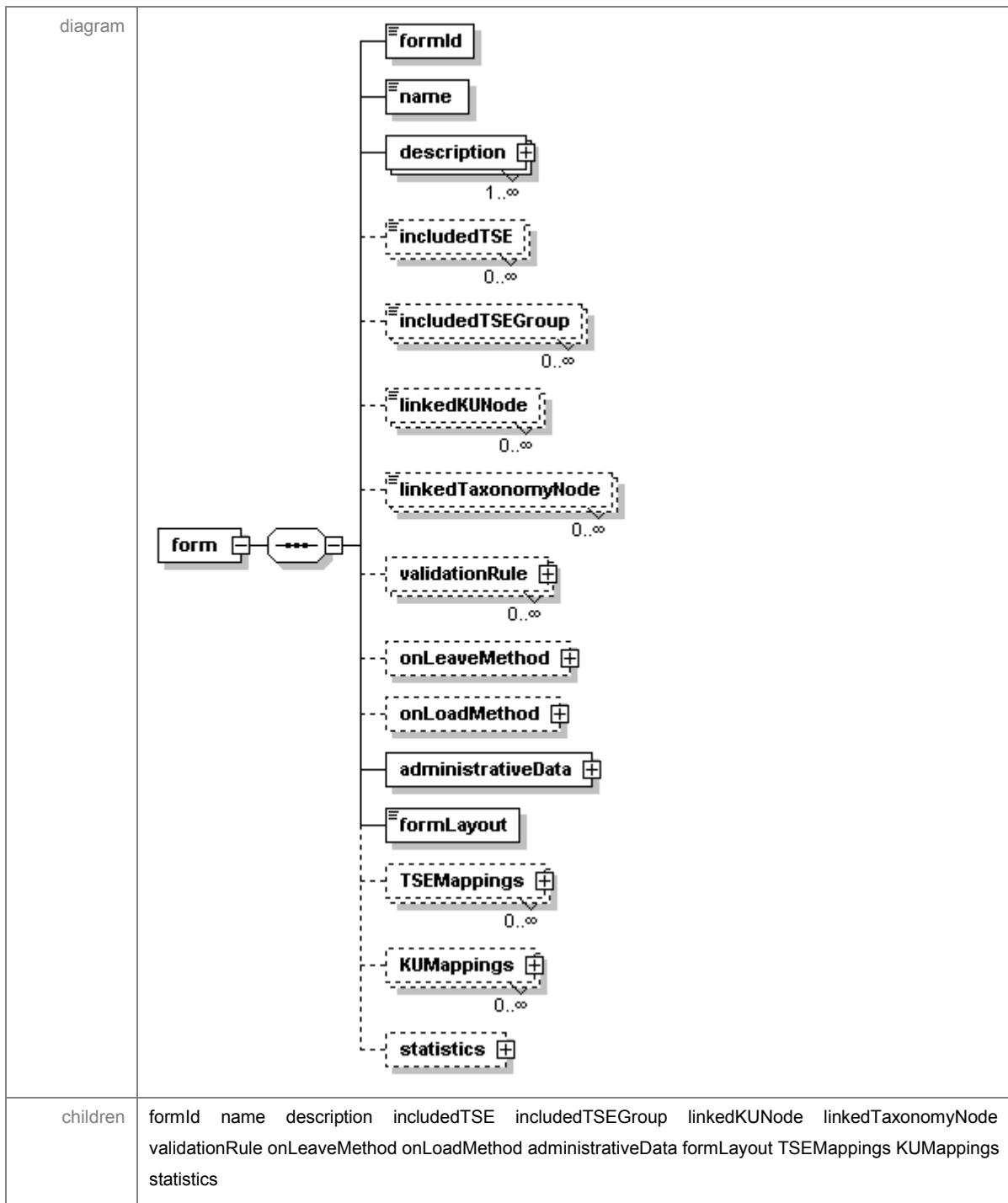
The elements of the TS entity are described in the following paragraphs.

- *TSId*: a string-typed element in which the id of the Transaction Service (TS) is stored.
- *name*: The name of the transaction service. The element type is *string*.
- *description*: The description of the transaction service. Descriptions in multiple languages may be accommodated. The element type is *multilingualText*.
- *includedFormSets*. This element contains one form set for each platform that will be used for the dissemination of the service. Each form set is tagged with the name of the target platform; for each platform one or more forms may be used. The element type is *formSet*.
- *linkedKUNode*: This element provides references to the KU nodes with which the TS is associated. The type of the element is *string* and it may occur multiple times.
- *linkedTaxonomyNode*: This element provides references to the taxonomy nodes with which the TS is associated. The type of the element is *string* and it may occur multiple times.
- *validationRule*: The rules used to ensure the validity of a document submitted via the specific TS. A submitted document is considered valid if

all validation rules associated with the TS succeed. The type of the element is *validationMethod* and it may occur multiple times.

- *authenticationRequirements*: Description of the authentication procedure necessary for the use of the service. It could be "none", "username and password", etc
- *preaction*: Code executed during the initialization of the service. If it fails, the service does not commence. This code fragment may be used to check the necessary preconditions for service execution. The type of the element is *method*.
- *postaction*: Code that is executed after the conclusion of the service. This code fragment may be used to enable the usage of other transaction services, dependent on the current one. The type of the element is *method*.
- *allowSave*: If this property is set, the user of the service will be provided with a "Save" functionality in order to be able to save a not-yet-completed form to submit it later. The element type is *boolean*.
- *allowEdit*: If this property is set, the users of the service will be provided with an "Edit" functionality, which will enable them to edit documents submitted through the TS, until these documents are characterized as final by the PA. The element type is *boolean*.
- *allowDelete*: If this property is set, the users of the service will be provided with a "Delete" functionality, which will enable them to delete documents submitted via the specific TS, until these documents are characterized as "final" by the PA. The element type is *boolean*.
- *deadline*: Deadline for the submission documents for the specific service. The element type is *date*.
- *lifecycle*: Information pertaining to the life cycle of the transaction service. The element type is *lifeCycleType*.
- *statistics*: Definition of the statistics to be collected for the specific TS. The element type is *TSStatistics*.

2.8.3 Modeling of Forms



identity constraints	Name	Refer	Selector	Field(s)
	key	formKey	../form	formId
	keyref	instantiatedTSEKeyRefIncludedTSE	../form	includedTSE
	keyref	instantiatedTSEGroupKeyRefIncludedTSEGroup	../form	includedTSEGroup
	keyref	formLinkedKUNodeRef	../form	linkedKUNode
	keyref	formLinkedTaxonomyNodeRef	../form	linkedTaxonomyNode
	keyref	forminstantiatedTSEKeyRefTSEMappings	../form/TSEMappings	instantiatedTSEId
	keyref	formKUNodeRefKUMappings	../form/KUMappings	KUIId
source	<pre> <xs:element name="form"> <xs:complexType> <xs:sequence> <xs:element name="formId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="description" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="includedTSE" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="includedTSEGroup" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedKUNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="validationRule" type="validationMethod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="onLeaveMethod" type="method" minOccurs="0"/> <xs:element name="onLoadMethod" type="method" minOccurs="0"/> <xs:element name="administrativeData" type="administrativeInfo"/> <xs:element name="formLayout" type="xs:anyURI"/> <xs:element name="TSEMappings" type="TSEToFormElement" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="KUMappings" type="KUToHelpItem" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="statistics" type="formStatistics" minOccurs="0"/> </xs:sequence> </xs:complexType> <xs:key name="formKey"> <xs:selector xpath="../form"/> <xs:field xpath="formId"/> </xs:key> <xs:keyref name="instantiatedTSEKeyRefIncludedTSE" refer="instantiatedTSEKey"> <xs:selector xpath="../form"/> <xs:field xpath="includedTSE"/> </xs:keyref> <xs:keyref name="instantiatedTSEGroupKeyRefIncludedTSEGroup" refer="instantiatedTSEGroupKey"> <xs:selector xpath="../form"/> <xs:field xpath="includedTSEGroup"/> </xs:keyref> <xs:keyref name="formLinkedKUNodeRef" refer="KUNodeKey"> <xs:selector xpath="../form"/> <xs:field xpath="linkedKUNode"/> </xs:keyref> <xs:keyref name="formLinkedTaxonomyNodeRef" refer="taxonomyNodeKey"> <xs:selector xpath="../form"/> <xs:field xpath="linkedTaxonomyNode"/> </xs:keyref> <xs:keyref name="forminstantiatedTSEKeyRefTSEMappings" refer="instantiatedTSEKey"> <xs:selector xpath="../form/TSEMappings"/> <xs:field xpath="instantiatedTSEId"/> </xs:keyref> <xs:keyref name="formKUNodeRefKUMappings" refer="KUNodeKey"> <xs:selector xpath="../form/KUMappings"/> </pre>			

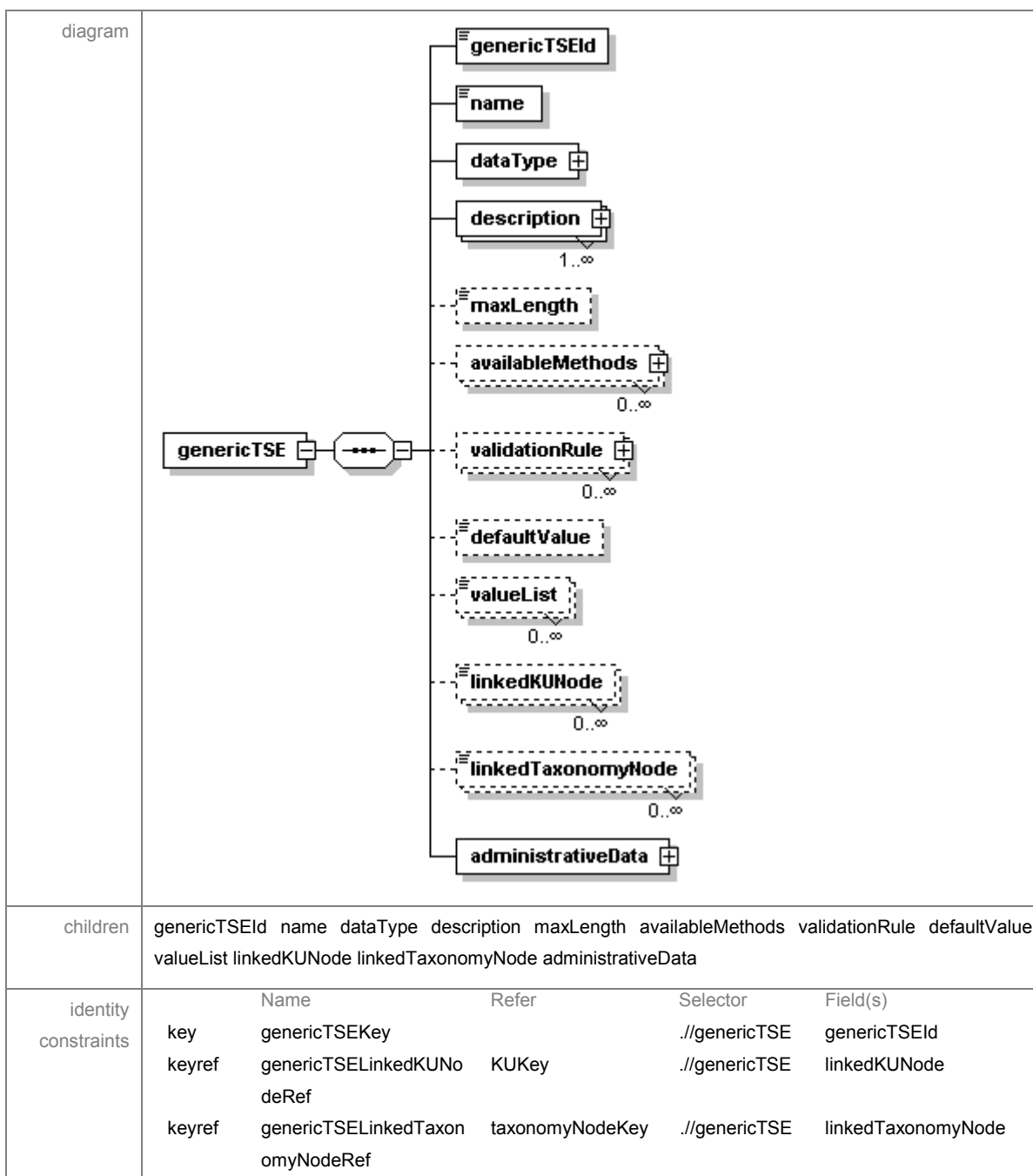
	<pre><xs:field xpath="KUId"/> </xs:keyref> </xs:element></pre>
	<p>This element models the <i>form</i> object of the SmartGov platform. Each form is a “page” of a document that may be submitted through a transaction service.</p>

The elements of the form entity are described in the following paragraphs.

- *formId*: A string-typed element containing the id of the form.
- *name*: The name of the form. The element type is *string*.
- *description*: The description of the form. Descriptions in multiple languages may be accommodated. The element type is *multilingualText*.
- *includedTSE*. A form contains one such element for each TSE directly included in it. The element type is *string* and it contains the id of the TSE.
- *includedTSEGroup*. A form contains one such element for each TSE group included in it. The element type is *string* and it contains the id of the TSE group.
- *linkedKUNode*: This element provides references to the KU nodes with which the form is associated. The type of the element is *string* and it may occur multiple times.
- *linkedTaxonomyNode*: This element provides references to the taxonomy nodes with which the form is associated. The type of the element is *string* and it may occur multiple times.
- *validationRule*: The rules used to ensure the validity of a the form contents. A form is considered valid if all validation rules associated with the form succeed. The type of the element is *validationMethod* and it may occur multiple times.
- *onLeave*: Code that is executed when the user leaves the form. The element type is *method*.
- *onLoad*: Code that is executed when the form is loaded. The element type is *method*.
- *administrativeData*: Information useful for administrative purposes for the form within the SmartGov development platform. The type of the element is *administrativeInfo*.
- *formLayout*: A pointer to the file containing the layout for the form. The location is specified by means of a URI, pointing to the file.
- *TSEMappings*: The mapping of TSEs to the visual elements of the form. This element is populated through the procedure described in section 2.3.2.6 - “Link Establishment Between Form Layout and Form Semantics”. The element type is *TSEToFormElement* and may occur multiple times, one for each mapping.

- *KUMappings*: The mapping of KUs to the visual elements of the form. This element is populated through the procedure described in section 2.3.2.6 - "Link Establishment Between Form Layout and Form Semantics". The element type is *KUToHelpItem* and may occur multiple times, one for each mapping.
- *statistics*: Definition of the statistics to be collected for the specific form. The element type is *formStatistics*.

2.8.4 Modelling of Generic TSEs



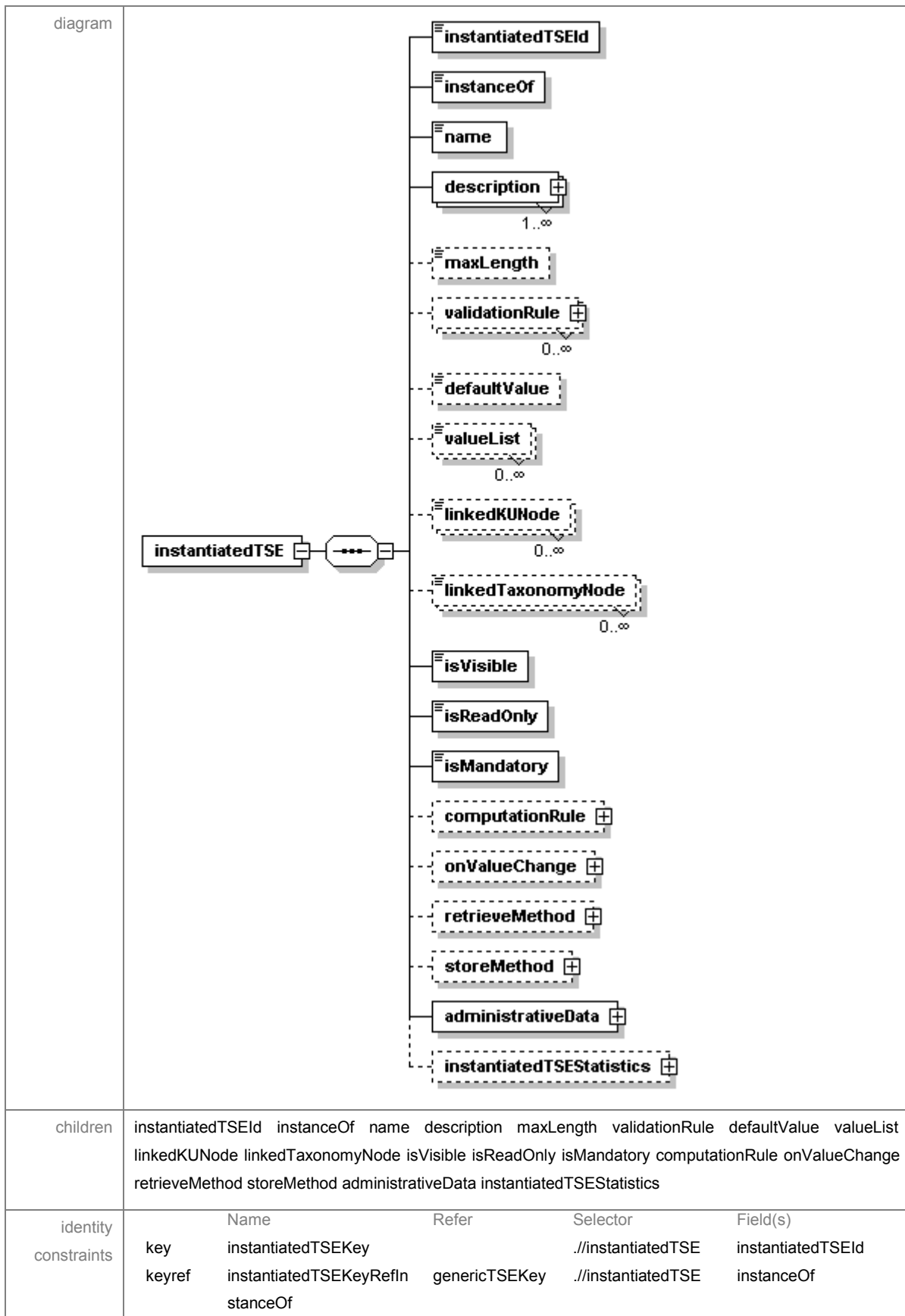
<p>source</p>	<pre> <xs:element name="genericTSE"> <xs:complexType> <xs:sequence> <xs:element name="genericTSEId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="dataType" type="TSEDataType"/> <xs:element name="description" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="maxLength" type="xs:positiveInteger"/> <xs:element name="availableMethods" type="method" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="validationRule" type="validationMethod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="defaultValue" type="xs:string"/> <xs:element name="valueList" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedKUNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="workGroup" type="xs:string"/> </xs:sequence> </xs:complexType> <xs:key name="genericTSEKey"> <xs:selector xpath="//genericTSE"/> <xs:field xpath="genericTSEId"/> </xs:key> <xs:keyref name="genericTSELinkedKUNodeRef" refer="KUKey"> <xs:selector xpath="//genericTSE"/> <xs:field xpath="linkedKUNode"/> </xs:keyref> <xs:keyref name="genericTSELinkedTaxonomyNodeRef" refer="taxonomyNodeKey"> <xs:selector xpath="//genericTSE"/> <xs:field xpath="linkedTaxonomyNode"/> </xs:keyref> </xs:element> </pre>
	<p>This element models the generic TSE object of the SmartGov platform.</p>

The elements of the generic TSE entity are described in the following paragraphs.

- *genericTSEId*: A string-typed element containing the id of the generic TSE.
- *name*: The name of the generic TSE. The element type is *string*.
- *dataType*: The data type of the generic TSE. The type of the element is *TSEDataType*.
- *description*: The description of the generic TSE. Descriptions in multiple languages may be accommodated. The element type is *multilingualText*.
- *maxLength*: The maximum length of the values that this generic TSE accepts. The type of the element is *positiveInteger*.
- *availableMethods*: Pieces of code that may apply to instances of this TSE type. They may implement conditional checks, formatting, transformations, value assignments etc. The type of the element is *method* and may occur multiple times.
- *validationRule*: Conditions that must be fulfilled for the TSE value to be considered acceptable. A generic TSE may contain any number of validation rules (one instance of this element for each rule). The type of the element is *validationMethod*.
- *defaultValue*: The default value for instances of this TSE. The type of the element is *string*.

- *valueList*: The possible values that may be assigned to the generic TSE. Used only if the values are restricted to a specific set. The type of the element is *string* and may occur multiple times, one for each possible value.
- *linkedKUNode*: This element provides references to the KU nodes with which the generic TSE is associated. The type of the element is *string* and it may occur multiple times.
- *linkedTaxonomyNode*: This element provides references to the taxonomy nodes with which the generic TSE is associated. The type of the element is *string* and it may occur multiple times.
- *administrativeData*: Information useful for administrative purposes for the form within the SmartGov development platform. The type of the element is *administrativeInfo*.

2.8.5 Modelling of Instantiated TSEs



	keyref	instantiatedTSELinkedKUNodeRef	KUKey	../instantiatedTSE	linkedKUNode
	keyref	instantiatedTSELinkedTaxonomyNodeRef	taxonomyNodeKey	../instantiatedTSE	linkedTaxonomyNode
source	<pre> <xs:element name="instantiatedTSE"> <xs:complexType> <xs:sequence> <xs:element name="instantiatedTSEId" type="xs:string"/> <xs:element name="instanceOf" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="description" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="maxLength" type="xs:positiveInteger"/> <xs:element name="validationRule" type="validationMethod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="defaultValue" type="xs:string"/> <xs:element name="valueList" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedKUNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="isVisible" type="xs:boolean"/> <xs:element name="isReadOnly" type="xs:boolean"/> <xs:element name="isMandatory" type="xs:boolean"/> <xs:element name="computationRule" type="method" minOccurs="0"/> <xs:element name="onValueChange" type="method" minOccurs="0"/> <xs:element name="retrieveMethod" type="method" minOccurs="0"/> <xs:element name="storeMethod" type="method" minOccurs="0"/> <xs:element name="administrativeData" type="administrativeInfo"/> <xs:element name="instantiatedTSEStatistics" type="TSEStatistics" minOccurs="0"/> </xs:sequence> </xs:complexType> <xs:key name="instantiatedTSEKey"> <xs:selector xpath="../instantiatedTSE"/> <xs:field xpath="instantiatedTSEId"/> </xs:key> <xs:keyref name="instantiatedTSEKeyRefInstanceOf" refer="genericTSEKey"> <xs:selector xpath="../instantiatedTSE"/> <xs:field xpath="instanceOf"/> </xs:keyref> <xs:keyref name="instantiatedTSELinkedKUNodeRef" refer="KUKey"> <xs:selector xpath="../instantiatedTSE"/> <xs:field xpath="linkedKUNode"/> </xs:keyref> <xs:keyref name="instantiatedTSELinkedTaxonomyNodeRef" refer="taxonomyNodeKey"> <xs:selector xpath="../instantiatedTSE"/> <xs:field xpath="linkedTaxonomyNode"/> </xs:keyref> </xs:element> </pre>				
	This element represents the instantiated TSE, i.e. the TSE defined within the context of a specific service.				

The elements of the instantiated TSE entity are described in the following paragraphs.

- *instantiatedTSEId*: A string-typed element containing the id of the instantiated TSE.
- *instanceOf*: A reference to the id of the generic TSE that the instantiated TSE is modeled after. The element type is *string*.
- *name*: The name of the instantiated TSE. The element type is *string*.
- *description*: The description of the instantiated TSE. Descriptions in multiple languages may be accommodated. The element type is *multilingualText*.

- *maxLength*: The maximum length of the values that this instantiated TSE accepts. The type of the element is *positiveInteger*. If specified, it overrides the value inherited from the generic TSE.
- *validationRule*: Conditions that must be fulfilled for the TSE value to be considered acceptable. An instantiated TSE may contain any number of validation rules (one instance of this element for each rule), which complement the ones inherited from the generic TSE. The type of the element is *validationMethod*.
- *defaultValue*: The default value for this TSE. The type of the element is *string*. If specified, it overrides the value inherited from the generic TSE.
- *valueList*: The possible values that may be assigned to this TSE. Used only if the values are restricted to a specific set. The type of the element is *string* and may occur multiple times, one for each possible value. If specified, it overrides the value inherited from the generic TSE.
- *linkedKUNode*: This element provides references to the KU nodes with which the instantiated TSE is associated. The type of the element is *string* and it may occur multiple times. These references, if specified, supplement the ones inherited from the generic TSE.
- *linkedTaxonomyNode*: This element provides references to the taxonomy nodes with which the instantiated TSE is associated. The type of the element is *string* and it may occur multiple times. These references, if specified, supplement the ones inherited from the generic TSE.
- *isVisible*: Indicates whether the occurrences of the instantiated TSE should be visible or not. The element type is *boolean*.
- *isReadOnly*: Indicates whether the users may change the values of occurrences of thus instantiated TSE or not. The element type is *boolean*.
- *isMandatory*: Indicates whether the instantiated TSE is mandatory to be completed or not. The element type is *boolean*.
- *computationRule*: Code that computes the value of the instantiated TSE, when its value depends on the values of other TSEs. TSEs with computation rules are considered as read-only. The type of this element is *method*.
- *onValueChange*: Code that is executed when the value of the instantiated TSE changes. The type of this element is *method*.
- *retrieveMethod*: Code that is executed in order to retrieve the value of the instantiated TSE. This code fragment is executed once, when the user enters the transaction service. The type of this element is *method*.

- *storeMethod*: Code that is executed in order to store the value of the instantiated TSE. This code fragment is executed once, when the user submits the document modeled by the transaction service. The type of this element is *method*.
- *administrativeData*: Information useful for administrative purposes for the instantiatedTSE. This information is used within the SmartGov platform development environment. The type of this element is *administrativeInfo*.
- *instantiatedTSEStatistics*. Definition of statistics that should be collected for the specific instantiated TSE. The type of this element is *TSEStatistics*.

2.8.5.1 Inheritance rules for TSE instantiation

The SmartGov platform allows for the definition of two types of TSEs, namely generic and instantiated ones. Generic TSEs allow for grouping of desired element properties into a single entity, while instantiated TSEs are concrete occurrences of such elements within transaction services. An instantiated TSE contains a reference to the generic TSE it is modeled after, and this reference implies that the instantiated TSE inherits the properties defined in the generic TSE. The instantiated TSE, however, is by nature more specific than its generic counterpart, so the SmartGov platform should provide the capability to redefine or supplement various aspects of the inherited properties. To this end, the instantiated TSE contains information slots, which may be filled with values that either override or complement the information inherited from the homonymous data slots of its template. The rules for determining whether the instantiated TSE information slots override or supplement the inherited values have been listed in the element descriptions above and are summarised in the following table:

Information slot	Override or supplement
Default value	Override
Validation rules	Supplement
References to knowledge base	Supplement
Maximum Length	Override
Value List	Override
References to taxonomy	Supplement

2.8.6 Modelling of Generic TSE Groups

<p>diagram</p>																																			
<p>children</p>	<p>genericTSEGroupId name description includedTSE linkedKUNode linkedTaxonomyNode validationRule repeats administrativeData</p>																																		
<p>identity constraints</p>	<table border="1"> <thead> <tr> <th>key</th> <th>Name</th> <th>Refer</th> <th>Selector</th> <th>Field(s)</th> </tr> </thead> <tbody> <tr> <td>key</td> <td>genericTSEGroupKey</td> <td></td> <td>//genericTSEGroup</td> <td>genericTSEGroupId</td> </tr> <tr> <td>keyref</td> <td>includedGenericTSEKey</td> <td>genericTSEKey</td> <td>//genericTSEGroup/includedTSE</td> <td>genericTSEId</td> </tr> <tr> <td>keyref</td> <td>repeatsGenericTSEKey</td> <td>genericTSEKey</td> <td>//genericTSEGroup/repeats</td> <td>uniqueColumn</td> </tr> <tr> <td>keyref</td> <td>genericTSEGroupLinkedKUNodeRef</td> <td>KUKey</td> <td>//genericTSEGroup</td> <td>linkedKUNode</td> </tr> <tr> <td>keyref</td> <td>genericTSEGroupLinkedTaxonomyNodeRef</td> <td>taxonomyNodeKey</td> <td>//genericTSEGroup</td> <td>linkedTaxonomyNode</td> </tr> </tbody> </table>	key	Name	Refer	Selector	Field(s)	key	genericTSEGroupKey		//genericTSEGroup	genericTSEGroupId	keyref	includedGenericTSEKey	genericTSEKey	//genericTSEGroup/includedTSE	genericTSEId	keyref	repeatsGenericTSEKey	genericTSEKey	//genericTSEGroup/repeats	uniqueColumn	keyref	genericTSEGroupLinkedKUNodeRef	KUKey	//genericTSEGroup	linkedKUNode	keyref	genericTSEGroupLinkedTaxonomyNodeRef	taxonomyNodeKey	//genericTSEGroup	linkedTaxonomyNode				
key	Name	Refer	Selector	Field(s)																															
key	genericTSEGroupKey		//genericTSEGroup	genericTSEGroupId																															
keyref	includedGenericTSEKey	genericTSEKey	//genericTSEGroup/includedTSE	genericTSEId																															
keyref	repeatsGenericTSEKey	genericTSEKey	//genericTSEGroup/repeats	uniqueColumn																															
keyref	genericTSEGroupLinkedKUNodeRef	KUKey	//genericTSEGroup	linkedKUNode																															
keyref	genericTSEGroupLinkedTaxonomyNodeRef	taxonomyNodeKey	//genericTSEGroup	linkedTaxonomyNode																															
<p>source</p>	<pre> <xs:element name="genericTSEGroup"> <xs:complexType> <xs:sequence> <xs:element name="genericTSEGroupId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="description" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="includedTSE" type="containedTSE" maxOccurs="unbounded"/> <xs:element name="linkedKUNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="validationRule" type="validationMethod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="repeats" type="repetitionInformation" minOccurs="0"/> <xs:element name="administrativeData" type="administrativeInfo"/> </xs:sequence> </xs:complexType> <xs:key name="genericTSEGroupKey"> <xs:selector xpath="//genericTSEGroup"/> <xs:field xpath="genericTSEGroupId"/> </xs:key> </pre>																																		

	<pre> </xs:key> <xs:keyref name="includedGenericTSEKeyRef" refer="genericTSEKey"> <xs:selector xpath="//genericTSEGroup/includedTSE"/> <xs:field xpath="genericTSEId"/> </xs:keyref> <xs:keyref name="repeatsGenericTSEKeyRef" refer="genericTSEKey"> <xs:selector xpath="//genericTSEGroup/repeats"/> <xs:field xpath="uniqueColumn"/> </xs:keyref> <xs:keyref name="genericTSEGroupLinkedKUNodeRef" refer="KUKey"> <xs:selector xpath="//genericTSEGroup"/> <xs:field xpath="linkedKUNode"/> </xs:keyref> <xs:keyref name="genericTSEGroupLinkedTaxonomyNodeRef" refer="taxonomyNodeKey"> <xs:selector xpath="//genericTSEGroup"/> <xs:field xpath="linkedTaxonomyNode"/> </xs:keyref> </xs:element> </pre>
	This element represents the generic TSE Group object of the SmartGov platform.

The elements of the generic TSE group entity are described in the following paragraphs.

- *genericTSEGroupId*: A string-typed element containing the id of the generic TSE group.
- *name*: The name of the generic TSE group. The element type is *string*.
- *description*: The description of the generic TSE group. Descriptions in multiple languages may be accommodated. The element type is *multilingualText*.
- *includedTSE*: References to the generic TSEs that participate in the TSE group. The element may occur multiple times, one for each generic TSE that is included in the TSE group. The type of the element is *containedTSE*.
- *linkedKUNode*: This element provides references to the KU nodes with which the generic TSE group is associated. The type of the element is *string* and it may occur multiple times.
- *linkedTaxonomyNode*: This element provides references to the taxonomy nodes with which the generic TSE group is associated. The type of the element is *string* and it may occur multiple times.
- *validationRule*: Conditions that must be fulfilled for the values of the TSEs participating in the TSE group. A generic TSE group may contain any number of validation rules (one instance of this element for each rule). The type of the element is *validationMethod*.
- *repeats*: This element indicates whether a unique occurrence of the TSEs included in the group is required or whether multiple occurrences should be used (as the rows of a table for example). The type of the element is *repetitionInformation* and should not be present, if no repetition is desired.

- *administrativeData*: Information useful for administrative purposes for the generic TSE group. This information is used within the SmartGov platform development environment. The type of this element is *administrativeInfo*.

2.8.6.1 Repetition information for groups

In many administrative forms there is a need to provide tabular areas, in which citizens will fill in *rows* of data, one row for each pertinent entity. Examples of such tabular areas are:

1. *Declaration of family members*. For each family member the name, surname and date of birth is declared. This tabular area may have the following form:

	Surname	Name	Date of birth
1.			
2.			
3.			
4.			
5.			

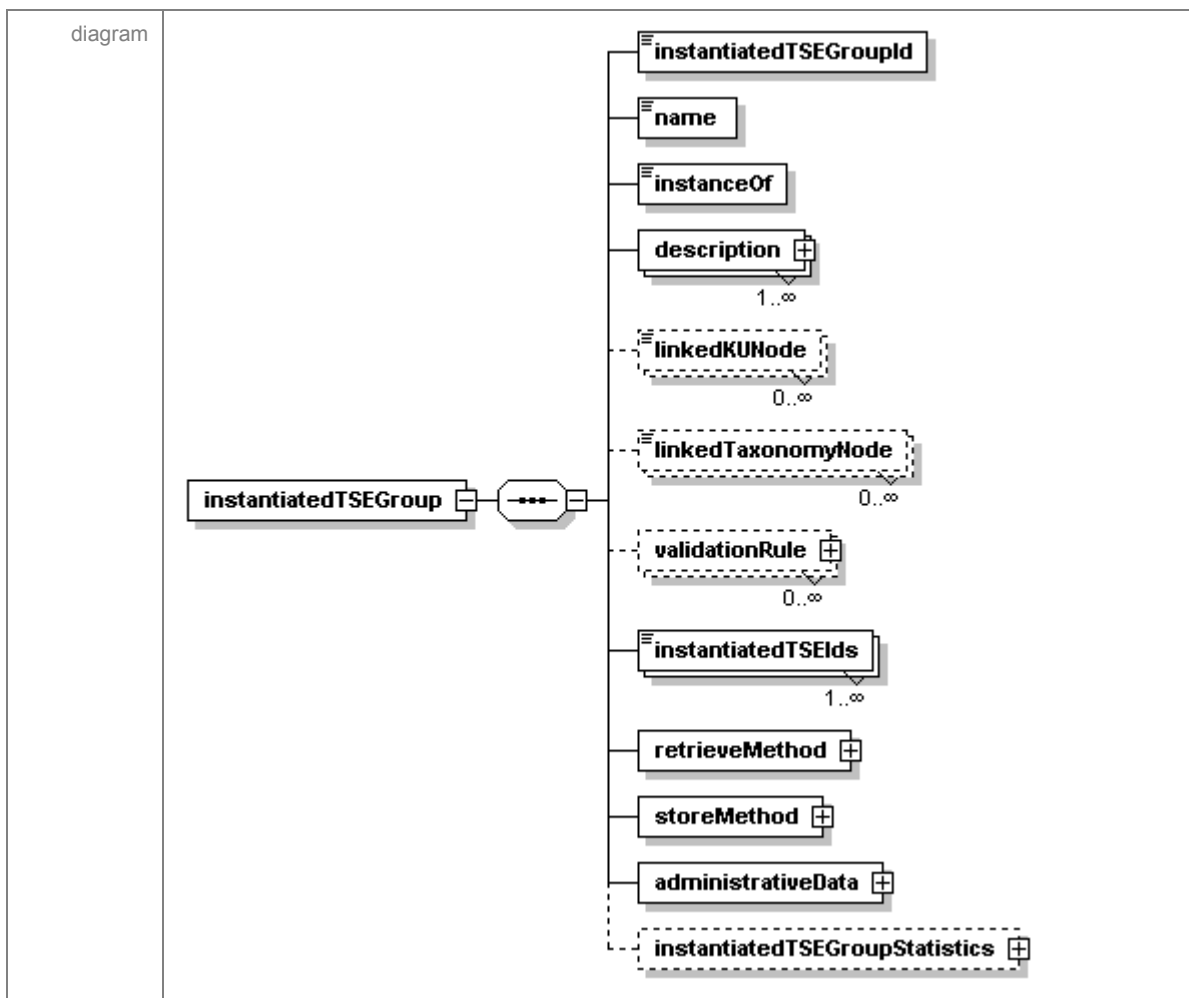
2. *Recapitulative Statement of intra-Community Supplies*. For each buyer with whom intra-community sales have taken place for the respective period, one row is filled in this document, stating the country and the VAT number of the buyer, along with the taxable amount(s) for intra-community goods supplies and triangular intra-community supplies. (For more details see D41.) This tabular area has the following form:

	Buyer country, etc (a)	Country Prefix (b)	VAT Number (c)	Taxable Amount	
				Intra-Community goods supplies (d)	Triangular Intra-Community Supplies (e)
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					

In the SmartGov platform the designer models such tabular areas by defining the fields appearing in each row and specifying suitable repetition information, rather than defining separate TSEs, for each cell of the tabular area. The repetition information associated with such constructs is represented by the following items:

- *minOccurrences*: The minimum number of rows that may appear on the form
- *maxOccurrences*: The maximum number of rows that can be placed on a form
- *initialRows*: the number of rows initially displayed
- *uniqueColumn*: one or more columns whose values must be unique within the tabular area. If multiple columns are present, the *value combination* of these columns should be unique. For instance in the *recapitulative statement of intra-community supplies* form, the combination of the *country prefix* and *VAT number* field should be unique.
- *groupControls*: Each such tabular area may need to have controls that enable the user to add rows (if more than the currently displayed are needed) or delete rows (if a large number of empty rows is considered impractical).
- *rowStep*: The number of rows that will be added/deleted in such a case.

2.8.7 Modelling of Instantiated TSE Groups



children	instantiatedTSEGroupId name instanceOf description linkedKUNode linkedTaxonomyNode validationRule instantiatedTSEIds retrieveMethod storeMethod administrativeData instantiatedTSEGroupStatistics				
identity constraints		Name	Refer	Selector	Field(s)
	key	instantiatedTSEGroupKey		../instantiatedTSEGroup	instantiatedTSEGroupId
	keyref	instantiatedTSEKeyRefId	instantiatedTSEKey	../instantiatedTSEGroup/instantiatedTSEIds	instantiatedTSEId
	keyref	instantiatedTSEGroupKeyRefInstanceOf	genericTSEGroupKey	../instantiatedTSEGroup	instanceOf
	keyref	instantiatedTSEGroupLinkedKUNodeRef	KUKey	../instantiatedTSEGroup	linkedKUNode
	keyref	instantiatedTSEGroupLinkedTaxonomyNodeRef	taxonomyNodeKey	../instantiatedTSEGroup	linkedTaxonomyNode
source	<pre> <xs:element name="instantiatedTSEGroup"> <xs:complexType> <xs:sequence> <xs:element name="instantiatedTSEGroupId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="instanceOf" type="xs:string"/> <xs:element name="description" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="linkedKUNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="validationRule" type="validationMethod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="instantiatedTSEIds" type="xs:string" maxOccurs="unbounded"/> <xs:element name="retrieveMethod" type="method"/> <xs:element name="storeMethod" type="method"/> <xs:element name="administrativeData" type="administrativeInfo"/> <xs:element name="instantiatedTSEGroupStatistics" type="TSEGroupStatistics" minOccurs="0"/> </xs:sequence> </xs:complexType> <xs:key name="instantiatedTSEGroupKey"> <xs:selector xpath="../instantiatedTSEGroup"/> <xs:field xpath="instantiatedTSEGroupId"/> </xs:key> <xs:keyref name="instantiatedTSEKeyRefId" refer="instantiatedTSEKey"> <xs:selector xpath="../instantiatedTSEGroup/instantiatedTSEIds"/> <xs:field xpath="instantiatedTSEId"/> </xs:keyref> <xs:keyref name="instantiatedTSEGroupKeyRefInstanceOf" refer="genericTSEGroupKey"> <xs:selector xpath="../instantiatedTSEGroup"/> <xs:field xpath="instanceOf"/> </xs:keyref> <xs:keyref name="instantiatedTSEGroupLinkedKUNodeRef" refer="KUKey"> <xs:selector xpath="../instantiatedTSEGroup"/> <xs:field xpath="linkedKUNode"/> </xs:keyref> <xs:keyref name="instantiatedTSEGroupLinkedTaxonomyNodeRef" refer="taxonomyNodeKey"> <xs:selector xpath="../instantiatedTSEGroup"/> <xs:field xpath="linkedTaxonomyNode"/> </xs:keyref> </xs:element> </pre>				
	This element represents the instantiated TSE Group, i.e. the TSE Group defined within the context of a specific service.				

The elements of the instantiated TSE group entity are described in the following paragraphs.

- *instantiatedTSEGroupId*: A string-typed element containing the id of the instantiated TSE group.
- *name*: The name of the instantiated TSE group. The element type is *string*.
- *instanceOf*: A reference to the generic TSE group that the specific instantiated TSE group is modelled after (actually, the id of the generic TSE group). The type of this element is *string*.
- *description*: The description of the instantiated TSE group. Descriptions in multiple languages may be accommodated. The element type is *multilingualText*.
- *linkedKUNode*: This element provides references to the KU nodes with which the instantiated TSE group is associated. The type of the element is *string* and it may occur multiple times. These elements, if specified, supplement the ones inherited from the generic TSE.
- *linkedTaxonomyNode*: This element provides references to the taxonomy nodes with which the instantiated TSE group is associated. The type of the element is *string* and it may occur multiple times. These elements, if specified, supplement the ones inherited from the generic TSE.
- *validationRule*: Conditions that must be fulfilled for the values of the TSEs participating in the TSE group. An instantiated TSE group may contain any number of validation rules (one instance of this element for each rule). The type of the element is *validationMethod*. These elements, if specified, supplement the ones inherited from the generic TSE.
- *instantiatedTSEIds*: A list of references to the instantiated TSE ids participating in the instantiated TSE group. Upon the instantiation of a generic TSE group (i.e. the placement on a form of the generic TSE group), an instance is created for each of the generic TSEs referenced by the generic TSE group, and this list is populated with the identities of the newly created instantiated TSEs. This list is not editable by the user, i.e. the user may not add or delete individual TSEs from this list. Individual properties, however, of the instantiated TSEs may be set through the pertinent instantiated TSE slots. If the instantiated TSE group is deleted, all referenced instantiated TSEs should be removed from the SmartGov platform. The type of this element is *string* and may occur multiple times.
- *retrieveMethod*: Code that is executed in order to retrieve the values of the instantiated TSEs contained within the group. This code fragment is

executed once, when the user enters the transaction service. The type of this element is *method*.

- *storeMethod*: Code that is executed in order to store the values of the instantiated TSEs contained within the group. This code fragment is executed once, when the user submits the document modeled by the transaction service. The type of this element is *method*.
- *administrativeData*: Information useful for administrative purposes for the instantiated TSE. This information is used within the SmartGov platform development environment. The type of this element is *administrativeInfo*.
- *instantiatedTSEGroupStatistics*. Definition of statistics that should be collected for the specific instantiated TSE group. The type of this element is *TSEGroupStatistics*.

2.8.7.1 Inheritance Rules for TSE Groups

The SmartGov platform allows for the definition of two types of TSE groups, namely generic and instantiated ones. Generic TSE groups allow for grouping of basic elements (TSEs) into a single entity, while instantiated TSE groups are concrete occurrences of such element groups within transaction services. An instantiated TSE group contains a reference to the generic TSE groups it is modeled after, and this reference implies that the instantiated TSE group inherits the elements and element properties defined in the generic TSE group. The instantiated TSE group, however, is more specific than its generic counterpart, so the SmartGov platform should provide the capability to redefine or supplement various aspects of the inherited element properties and group properties. To this end, the instantiated TSE group contains information slots, which may be filled with values that either override or complement the information inherited from the homonymous data slots of its template. The rules for determining whether the instantiated TSE group information slots override or supplement the inherited values have been defined in the descriptions of the pertinent slots and are summarized in the following table:

Information slot	Override or supplement
Validation rules	Supplement
References to knowledge base	Supplement
References to taxonomy	Supplement

2.8.8 Utility types

This section documents the utility types that are used for modelling the key SmartGov entities described in the previous sections.

2.8.8.1 Complex Type *administrativeInfo*

The *administrativeInfo* type is used to store the last modification date and the workgroup pertaining to the containing entity. This information is used within the SmartGov platform development environment. The XML source for this type is illustrated in the following listing.

Source	<pre><xs:complexType name="administrativeInfo"> <xs:sequence> <xs:element name="lastModificationDate" type="xs:date"/> <xs:element name="workgroup" type="xs:string"/> </xs:sequence> </xs:complexType></pre>
--------	---

2.8.8.2 Complex Type *containedTSE*

The *containedTSE* type is used for specifying the generic TSEs contained in a generic TSE group and the respective default values, which may be different than the default values specified for the generic TSEs. The XML source for this type is illustrated in the following listing.

source	<pre><xs:complexType name="containedTSE"> <xs:sequence> <xs:element name="defaultValue" type="xs:string" minOccurs="0"/> <xs:element name="genericTSEId" type="xs:string"/> </xs:sequence> </xs:complexType></pre>
--------	--

2.8.8.3 Complex Type *formSet*

The *formSet* type is used to store the sets of forms used for each platform a transaction service is disseminated from within. Each form set contains an identification of the target platform and a list of references to the forms. If a service is disseminated through multiple platforms, a different set of forms for each platform will be required. The XML source for this type is illustrated in the following listing.

source	<pre><xs:complexType name="formSet"> <xs:sequence> <xs:element name="formId" type="xs:string" maxOccurs="unbounded"/> <xs:element name="targetPlatform" type="xs:string"/> </xs:sequence> </xs:complexType></pre>
--------	---

2.8.8.4 Complex Type formStatistics

The *formStatistics* type is used for defining the statistics that should be collected for a specific form, when the service is deployed. The statistics supported directly by the SmartGov platform are the following:

- Number of non-empty forms (i.e. forms with at least one field filed-in)
- Number of form invocations (how many times a form was requested)
- Minimum number of non-empty fields
- Maximum number of non-empty fields
- Mean number of non-empty fields
- User time in form (the time users spent within the specific form)

In order to collect a specific statistic, the corresponding element of the XML document should be set to *true*. The XML source for this type is illustrated in the following listing.

source	<pre><xs:complexType name="formStatistics"> <xs:sequence> <xs:element name="numberOfNonEmptyForms" type="xs:boolean" minOccurs="0"/> <xs:element name="numberOfFormInvocations" type="xs:boolean" minOccurs="0"/> <xs:element name="minimumNumberOfNonEmptyFields" type="xs:boolean" minOccurs="0"/> <xs:element name="maximumNumberOfNonEmptyFields" type="xs:boolean" minOccurs="0"/> <xs:element name="meanNumberOfNonEmptyFields" type="xs:boolean" minOccurs="0"/> <xs:element name="userTimeInForm" type="xs:boolean" minOccurs="0"/> </xs:sequence> </xs:complexType></pre>
--------	--

2.8.8.5 Complex Type KUToHelpItem

The *KUToHelpItem* type is used in form schema and describes the mapping of KUs to the elements of the form. Each occurrence of this element maps a KU id to the id of a visual element of the form. The XML source for this type is illustrated in the following listing.

source	<pre><xs:complexType name="KUToHelpItem"> <xs:sequence> <xs:element name="KUId" type="xs:string"/> <xs:element name="helpItemName" type="xs:string"/> </xs:sequence> </xs:complexType></pre>
--------	--

2.8.8.6 Complex Type lifeCycleType

The *LifeCycleType* type is used for storing information pertaining to the life cycle of the containing entity. It covers both the dynamic and the static facets of an object's lifecycle; the static facet models the workgroup, author, creation date, expiration date and service expiration of the object, whereas the dynamic facet models the current state, the last modification date and the id of the user that modified the object. The XML source for this type is illustrated in the following listing.

source	<pre> <xs:complexType name="lifeCycleType"> <xs:sequence> <xs:element name="dynamic"> <xs:complexType> <xs:sequence> <xs:element name="state" type="xs:string"/> <xs:element name="performer" type="xs:string" minOccurs="0"/> <xs:element name="lastModificationDate" type="xs:date"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="static"> <xs:complexType> <xs:sequence> <xs:element name="workGroup" type="xs:string"/> <xs:element name="author" type="xs:string"/> <xs:element name="creationDate" type="xs:date"/> <xs:element name="expirationDate" type="xs:date" minOccurs="0"/> <xs:element name="serviceExpiration" type="xs:boolean" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </pre>
--------	---

2.8.8.7 Complex Type method

The *method* type is used to model actions associated with TSEs and TSE groups. A method has an associated description and may be defined in either of the following two ways:

- Using *SmartGovLang*, a high-level description language defined within SmartGov and useful for defining simple rules. The exact syntax of the language will be determined within the implementation phase. This language may easily be mapped to platform-specific languages (such as Java, PHP, ColdFusion, Javascript). This mapping will be performed at service instantiation phase.
- Using a programming language code (java, javascript, php, etc). These rules may only be executed at platforms that support the specific language; for instance, rules coded in Javascript may only be executed within Web browsers. The code for such a method may either be typed-in directly or by providing a reference to the file containing it.

The XML source for this type, along with the source for the types referenced by it, is illustrated in the following listing.

source	<pre> <xs:complexType name="method"> </pre>
--------	---

	<pre> <xs:sequence> <xs:element name="description" type="multilingualText"/> </xs:choice> <xs:choice> <xs:element name="smartgovLangCheck" type="xs:string"/> <xs:element name="nativeLangCheck" type="nativeCodeFragment"/> </xs:choice> </xs:sequence> </xs:complexType> <xs:complexType name="nativeCodeFragment"> <xs:sequence> <xs:element name="langId" type="xs:string"/> <xs:element name="usefulFor"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="front end"/> <xs:enumeration value="back end"/> <xs:enumeration value="front end and back end"/> </xs:restriction> </xs:simpleType> </xs:element> </xs:choice> <xs:element name="codeText" type="xs:string"/> <xs:element name="fileSpec" type="xs:anyURI"/> </xs:choice> </xs:sequence> </xs:complexType> </pre>
--	--

2.8.8.8 Complex Type multilingualText

The *multilingualText* type is used to model textual information that may need to be stored in multiple languages. Each occurrence of a *multilingualText* describes some textual information in a specific language. The XML source for this type is illustrated in the following listing.

source	<pre> <xs:complexType name="multilingualText"> <xs:sequence> <xs:element name="localeId" type="xs:string"/> <xs:element name="content" type="xs:string"/> </xs:sequence> </xs:complexType> </pre>
--------	---

2.8.8.9 Complex Type repetitionInformation

The *repetitionInformation* is used for specifying properties for tabular areas of administrative forms. The repetition information associated with such constructs is represented by the following items:

- *minOccurrences*: The minimum number of rows that may appear on the form
- *maxOccurrences*: The maximum number of rows that can be placed on a form
- *initialRows*: the number of rows initially displayed
- *uniqueColumn*: one or more columns whose values must be unique within the tabular area. If multiple columns are present, the *value combination* of these columns should be unique. For instance in the *recapitulative statement of intra-community supplies* form, the combination of the *country prefix* and *VAT number* field should be unique.

- *groupControls*: Each such tabular area may need to have controls that enable the user to add rows (if more than the currently displayed are needed) or delete rows (if a large number of empty rows is considered impractical).
- *rowStep*: The number of rows that will be added/deleted in such a case.

The XML source for this type is illustrated in the following listing.

source	<pre> <xs:complexType name="repetitionInformation"> <xs:sequence> <xs:element name="minOccurrences" type="xs:positiveInteger"/> <xs:element name="maxOccurrences" type="xs:positiveInteger"/> <xs:element name="initialRows" type="xs:positiveInteger"/> <xs:element name="uniqueColumn" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="groupControls"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="none"/> <xs:enumeration value="add rows"/> <xs:enumeration value="delete rows"/> <xs:enumeration value="add and delete rows"/> </xs:restriction> </xs:simpleType> </xs:element> <xs:element name="rowStep" type="xs:positiveInteger"/> </xs:sequence> </xs:complexType> </pre>
--------	---

2.8.8.10 Complex Type TSEDataType

The *TSEDataType* type is used for specifying types for generic TSEs. A TSE data type may be a built-in data type or a user-defined data type. Regarding built-in data types, integer numbers, real numbers, text, dates and currency are supported, whereas a user-defined data type may be built on top of these data types, by adding descriptions, restrictions and knowledge.

The XML source for this type, along with the source for the types referenced by it, is illustrated in the following listing.

source	<pre> <xs:complexType name="TSEDataType"> <xs:sequence> <xs:choice> <xs:element name="TSEBuiltInDataType" type="builtInDataType"/> <xs:element name="TSEUserDefinedDataType" type="userDefinedDataType"/> </xs:choice> </xs:sequence> </xs:complexType> <xs:complexType name="userDefinedDataType"> <xs:sequence> <xs:element name="dataTypeId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="baseDataType" type="TSEDataType"/> <xs:element name="description" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="maxLength" type="xs:positiveInteger"/> <xs:element name="availableMethods" type="method" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="validationRule" type="validationMethod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="defaultValue" type="xs:string"/> <xs:element name="valueList" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedKUNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="lifeCycle" type="lifeCycleType"/> </xs:sequence> </pre>
--------	--

	<pre> </xs:complexType> <xs:simpleType name="builtinDataType"> <xs:restriction base="xs:string"> <xs:enumeration value="integer"/> <xs:enumeration value="real"/> <xs:enumeration value="text"/> <xs:enumeration value="date"/> <xs:enumeration value="currency"/> </xs:restriction> </xs:simpleType> </pre>
--	--

2.8.8.11 Complex Type TSEGroupStatistics

The type *TSEGroupStatistics* is used for defining the statistics to be collected for a TSE group. For any group, statistics on the number of non-empty values may be collected, whereas for repeating groups the following statistics may be additionally collected:

- Minimum, maximum and average number of rows
- Minimum, maximum and average value of a column
- Number of non-empty values in a column
- Number of distinct values for a column group
- Number of distinct values for a column

The XML source for this type, along with the source for the types referenced by it, is illustrated in the following listing.

Source	<pre> <xs:complexType name="TSEGroupStatistics"> <xs:sequence> <xs:element name="numberOfNonEmptyValues" type="xs:boolean" minOccurs="0"/> <xs:element name="repetitionStatistics" type="TSEGroupRepeatingStatistics" minOccurs="0"/> </xs:sequence> </xs:complexType><xs:complexType name="TSEGroupRepeatingStatistics"> <xs:sequence> <xs:element name="minimumNumberOfRows" type="xs:boolean" minOccurs="0"/> <xs:element name="maximumNumberOfRows" type="xs:boolean" minOccurs="0"/> <xs:element name="meanNumberOfRows" type="xs:boolean" minOccurs="0"/> <xs:element name="minimumValueOfAColumn" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="maximumValueOfAColumn" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="meanValueOfAColumn" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="numberOfNonEmptyValuesOfAColumn" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="numberOfDistinctValuesOfAColumnGroup" minOccurs="0" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="numberOfDistinctValuesOfAColumn" type="xs:string" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </pre>
--------	---

2.8.8.12 Complex Type TSEStatistics

The *TSEStatistics* type is used for specifying statistics to be collected for a specific TSE. The following statistics may be collected:

- Number of non-empty values
- Number of distinct values
- The sum of all values (for numeric fields only)
- The minimum value entered
- The maximum value entered
- The average of the entered values (for numeric fields only)

Finally, a histogram may be specified by designating the limits of the areas on the X-axis. The XML source for this type is illustrated in the following listing.

source	<pre><xs:complexType name="TSEStatistics"> <xs:sequence> <xs:element name="numberOfNonEmptyValues" type="xs:boolean" minOccurs="0"/> <xs:element name="numberOfDistinctValues" type="xs:boolean" minOccurs="0"/> <xs:element name="sumOfAllValues" type="xs:boolean" minOccurs="0"/> <xs:element name="minimumValue" type="xs:boolean" minOccurs="0"/> <xs:element name="maximumValue" type="xs:boolean" minOccurs="0"/> <xs:element name="meanValue" type="xs:boolean" minOccurs="0"/> <xs:element name="histogramLimits" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType></pre>
--------	--

2.8.8.13 Complex Type TSEToFormElement

This complex type is used in the form schema and describes the mapping of the form elements to the corresponding TSEs. The XML source for this type is illustrated in the following listing.

Source	<pre><xs:complexType name="TSEToFormElement"> <xs:sequence> <xs:element name="instantiatedTSEId" type="xs:string"/> <xs:element name="formElementName" type="xs:string"/> </xs:sequence> </xs:complexType></pre>
--------	--

2.8.8.14 Complex Type TSStatistics

The *TSStatistics* type is used for specifying statistics to be collected for a specific TS. The following statistics may be collected:

- Number of submissions. This may be broken down on daily, weekly, monthly or yearly basis; the total number of submissions may be also requested.
- Number of saved documents that were not finally submitted.
- Number of submissions with warnings. This may be broken down on daily, weekly, monthly or yearly basis; the total number of submissions with warnings may be also requested.
- Number of submissions rejected due to failures in validation checks. This may be broken down on daily, weekly, monthly or yearly basis; the total number of rejected submissions may be also requested.

- Number of sessions during which previously saved documents were edited. This may be broken down on daily, weekly, monthly or yearly basis; the total number of editing sessions may be also requested.
- Number of deletions of previously saved documents. This may be broken down on daily, weekly, monthly or yearly basis; the total number of deletions may be also requested.
- The time needed to fill in and submit the forms of the TS (from the instant that the user starts a TS to the moment of the final document submission).
- The time within the error-correction phase, i.e. the time between the first erroneous submission and the final correct submission.
- System time to handle a submission request.

The XML source for this type is illustrated in the following listing.

source	<pre> <xs:complexType name="TSStatistics"> <xs:sequence> <xs:element name="numberOfSubmissions" type="statisticsPeriod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="numberOfSavedNonSubmittedSessions" type="xs:boolean" minOccurs="0"/> <xs:element name="numberOfRejectedSubmissions" type="statisticsPeriod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="numberOfSubmissionsWithWarnings" type="statisticsPeriod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="numberOfEdits" type="statisticsPeriod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="numberOfDeletions" type="statisticsPeriod" minOccurs="0" maxOccurs="unbounded"/> <xs:element name="fullSubmissionTime" type="xs:boolean" minOccurs="0"/> <xs:element name="errorCorrectionTime" type="xs:boolean" minOccurs="0"/> <xs:element name="handleSubmissionTime" type="xs:boolean" minOccurs="0"/> </xs:sequence> </xs:complexType> <xs:simpleType name="statisticsPeriod"> <xs:restriction base="xs:string"> <xs:enumeration value="day"/> <xs:enumeration value="week"/> <xs:enumeration value="month"/> <xs:enumeration value="year"/> <xs:enumeration value="total"/> </xs:restriction> </xs:simpleType> </pre>
--------	---

2.8.8.15 Complex Type validationMethod

This complex type is used to describe the validation method, the rule used to verify the validity of a combination of data. The code that implements the validation check may be provided either in *SmartGovLang* or using some native language and should return the boolean *true* for success or *false* for failure. If the check is implemented in some native language, it is useful only for the platforms that support the specific language. The specification designates the error message that should be displayed to the user in the case the validation check fails, whether the check should be performed only at the back-end or in the front-end as well, and the severity of check failures. A severity level of "error" indicates

that the failing document may not be submitted, whereas a severity level of "warning" indicates that the user should be alerted but not precluded from submitting the document.

The XML source for this type is illustrated in the following listing.

source	<pre> <xs:complexType name="validationMethod"> <xs:sequence> <xs:element name="ruleId" type="xs:string"/> <xs:element name="code" type="method"/> <xs:element name="validationMessage" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="validateAt"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="front end"/> <xs:enumeration value="back end"/> <xs:enumeration value="front end and back end"/> </xs:restriction> </xs:simpleType> </xs:element> <xs:element name="severity"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="warning"/> <xs:enumeration value="error"/> </xs:restriction> </xs:simpleType> </xs:element> <xs:element name="statistics" type="validationMethodStatistics" minOccurs="0"/> </xs:sequence> </xs:complexType> </pre>
--------	--

2.8.8.16 Complex Type validationMethodStatistics

This type is used for specifying the statistics that should be collected for a method. The overall number of failures for the validation check and the system time spent for the execution of the code may be collected.

The XML source for this type is illustrated in the following listing.

Source	<pre> <xs:complexType name="validationMethodStatistics"> <xs:sequence> <xs:element name="numberOfFailures" type="xs:boolean" minOccurs="0"/> <xs:element name="executionTime" type="xs:boolean" minOccurs="0"/> </xs:sequence> </xs:complexType> </pre>
--------	---

2.8.9 Modeling of Knowledge Units

element **KU**

<p>diagram</p>											
<p>children</p>	<p>KUId header lifeCycle linkedKU sections linkedTaxonomyNode statistics</p>										
<p>identity constraints</p>	<table border="1"> <thead> <tr> <th>key</th> <th>Name</th> <th>Refer</th> <th>Selector</th> <th>Field(s)</th> </tr> </thead> <tbody> <tr> <td>key</td> <td>KUKey</td> <td></td> <td>../KU</td> <td>KUId</td> </tr> </tbody> </table>	key	Name	Refer	Selector	Field(s)	key	KUKey		../KU	KUId
key	Name	Refer	Selector	Field(s)							
key	KUKey		../KU	KUId							
<p>annotation</p>	<p>documentation Knowledge Unit</p>										
<p>source</p>	<pre> <xs:element name="KU"> <xs:annotation> <xs:documentation>Knowledge Unit</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="KUId" type="xs:string"/> <xs:element name="header"> <xs:complexType> <xs:sequence> <xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="abstract" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="type"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="Best Practice"/> <xs:enumeration value="Example"/> <xs:enumeration value="Help"/> <xs:enumeration value="Just-in-time Training"/> </xs:restriction> </xs:simpleType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>										

```

<xs:enumeration value="Lessons Learned"/>
<xs:enumeration value="Storytelling"/>
<xs:enumeration value="Troubleshooting"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="lifeCycle" type="lifeCycleType"/>
<xs:element name="linkedKU" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="sections" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="addressedTo">
        <xs:complexType>
          <xs:sequence>
            <xs:choice>
              <xs:element name="allRoles" type="xs:boolean" default="true"/>
              <xs:element name="role" type="xs:string" maxOccurs="unbounded"/>
            </xs:choice>
            <xs:element name="comments" type="xs:string" minOccurs="0">
              <xs:annotation>
                <xs:documentation>To allow special circumstances to be documented</xs:documentation>
              </xs:annotation>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="langDescription" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="localeId" type="xs:string"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="content" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="link" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="multilingualText" maxOccurs="unbounded"/>
            <xs:element name="url" maxOccurs="unbounded">
              <xs:annotation>
                <xs:documentation>The name of the Link could be in several languages, but the url leads to one or more
links depending on whether there are available versions with the same content in one or more languages.
              </xs:documentation>
            </xs:annotation>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```


```

<xs:sequence>
  <xs:element name="localeId" type="xs:string"/>
  <xs:element name="urlAddress" type="xs:anyURI"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="statistics">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="workflow">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="taskLog" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="performer" type="xs:string"/>
                  <xs:element name="date" type="xs:dateTime"/>
                  <xs:element name="newState" type="xs:string"/>
                  <xs:element name="comments" type="xs:string" minOccurs="0"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="metrics">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="complexity" type="levelType"/>
            <xs:element name="relevance" type="levelType"/>
            <xs:element name="richness" type="levelType"/>
            <xs:element name="grade">
              <xs:annotation>
                <xs:documentation>Type of Knowledge. Core: Basic. Advance: new and tested ideas. Innovative: new
ideas (not tested yet)</xs:documentation>
              </xs:annotation>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

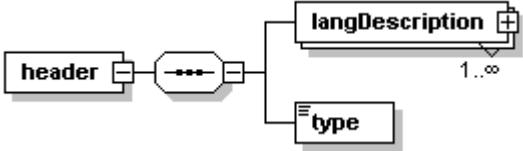
```


	<pre> </xs:simpleType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="deliveryEnvironmentStatistics"> <xs:complexType> <xs:sequence> <xs:element name="lastAccess" type="xs:boolean"/> <xs:element name="numberOfInvocations" type="xs:boolean"/> <xs:element name="allowEndUserComments" type="xs:boolean"/> <xs:element name="allowEndUserRating" type="xs:boolean"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> <xs:key name="KUKey"> <xs:selector xpath="."/> <xs:field xpath="KUId"/> </xs:key> </xs:element> </pre>
	This element represents a Knowledge Unit object of the SmartGov platform.

element KU/KUId

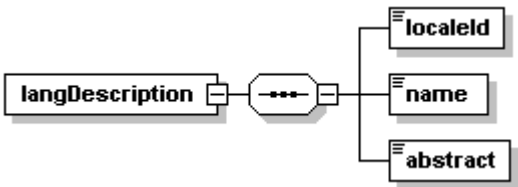
diagram	
type	xs:string
source	<code><xs:element name="KUId" type="xs:string"/></code>
	The id of the KU.

element KU/header

diagram	
children	langDescription type


<p>source</p>	<pre> <xs:element name="header"> <xs:complexType> <xs:sequence> <xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="abstract" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="type"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="Best Practice"/> <xs:enumeration value="Example"/> <xs:enumeration value="Help"/> <xs:enumeration value="Just-in-time Training"/> <xs:enumeration value="Lessons Learned"/> <xs:enumeration value="Storytelling"/> <xs:enumeration value="Troubleshooting"/> </xs:restriction> </xs:simpleType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>The header contains the name, and the type of the KU.</p>

element KU/header/langDescription


<p>diagram</p>	 <pre> graph LR langDescription[langDescription] --- ellipsis[...] ellipsis --- localeId[localeId] ellipsis --- name[name] ellipsis --- abstract[abstract] </pre>
<p>children</p>	<p>localeId name abstract</p>
<p>source</p>	<pre> <xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="abstract" type="xs:string"/> </xs:sequence> </pre>

	<pre></xs:complexType> </xs:element></pre>
	The name and abstract of the KU that specifies what the KU represents.


element KU/header/langDescription/localeId

diagram	
type	xs:string
source	<pre><xs:element name="localeId" type="xs:string"/></pre>
	The id of the language the content (text) is written in

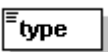
element KU/header/langDescription/name

diagram	
type	xs:string
source	<pre><xs:element name="name" type="xs:string"/></pre>
	The name of the KU.

element KU/header/langDescription/abstract

diagram	
type	xs:string
source	<pre><xs:element name="abstract" type="xs:string"/></pre>
	The abstract of the KU.

element KU/header/type

diagram	
type	restriction of xs:string
facets	<ul style="list-style-type: none"> enumeration Best Practice enumeration Example enumeration Help enumeration Just-in-time Training

	<p>enumeration Lessons Learned</p> <p>enumeration Storytelling</p> <p>enumeration Troubleshooting</p>
source	<pre><xs:element name="type"> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="Best Practice"/> <xs:enumeration value="Example"/> <xs:enumeration value="Help"/> <xs:enumeration value="Just-in-time Training"/> <xs:enumeration value="Lessons Learned"/> <xs:enumeration value="Storytelling"/> <xs:enumeration value="Troubleshooting"/> </xs:restriction> </xs:simpleType> </xs:element></pre>
	<p>The Type of the KU. This type can be:</p> <ul style="list-style-type: none"> • Best practices. They are not static documents describing "how to do x", but rather collections of guidelines, based on ever-evolving experiences in a particular domain. • Examples. Practical cases that shows a particular solution. • Help. Information to explain the use of a specific element (TSs, TSEs, Forms KUs...). Those are the KUs that are more suitable to deploy with the service. • Just in time training. Condensed pieces of training to speed up the acquisition of new competencies or abilities for the job. • Lessons learned. A record of the success and failure experiences of the organization. • Storytelling. Narratives to communicate complex ideas in simple terms. • Troubleshooting. A record of problems and solutions for a particular context.

element KUlifecycle

diagram	<p>The diagram shows a class hierarchy. A box labeled 'lifeCycle' is connected to a box labeled 'lifeCycleType' (which is highlighted in yellow). 'lifeCycleType' has two subclasses: 'dynamic' and 'static', both indicated by plus signs in boxes. 'lifeCycle' has a dashed line with an arrow pointing to 'lifeCycleType', indicating a generalization relationship.</p>
type	lifeCycleType
children	dynamic static
source	<pre><xs:element name="lifeCycle" type="lifeCycleType"/></pre>
	<p>Contains information about the life-cycle of a SmartGov object (KU). This information could be static or dynamic:</p> <ul style="list-style-type: none"> • Static: general information about the element that it does not change frequently. <ul style="list-style-type: none"> ○ workgroup: the Work Group under what the element has been created. It can be used to know the security related with the element ○ author: the SmartGov user that creates the element

	<ul style="list-style-type: none"> ○ creationDate: date of the creation of the element. ○ expirationDate: date of expiration of the element. When the element expires, it should not be shown to the end-user, and an alarm should be sent to the administrator. ○ serviceExpiration: indicates that the element (a KU) should be deleted when all its related services expired. • Dynamic: information about the current status of the element. <ul style="list-style-type: none"> ○ state: if the element is involved in a life-cycle workflow (KUs and TSs), indicates the current state of the element ○ performer: if the element is involved in a life-cycle workflow (KUs and TSs), indicates the user that has the current task. If the task has not been taken by a specific user (belongs to a role), the performer will have null value. ○ lastModificationDate: date of last modification. It will be used to control concurrence.
--	--

element KU/linkedKU

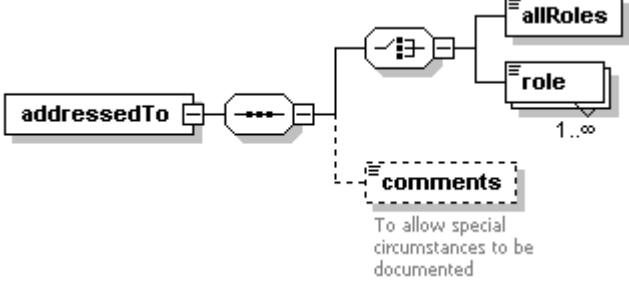
diagram	
type	xs:string
source	<code><xs:element name="linkedKU" type="xs:string" minOccurs="0" maxOccurs="unbounded"/></code>
	The id of the KUs to which the current KU is attached.

element KU/sections


diagram	
children	addressedTo langDescription link
source	<pre> <xs:element name="sections" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="addressedTo"> <xs:complexType> <xs:sequence> <xs:choice> <xs:element name="allRoles" type="xs:boolean" default="true"/> <xs:element name="role" type="xs:string" maxOccurs="unbounded"/> </xs:choice> <xs:element name="comments" type="xs:string" minOccurs="0"> </pre>

	<pre> <xs:annotation> <xs:documentation>To allow special circumstances to be documented</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeId" type="xs:string"/> <xs:element name="title" type="xs:string"/> <xs:element name="content" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="link" minOccurs="0" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="name" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="url" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>The name of the Link could be in several languages, but the url leads to one or more links depending on whether there are available versions with the same content in one or more languages. </xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>A section represents a role related information. The KU could be composed by several sections whose visibility may depend on the user's role.</p>

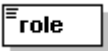
element **KU/sections/addressedTo**

<p>diagram</p>	
<p>children</p>	<p>allRoles role comments</p>
<p>source</p>	<pre><xs:element name="addressedTo"> <xs:complexType> <xs:sequence> <xs:choice> <xs:element name="allRoles" type="xs:boolean" default="true"/> <xs:element name="role" type="xs:string" maxOccurs="unbounded"/> </xs:choice> <xs:element name="comments" type="xs:string" minOccurs="0"> <xs:annotation> <xs:documentation>To allow special circumstances to be documented</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element></pre>
	<p>A KU may have different visibilities depending on the roles of the users that have access to the KU. For instance, It may exist one section with allRoles visibility (all users can access), and more sections with specific information that just one or several SmartGov roles will be able to retrieve.</p>


element **KU/sections/addressedTo/allRoles**

<p>diagram</p>	
<p>type</p>	<p>xs:Boolean</p>
<p>source</p>	<pre><xs:element name="allRoles" type="xs:boolean" default="true"/></pre>
	<p>If it is true, means that all users are able to retrieve the information of the section.</p>

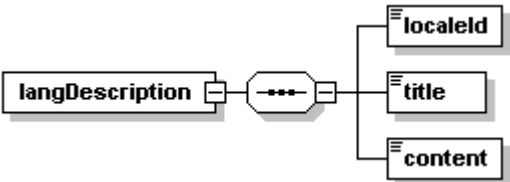
element KU/sections/addressedTo/role

diagram	
type	Xs:string
source	<code><xs:element name="role" type="xs:string" maxOccurs="unbounded"/></code>
	A list of SmartGov roles that will be able to retrieve the information of the section.

element KU/sections/addressedTo/comments


diagram	
type	xs:string
annotation	documentation To allow special circumstances to be documented
source	<code><xs:element name="comments" type="xs:string" minOccurs="0"> <xs:annotation> <xs:documentation>To allow special circumstances to be documented</xs:documentation> </xs:annotation> </xs:element></code>
	Allow special circumstances related with the visibility of the section to be documented.

element KU/sections/langDescription

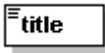
diagram	
children	localeId title content
source	<code><xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeId" type="xs:string"/> <xs:element name="title" type="xs:string"/> <xs:element name="content" type="xs:string"/> </xs:sequence></code>

	<pre></xs:complexType> </xs:element></pre>
	Title and textual content of the section.


element **KU/sections/langDescription/localeId**

diagram	
type	xs:string
source	<pre><xs:element name="localeId" type="xs:string"/></pre>
	The id of the language the content (text) is written in

element **KU/sections/langDescription/title**

diagram	
type	xs:string
source	<pre><xs:element name="title" type="xs:string"/></pre>
	Title of the section. Just for easy identification reasons.

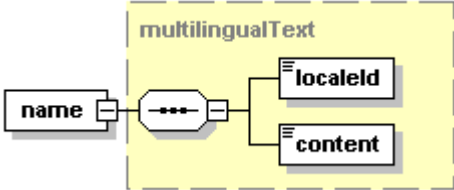
element **KU/sections/langDescription/content**

diagram	
type	xs:string
source	<pre><xs:element name="content" type="xs:string"/></pre>
	Textual content of the section.

element **KU/sections/link**

<p>diagram</p>	 <p>The name of the Link could be in several languages, but the url leads to one or more links depending on whether there are available versions with the same content in one or more languages.</p>
<p>children</p>	<p>name url</p>
<p>source</p>	<pre><xs:element name="link" minOccurs="0" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="name" type="multilingualText" maxOccurs="unbounded"/> <xs:element name="url" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>The name of the Link could be in several languages, but the url leads to one or more links depending on whether there are available versions with the same content in one or more languages. </xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element></pre>
	<p>Links or attachments related with the KU.</p>

element **KU/sections/link/name**

<p>diagram</p>	
----------------	---

type	multilingualText
children	localeId content
source	<code><xs:element name="name" type="multilingualText" maxOccurs="unbounded"/></code>
	Name of the link to be shown to the user. It should be shown in the language of the user or in a default language.

element KU/sections/link/url

diagram	<p>The name of the Link could be in several languages, but the url leads to one or more links depending on whether there are available versions with the same content in one or more languages.</p>
children	localeId urlAddress
annotation	documentation The name of the Link could be in several languages, but the url leads to one or more links depending on whether there are available versions with the same content in one or more languages.
source	<pre> <xs:element name="url" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>The name of the Link could be in several languages, but the url leads to one or more links depending on whether there are available versions with the same content in one or more languages. </xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="localeId" type="xs:string"/> <xs:element name="urlAddress" type="xs:anyURI"/> </xs:sequence> </xs:complexType> </xs:element> </pre>
	The page or file linked may be in more than one available languages (different URLs).

element KU/sections/link/url/localeId

diagram	
type	xs:string
source	<code><xs:element name="localeId" type="xs:string"/></code>
	The id of the language the content (text) is written in

element KU/sections/link/url/urlAddress

diagram	
type	xs:anyURI
source	<code><xs:element name="urlAddress" type="xs:anyURI"/></code>
	The URL address where the attachment is located. It should be available from the SmartGov or Delivery environments, depending on the location of the KU.

element KU/linkedTaxonomyNode

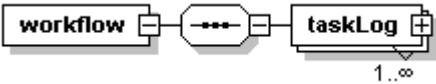
diagram	
type	xs:string
source	<code><xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"/></code>
	The id of the taxonomyNode to which the KU is attached.

element KU/statistics

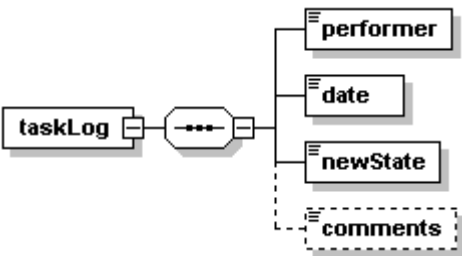
diagram	
children	workflow metrics deliveryEnvironmentStatistics
source	<code><xs:element name="statistics"> <xs:complexType> <xs:sequence> <xs:element name="workflow"> <xs:complexType> <xs:sequence> <xs:element name="taskLog" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="performer" type="xs:string"/> <xs:element name="date" type="xs:dateTime"/> <xs:element name="newState" type="xs:string"/> <xs:element name="comments" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element></code>

	<pre> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="metrics"> <xs:complexType> <xs:sequence> <xs:element name="complexity" type="levelType"/> <xs:element name="relevance" type="levelType"/> <xs:element name="richness" type="levelType"/> <xs:element name="grade"> <xs:annotation> <xs:documentation>Type of Knowledge. Core: Basic. Advance: new and tested ideas. Innovative: new ideas (not tested yet)</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="deliveryEnvironmentStatistics"> <xs:complexType> <xs:sequence> <xs:element name="lastAccess" type="xs:boolean"/> <xs:element name="numberOfInvocations" type="xs:boolean"/> <xs:element name="allowEndUserComments" type="xs:boolean"/> <xs:element name="allowEndUserRating" type="xs:boolean"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>The statistics defined for the KU. There are two kind of KU Statistics:</p> <ul style="list-style-type: none"> • SmartGov Design Environment Statistics: <ul style="list-style-type: none"> ○ Workflow: statistics related with the KU life-cycle ○ Metrics: information to be filled in by knowledge administrators • Delivery Environment Statistics: <ul style="list-style-type: none"> ○ Definition of the KU statistics to be collected in the Delivery Environment (end-user statistics).

element **KU/statistics/workflow**


diagram	
children	taskLog
source	<pre> <xs:element name="workflow"> <xs:complexType> <xs:sequence> <xs:element name="taskLog" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="performer" type="xs:string"/> <xs:element name="date" type="xs:dateTime"/> <xs:element name="newState" type="xs:string"/> <xs:element name="comments" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>Statistics related with the KU life-cycle in the SmartGov Design Environment.</p>

element **KU/statistics/workflow/taskLog**

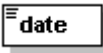
diagram	
children	performer date newState comments
source	<pre> <xs:element name="taskLog" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="performer" type="xs:string"/> <xs:element name="date" type="xs:dateTime"/> <xs:element name="newState" type="xs:string"/> <xs:element name="comments" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element> </pre>

	<pre></xs:complexType> </xs:element></pre>
	Log information to be filled in automatically after every step or task in the KU life-cycle


element **KU/statistics/workflow/taskLog/performer**

diagram	
type	xs:string
source	<pre><xs:element name="performer" type="xs:string"/></pre>
	Life-cycle log information: User that performs the task.


element **KU/statistics/workflow/taskLog/date**

diagram	
type	xs:dateTime
source	<pre><xs:element name="date" type="xs:dateTime"/></pre>
	Life-cycle log information: time when the task was done.

element **KU/statistics/workflow/taskLog/newState**

diagram	
type	xs:string
source	<pre><xs:element name="newState" type="xs:string"/></pre>
	Life-cycle log information: new state of the KU after the completion of the task.


element **KU/statistics/workflow/taskLog/comments**

diagram	
type	xs:string
source	<pre><xs:element name="comments" type="xs:string" minOccurs="0"/></pre>
	Life-cycle log information: user comments captured during the task.

element **KU/statistics/metrics**


<p>diagram</p>	 <p>Type of Knowledge. Core: Basic. Advance: new and tested ideas. Innovative: new ideas (not tested yet)</p>
<p>children</p>	<p>complexity relevance richness grade</p>
<p>source</p>	<pre><xs:element name="metrics"> <xs:complexType> <xs:sequence> <xs:element name="complexity" type="levelType"/> <xs:element name="relevance" type="levelType"/> <xs:element name="richness" type="levelType"/> <xs:element name="grade"> <xs:annotation> <xs:documentation>Type of Knowledge. Core: Basic. Advance: new and tested ideas. Innovative: new ideas (not tested yet)</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="Innovative"/> <xs:enumeration value="Advance"/> <xs:enumeration value="Core"/> </xs:restriction> </xs:simpleType> </xs:element> </xs:sequence> </xs:complexType> </xs:element></pre>
	<p>Knowledge administrator statistics. The administrator should fill in or review these concepts after reviewing the statistics.</p>

element **KU/statistics/metrics/complexity**


<p>diagram</p>	
<p>type</p>	<p>levelType</p>
<p>facets</p>	<p>enumeration Low</p>

	enumeration Medium enumeration High
source	<xs:element name="complexity" type="levelType"/>
	Knowledge administrator statistics. The complexity of the KU. The administrator should fill in or review these concepts after reviewing the statistics.


element **KU/statistics/metrics/relevance**

diagram	
type	levelType
facets	enumeration Low enumeration Medium enumeration High
source	<xs:element name="relevance" type="levelType"/>
	Knowledge administrator statistics. The relevance of the KU. The administrator should fill in or review these concepts after reviewing the statistics.

element **KU/statistics/metrics/richness**

diagram	
type	levelType
facets	enumeration Low enumeration Medium enumeration High
source	<xs:element name="richness" type="levelType"/>
	Knowledge administrator statistics. The richness of the KU. The administrator should fill in or review these concepts after reviewing the statistics.

element **KU/statistics/metrics/grade**


diagram	 Type of Knowledge, Core: Basic, Advance: new and tested ideas, Innovative: new ideas (not tested yet)
type	restriction of xs:string
facets	enumeration Innovative enumeration Advance enumeration Core

	enumeration Core
annotation	documentation Type of Knowledge. Core: Basic. Advance: new and tested ideas. Innovative: new ideas (not tested yet)
source	<pre> <xs:element name="grade"> <xs:annotation> <xs:documentation>Type of Knowledge. Core: Basic. Advance: new and tested ideas. Innovative: new ideas (not tested yet)</xs:documentation> </xs:annotation> <xs:simpleType> <xs:restriction base="xs:string"> <xs:enumeration value="Innovative"/> <xs:enumeration value="Advance"/> <xs:enumeration value="Core"/> </xs:restriction> </xs:simpleType> </xs:element> </pre>
	Knowledge administrator statistics. The grade or type of knowledge of the KU. The administrator should fill in or review these concepts after reviewing the statistics.

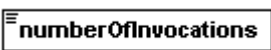
element **KU/statistics/deliveryEnvironmentStatistics**

diagram	<pre> classDiagram class deliveryEnvironmentStatistics { lastAccess numberOfInvocations allowEndUserComments allowEndUserRating } </pre>
children	lastAccess numberOfInvocations allowEndUserComments allowEndUserRating
source	<pre> <xs:element name="deliveryEnvironmentStatistics"> <xs:complexType> <xs:sequence> <xs:element name="lastAccess" type="xs:boolean"/> <xs:element name="numberOfInvocations" type="xs:boolean"/> <xs:element name="allowEndUserComments" type="xs:boolean"/> <xs:element name="allowEndUserRating" type="xs:boolean"/> </xs:sequence> </xs:complexType> </xs:element> </pre>
	The Delivery Environment statistics to be defined for the KU.

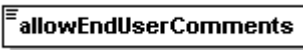
element **KU/statistics/deliveryEnvironmentStatistics/lastAccess**

diagram	
type	xs:boolean
source	<code><xs:element name="lastAccess" type="xs:boolean"/></code>
	Date of last access to the KU.

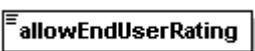
element **KU/statistics/deliveryEnvironmentStatistics/numberOfInvocations**

diagram	
type	xs:boolean
source	<code><xs:element name="numberOfInvocations" type="xs:boolean"/></code>
	Number of invocations of the KU.

element **KU/statistics/deliveryEnvironmentStatistics/allowEndUserComments**

diagram	
type	xs:boolean
source	<code><xs:element name="allowEndUserComments" type="xs:boolean"/></code>
	Allow the possibility of collect end-user comments.

element **KU/statistics/deliveryEnvironmentStatistics/allowEndUserRating**

diagram	
type	xs:boolean
source	<code><xs:element name="allowEndUserRating" type="xs:boolean"/></code>
	Allow the possibility of collect end-user opinion about the KU, according to a specific rating (for instance, 1 to 5).

simpleType **levelType**

type	restriction of xs:string
used by	elements KU/statistics/metrics/complexity KU/statistics/metrics/relevance KU/statistics/metrics/richness
facets	enumeration Low enumeration Medium enumeration High
source	<pre><xs:simpleType name="levelType"> <xs:restriction base="xs:string"> <xs:enumeration value="Low"/> <xs:enumeration value="Medium"/> <xs:enumeration value="High"/> </xs:restriction> </xs:simpleType></pre>
	Level Type related with some of the KU statistics.

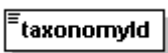
2.8.10 Modeling of Taxonomy

element **taxonomy**

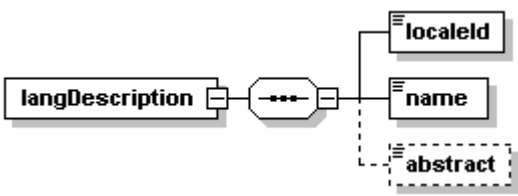
diagram	
children	taxonomyId langDescription firstLevelTaxonomyNode
annotation	documentation Define the SmartGov taxonomy structure
source	<pre><xs:element name="taxonomy"> <xs:annotation> <xs:documentation>Define the SmartGov taxonomy structure</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="taxonomyId" type="xs:string"/> <xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeId" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element></pre>

	<pre> <xs:element name="name" type="xs:string"/> <xs:element name="abstract" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="firstLevelTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>List of firts level taxonomy nodes Id (the taxonomyNodeId should correspond with an existing taxonomy Node) </xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>A taxonomy in SmartGov represents a categorization of the reality, a particular form of classifying SmartGov objects (KUs, TSs, TSEs, TSE Groups and Forms).</p> <p>In the SmartGov environment is a collection of taxonomy nodes in a tree-like form.</p>

element taxonomy/taxonomyId


diagram	
type	xs:string
source	<code><xs:element name="taxonomyId" type="xs:string"/></code>
	The Id of the Taxonomy element.

element taxonomy/langDescription


diagram	
children	localeId name abstract
source	<pre> <xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeId" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="abstract" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType> </pre>

	<code></xs:element></code>
	The name and abstract of the taxonomy that specifies what the taxonomy represents.


element `taxonomy/langDescription/localeId`

diagram	
type	<code>xs:string</code>
source	<code><xs:element name="localeId" type="xs:string"/></code>
	The id of the language the content (text) is written in.


element `taxonomy/langDescription/name`

diagram	
type	<code>xs:string</code>
source	<code><xs:element name="name" type="xs:string"/></code>
	The name of the Taxonomy.

element `taxonomy/langDescription/abstract`

diagram	
type	<code>xs:string</code>
source	<code><xs:element name="abstract" type="xs:string" minOccurs="0"/></code>
	The abstract of the Taxonomy.

element `taxonomy/firstLevelTaxonomyNode`

diagram	 List of first level taxonomy nodes Id (the taxonomyNodeId should correspond with an existing taxonomy Node)
type	<code>xs:string</code>
annotation	documentation List of first level taxonomy nodes Id (the taxonomyNodeId should correspond with an existing taxonomy Node)

source	<pre><xs:element name="firstLevelTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>List of first level taxonomy nodes Id (the taxonomyNodeid should correspond with an existing taxonomy Node) </xs:documentation> </xs:annotation> </xs:element></pre>
	<p>List of first level taxonomy nodes Id. The taxonomyNodeid should correspond with an existing taxonomy Node in taxonomyNodes.xsd XML files. The taxonomy will store just the first level of nodes.</p>


2.8.11 Modeling of Taxonomy node

element taxonomyNode

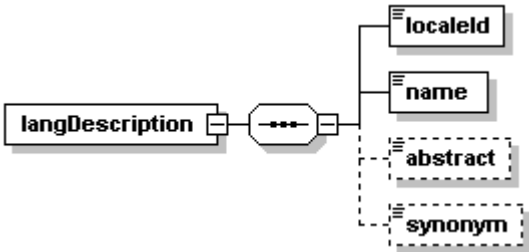
diagram	<p>Define each taxonomy node or subnode. It can be used in different taxonomies.</p> <p>The same as the KU case</p>			
children	taxonomyNodeid langDescription linkedTaxonomyNode			
identity constraints	key	Name taxonomyNodeKey	Refer	Selector ../taxonomyNode Field(s) taxonomyNodeid
annotation	documentation Define each taxonomy node or subnode. It can be used in different taxonomies.			
source	<pre><xs:element name="taxonomyNode"> <xs:annotation> <xs:documentation>Define each taxonomy node or subnode. It can be used in different taxonomies.</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="taxonomyNodeid" type="xs:string"/> <xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeid" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="abstract" type="xs:string" minOccurs="0"/> <xs:element name="synonym" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"> <xs:annotation></pre>			

	<pre> <xs:documentation>The same as the KU case</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> <xs:key name="taxonomyNodeKey"> <xs:selector xpath="."/> <xs:field xpath="taxonomyNodeid"/> </xs:key> </xs:element> </pre>
	<p>Define each taxonomy node or sub-node and its dependencies. It can be used in different taxonomies.</p>


element taxonomyNode/taxonomyNodeid

<p>diagram</p>	
<p>type</p>	<p>xs:string</p>
<p>source</p>	<pre><xs:element name="taxonomyNodeid" type="xs:string"/></pre>
	<p>The id of the Taxonomy Node element.</p>

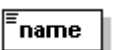
element taxonomyNode/langDescription

<p>diagram</p>	
<p>children</p>	<p>localeid name abstract synonym</p>
<p>source</p>	<pre> <xs:element name="langDescription" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="localeid" type="xs:string"/> <xs:element name="name" type="xs:string"/> <xs:element name="abstract" type="xs:string" minOccurs="0"/> <xs:element name="synonym" type="xs:string" minOccurs="0"/> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>The name, abstract, and synonym of the taxonomy node, that specifies what the taxonomy node represents.</p>


element **taxonomyNode/langDescription/localeId**

diagram	
type	xs:string
source	<code><xs:element name="localeId" type="xs:string"/></code>
	The id of the language the content (text) is written in.

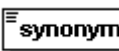
element **taxonomyNode/langDescription/name**

diagram	
type	xs:string
source	<code><xs:element name="name" type="xs:string"/></code>
	The name of the Taxonomy Node.


element **taxonomyNode/langDescription/abstract**

diagram	
type	xs:string
source	<code><xs:element name="abstract" type="xs:string" minOccurs="0"/></code>
	The abstract of the Taxonomy Node.

element **taxonomyNode/langDescription/synonym**

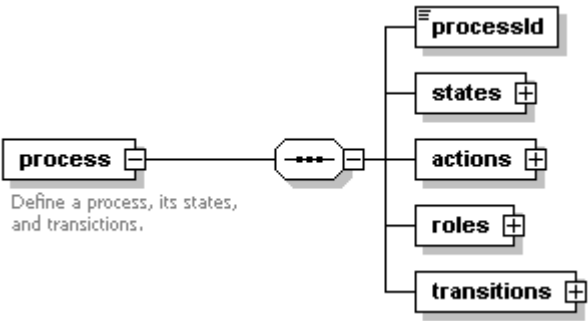
diagram	
type	xs:string
source	<code><xs:element name="synonym" type="xs:string" minOccurs="0"/></code>
	A synonym of the Taxonomy Node.

element **taxonomyNode/linkedTaxonomyNode**

diagram	
type	xs:string
annotation	documentation The same as the KU case
source	<pre><xs:element name="linkedTaxonomyNode" type="xs:string" minOccurs="0" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>The same as the KU case</xs:documentation> </xs:annotation> </xs:element></pre>
	The list of Taxonomy Nodes Ids that are child nodes of the current element.

2.8.12 Modeling of Workflow

element **process**

diagram					
children	processId states actions roles transitions				
identity constraints		Name	Refer	Selector	Field(s)
	key	stateKey		./states/state	.
	key	actionKey		./actions/action	.
	key	roleKey		./roles/role/roleId	.
	keyref	initialStateKeyRef	stateKey	./transitions/transition/ initialState	.
	keyref	finalStateKeyRef	stateKey	./transitions/transition/ finalState	.
	keyref	actionKeyRef	actionKey	./transitions/transition/ action	.
	keyref	roleKeyRef	roleKey	./transitions/transition/ role	.
	unique	transitionUnique		./transitions/transition	initialState action

annotation	documentation Define a process, its states, and transicions.
source	<pre> <xs:element name="process"> <xs:annotation> <xs:documentation>Define a process, its states, and transicions.</xs:documentation> </xs:annotation> <xs:complexType> <xs:sequence> <xs:element name="processId" type="xs:string"/> <xs:element name="states"> <xs:complexType> <xs:sequence> <xs:element name="state" type="xs:string" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="actions"> <xs:complexType> <xs:sequence> <xs:element name="action" type="xs:string" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="roles"> <xs:complexType> <xs:sequence> <xs:element name="roleScope"> <xs:complexType> <xs:sequence> <xs:element name="roleScopeld" type="xs:string"/> <xs:element name="roleScopeDescription" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="role" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="roleId" type="xs:string"/> <xs:element name="roleDescription" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="transitions"> <xs:complexType> <xs:sequence> <xs:element name="transition" maxOccurs="unbounded"> </pre>

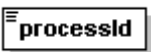
```

<xs:complexType>
  <xs:sequence>
    <xs:element name="initialState" type="xs:string"/>
    <xs:element name="finalState" type="xs:string"/>
    <xs:element name="action" type="xs:string"/>
    <xs:element name="role" type="xs:string" maxOccurs="unbounded">
      <xs:annotation>
        <xs:documentation>A unique restriction has been added at transition level to assure that the roles that are
able to perform an action cannot be repeated.</xs:documentation>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:unique name="nonRepeatingRoles">
  <xs:selector xpath="/role"/>
  <xs:field xpath="."/>
</xs:unique>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:key name="stateKey">
  <xs:selector xpath="/states/state"/>
  <xs:field xpath="."/>
</xs:key>
<xs:key name="actionKey">
  <xs:selector xpath="/actions/action"/>
  <xs:field xpath="."/>
</xs:key>
<xs:key name="roleKey">
  <xs:selector xpath="/roles/role/roleId"/>
  <xs:field xpath="."/>
</xs:key>
<xs:keyref name="initialStateKeyRef" refer="stateKey">
  <xs:selector xpath="/transitions/transition/initialState"/>
  <xs:field xpath="."/>
</xs:keyref>
<xs:keyref name="finalStateKeyRef" refer="stateKey">
  <xs:selector xpath="/transitions/transition/finalState"/>
  <xs:field xpath="."/>
</xs:keyref>
<xs:keyref name="actionKeyRef" refer="actionKey">
  <xs:selector xpath="/transitions/transition/action"/>
  <xs:field xpath="."/>
</xs:keyref>
<xs:keyref name="roleKeyRef" refer="roleKey">
  <xs:selector xpath="/transitions/transition/role"/>

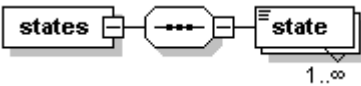
```

	<pre> <xs:field xpath="."/> </xs:keyref> <xs:unique name="transitionUnique"> <xs:selector xpath="./transitions/transition"/> <xs:field xpath="initialState"/> <xs:field xpath="action"/> </xs:unique> </xs:element> </pre>
	<p>This element represents a process in the SmartGov platform. This type of XML document is used to describe KU and TS life-cycles, and any additional process that arises during development.</p> <p>Several key and keyref constraints have been defined. These constraints are used to keep consistence between states, performers, and actions definitions, and the defined transitions, which involve these three types of elements. With these constraints, non existing states, performers or actions, and duplicated are avoided.</p> <p>There is also a unique constraint, to avoid transitions duplicated with the same initialState and action, because from one state an action always leads to the same final state, regardless of the performer.</p>


element process/processId

<p>diagram</p>	
<p>type</p>	<p>xs:string</p>
<p>source</p>	<pre> <xs:element name="processId" type="xs:string"/> </pre>
	<p>The id of the process</p>

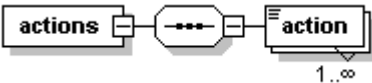
element process/states

<p>diagram</p>	
<p>children</p>	<p>state</p>
<p>source</p>	<pre> <xs:element name="states"> <xs:complexType> <xs:sequence> <xs:element name="state" type="xs:string" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>This element groups all the different states that the process can be in.</p>


element process/states/state

diagram	
type	xs:string
source	<code><xs:element name="state" type="xs:string" maxOccurs="unbounded"/></code>
	This element represents one of the different states that the process can be in during its life.

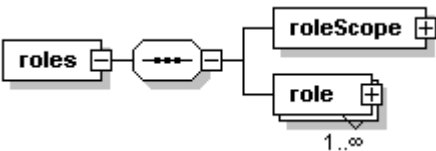
element process/actions

diagram	
children	action
source	<code><xs:element name="actions"> <xs:complexType> <xs:sequence> <xs:element name="action" type="xs:string" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType> </xs:element></code>
	This element groups all the actions that can be performed during the process. Actions will be available depending on the current state.

element process/actions/action

diagram	
type	xs:string
source	<code><xs:element name="action" type="xs:string" maxOccurs="unbounded"/></code>
	One of the actions that can be performed during the process life.

element process/roles

diagram	
---------	---


children	roleScope role
source	<pre> <xs:element name="roles"> <xs:complexType> <xs:sequence> <xs:element name="roleScope"> <xs:complexType> <xs:sequence> <xs:element name="roleScopeld" type="xs:string"/> <xs:element name="roleScopeDescription" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> <xs:element name="role" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="roleld" type="xs:string"/> <xs:element name="roleDescription" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>This element groups all the information regarding the roles involved in the process.</p> <p>First of all, it specifies the ‘scope’ of the defined roles. This scope is used in the roles system to group the roles relating to the same field, the same sphere: all the roles defined for a service, for a process... For example, in the roles system will exist the ‘SmartGov’ scope, with the Manager, Expert, IT Staff and End User roles.</p> <p>After the scope, a list of the participating roles is defined.</p>

element process/roles/roleScope

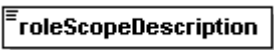
diagram	<pre> classDiagram class roleScope { roleScopeld roleScopeDescription } </pre>
children	roleScopeld roleScopeDescription
source	<pre> <xs:element name="roleScope"> <xs:complexType> <xs:sequence> <xs:element name="roleScopeld" type="xs:string"/> <xs:element name="roleScopeDescription" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element> </pre>

	As described in the previous paragraph, this element describes the field where the defined roles make sense.
--	--

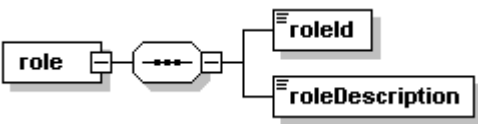
element process/roles/roleScope/roleScopeld

diagram	
type	xs:string
source	<code><xs:element name="roleScopeld" type="xs:string"/></code>
	The id of the Role Scope.


element process/roles/roleScope/roleScopeDescription

diagram	
type	xs:string
source	<code><xs:element name="roleScopeDescription" type="xs:string"/></code>
	A description of the role scope, probably describing where these roles are applicable.

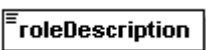
element process/roles/role

diagram	
children	roleId roleDescription
source	<pre><xs:element name="role" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="roleId" type="xs:string"/> <xs:element name="roleDescription" type="xs:string"/> </xs:sequence> </xs:complexType> </xs:element></pre>
	One of the different roles participating in the process.

element **process/roles/role/roleId**

diagram	
type	xs:string
source	<code><xs:element name="roleId" type="xs:string"/></code>
	The id of the role.

element **process/roles/role/roleDescription**

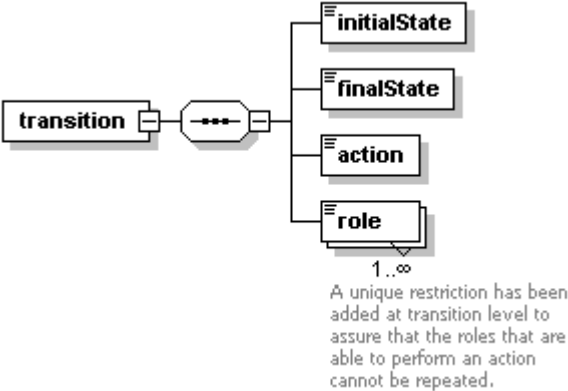
diagram	
type	xs:string
source	<code><xs:element name="roleDescription" type="xs:string"/></code>
	The description of the role.

element **process/transitions**

diagram	
children	transition
source	<pre> <xs:element name="transitions"> <xs:complexType> <xs:sequence> <xs:element name="transition" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="initialState" type="xs:string"/> <xs:element name="finalState" type="xs:string"/> <xs:element name="action" type="xs:string"/> <xs:element name="role" type="xs:string" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>A unique restriction has been added at transition level to assure that the roles that are able to perform an action cannot be repeated.</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </xs:sequence> </xs:complexType> </pre>


	<pre> <xs:unique name="nonRepeatingRoles"> <xs:selector xpath="/role"/> <xs:field xpath="."/> </xs:unique> </xs:element> </xs:sequence> </xs:complexType> </xs:element> </pre>
	<p>This element groups all the transitions allowed in the process.</p> <p>The transitions define the logic of the process, because they establish the state's changes that can be made, and who is able to make these changes.</p> <p>A constraint not allowing duplicated transitions with the same initial state and action is defined in the root element (process).</p>

element process/transitions/transition


<p>diagram</p>									
<p>children</p>	<p>initialState finalState action role</p>								
<p>identity constraints</p>	<table border="1"> <thead> <tr> <th>Name</th> <th>Refer</th> <th>Selector</th> <th>Field(s)</th> </tr> </thead> <tbody> <tr> <td>unique</td> <td>nonRepeatingRoles</td> <td>/role</td> <td>.</td> </tr> </tbody> </table>	Name	Refer	Selector	Field(s)	unique	nonRepeatingRoles	/role	.
Name	Refer	Selector	Field(s)						
unique	nonRepeatingRoles	/role	.						
<p>source</p>	<pre> <xs:element name="transition" maxOccurs="unbounded"> <xs:complexType> <xs:sequence> <xs:element name="initialState" type="xs:string"/> <xs:element name="finalState" type="xs:string"/> <xs:element name="action" type="xs:string"/> <xs:element name="role" type="xs:string" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>A unique restriction has been added at transition level to assure that the roles that are able to perform an action cannot be repeated.</xs:documentation> </xs:annotation> </xs:element> </xs:sequence> </xs:complexType> </xs:unique name="nonRepeatingRoles"> </pre>								

	<pre><xs:selector xpath="/role"/> <xs:field xpath="."/> </xs:unique> </xs:element></pre>
	<p>One of the different transitions that are defined in the process. The initial state, the final state and the action that fires the state's change is defined here, and also the roles that can perform this action.</p> <p>A constraint is defined to avoid that duplicated roles in the roles list.</p>


element `process/transitions/transition/initialState`

diagram	
type	<code>xs:string</code>
source	<code><xs:element name="initialState" type="xs:string"/></code>
	<p>This element establishes the required current status so that the transition is applicable.</p> <p>This status has to be one of the defined earlier in the file, in the states element.</p>

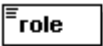
element `process/transitions/transition/finalState`

diagram	
type	<code>xs:string</code>
source	<code><xs:element name="finalState" type="xs:string"/></code>
	<p>If the transition is fired (the action is performed) this is the resulting status, the status to which the process changes.</p> <p>This status has to be one of the defined earlier in the file, in the states element.</p>

element `process/transitions/transition/action`

diagram	
type	<code>xs:string</code>
source	<code><xs:element name="action" type="xs:string"/></code>
	This element establishes the firing action that provokes the state change, from initial to final state.

element **process/transitions/transition/role**

diagram	 <p>A unique restriction has been added at transition level to assure that the roles that are able to perform an action cannot be repeated.</p>
type	xs:string
annotation	documentation A unique restriction has been added at transition level to assure that the roles that are able to perform an action cannot be repeated.
source	<pre><xs:element name="role" type="xs:string" maxOccurs="unbounded"> <xs:annotation> <xs:documentation>A unique restriction has been added at transition level to assure that the roles that are able to perform an action cannot be repeated.</xs:documentation> </xs:annotation> </xs:element></pre>
	<p>This element specifies the role or roles that are allowed to perform the action and, so, fire the transition.</p> <p>As it was said before, a constraint is defined to avoid duplicated roles.</p>

2.8.13 RDBMS Data Model

2.8.13.1 Users, Roles, and Work Groups

All data related with users, roles and work groups will be stored in a relational database. In Figure 62, we have to distinguish two structures:

- **Roles and Groups system:** It stores all the information related with the roles and groups existing in the SmartGov System, and the linking with the users. This system must be present in all SmartGov service design environments, and may be used in some delivery environments too. This system is composed by all the tables except Users and OuterUsers.
- **Smartgov Users system:** A very simple system to store all the required data of the users (OuterUsers table). This system will be used during Smartgov Platform development, to avoid integrating the platform with a more complex user system (for instance LDAP).

The bridge between these independent system is the table Users. In this table the link between a SmartGov user and the corresponding user in the outer system is established.

All the users imported from certain outer user system will be grouped in the same group, defined in the table Groups. This group will be related with a *OuterUserSystem* class, that will make possible the connection with the outer system. Thus, the whole model of roles and groups can be linked with different user's system just developing a simple interface. Even, the system may work with more than one outer user system, each one with its *OuterUserSystem* class, and each one with its own group.

To distinguish the groups related to user system from the work groups used in the front-end application, the field *type* in the table Groups will be used. Just two group types has been defined, but in the future more different types may be required, so the system is prepared for adding them:

- *WorkGroup* type: Groups users that are going to develop a new service.
- *Grouping* type: Groups all users from a *OuterUserSystem* included in the systems. In the SmartGov design environment users come from a defined SmartGov Users table. For instance in the future a different user's source could be a LDAP system. All those users coming from the same source will be under the same *Grouping* type.

Other issue that requires a deeper explanation is the scope for the roles (*RoleScopes*). This scope is used to group roles in the database system. There are three different kind of scopes defined:

- Scope for the SmartGov roles (*SmartGov* scope), used to control access to the SmartGov design environment (Domain Experts, IT Staff...). Each SmartGov user pertaining to a group of *Grouping* type, will have one (and just one) SmartGov role.
- Scope for e-services (scope defined by the user): roles to be used in a service execution when needed (names of the roles specific for each service). Every user belonging to a group of *Grouping* type, may have 0 to n service roles.
- Scope for Work Groups (*TS* or *KU* scope): Those are the roles in the processes related with TS or KU life-cycle (Reviewer, Approver...). Every

user pertaining to a group of *WorkGroup* type may have 0 to n TS or KU life-cycle roles.

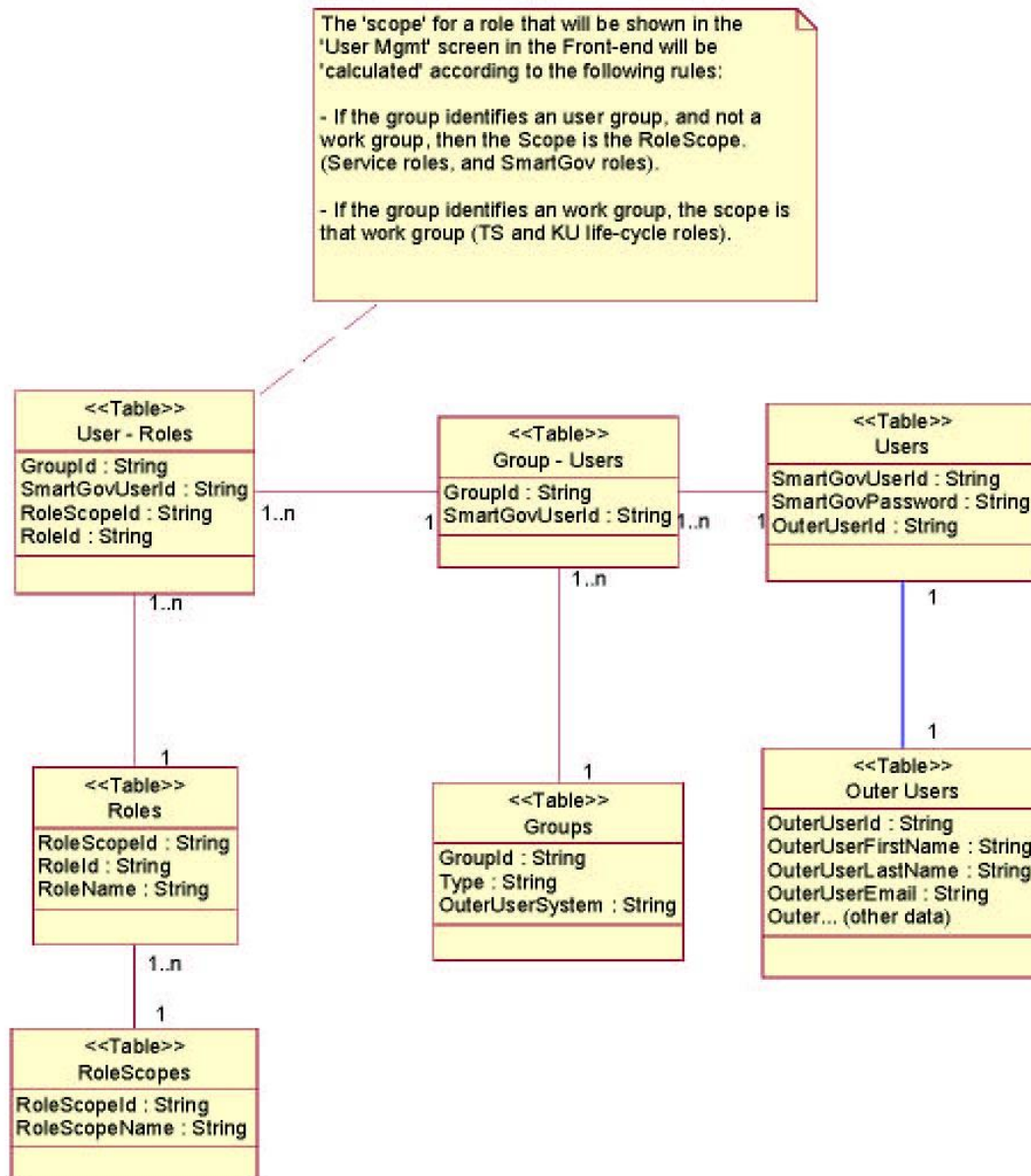


Figure 62 – SmartGov Roles, Groups and Users database schema.

2.8.13.2 Service Design Environment Statistics

In the design environment some statistics related with knowledge will be collected. Theses statistics will allow the managers to update and improve the knowledge pieces in the system, according to users comments, customs...

To store all these statistics will be used a system different from the delivery environment one. Instead of defined specific statistics to be collected for each object, some general statistics, common for all the elements of the same type, will be defined, and the system will store this data in a database.

Instead of using a specific schema for the defined statistics, a open schema has been defined. This way, new statistics types can be defined without modifying the system. New object types to define statistics over them can be added to the system (ObjectTypes Table), and also new types of statistics to be collected (Statistics Types Table). Therefore, these two tables will keep the static definition of the statistics, while the Statistics table will keep the dynamic information, that is, the collected data.

To store the data in a format suitable for each type, different columns are defined, for textual, numeric of percentage content, and in the statistics type definition will be defined which columns are applicable in every case.

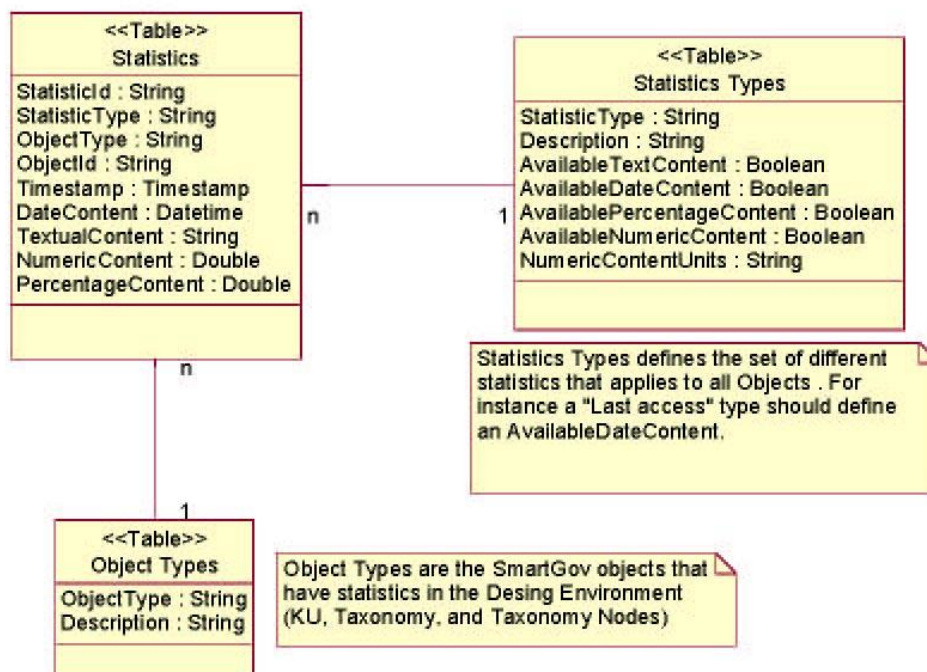


Figure 63 – Service design environment statistics database schema.

3 Conclusions

This deliverable presented the results of the first iteration of the analysis and design phase. As a result the low level specifications of the SmartGov platform were produced, elaborated and presented here. The design of the platform followed a component-based approach, which culminated in a number of self-contained components thus ensuring a clear separation of concerns and reusability. The interfaces between these components have been clearly set and described here.

This deliverable marks the point in the development phase where the final implementation path has been set, all roles and tasks are assigned and every implementation doubt is cleared. However the next iteration of the platform's development phase might affect the design and analysis presented here, but not to a significant degree.

4 References

- [Pressman2000] Roger S. Pressman, "Software Engineering", McGraw Hill, 2000.
- [Quatrani2000] Terry Quatrani, "Visual Modeling with Rational Rose 2000 and UML", Addison-Wesley, 2000.
- [EC2000] European Commission, 'Public Sector Information: A Key Resource for Europe', Green paper on Public Sector Information in the Information Society, <ftp.echo.lu/pub/info2000/publicsector/gppublicen.doc>
- [eEurope2000] eEurope, 2000. Common list of basic public services, http://europa.eu.int/information_society/eeurope/action_plan/pdf/basicpublicservices.pdf
- [Bolton2001] Fintan Bolton "Pure CORBA", SAMS Publications, 2001, ISBN: 0672318121
- [Sun2000] Sun Microsystems, "ONC+ Developer's Guide", available at <http://docs.sun.com/db?q=RPC&p=/doc/805-7224>
- [Scarborough1999] Scarborough H. "Knowledge Management: A Literature Review", Institute of Personnel and Development. 1999
- [Siebel2001] Thomas M. Siebel, "Taking Care of eBusiness", Doubleday Publications, 2001, ISBN: 0385502273
- [Stacey1996] Ralph D. Stacey, "Complexity and Creativity in Organizations", Berrett-Koehler, 1996, ISBN: 1583763007
- [Senge1999] Peter M. Senge et al., "The Dance of Change: The Challenges to Sustaining Momentum in Learning Organizations", Doubleday Publications, 1999, ISBN: 0385493223
- [Weick2000] Karl E. Weick, "Making sense of the organization", Blackwell Publishers, 2000, ISBN: 0631223193
- [Bowman2002] Brent Bowman, "Building knowledge management systems" July 2, 2002
- [Davenport1977] Thomas H. Davenport, Laurence Prusak and Lawrence Prusak, "Working Knowledge: How Organizations Manage What They Know", Harvard Business School Press, 1997, ISBN: 0875846556
- [Struts] The Jakarta project, The Struts Framework, available at <http://jakarta.apache.org/struts/index.html>
- [Ant] Apache foundation, The Apache Ant Project, available at <http://ant.apache.org/>
- [Xalan] The Apache XML Project, Xalan, available at <http://xml.apache.org/xalan-j/>
- [JavaCC] WebGain Company, JavaCC, available at http://www.Webgain.COM/Products/Java_CC
- [Castor] The Exolab Group, The Castor Project, available at <http://castor.exolab.org/index.html>
- [RUP] The Rational Unified Process, IBM Software Group, available at <http://www.rational.com/products/rup/index.jsp>
- [XML] World Wide Web Consortium, The XML Specification, available at <http://www.w3.org/xml>
- [J2EE] SUN Microsystems, The J2EE platform, available at <http://java.sun.com>