A repository-based framework for evolutionary software development

Martine Devos, Director Information Systems Departement (mdevos@argo.be)
Michel Tilman, Senior System Architect (mtilman@argo.be)

Abstract

This paper describes a framework developed to support the Argo administration Argo. Argo is a semi-government organization managing several hundred public schools. It uses this framework to develop its applications, which share a common business model, and require database, electronic document, workflow and Internet functionality.

The framework is based on a repository in two ways. First, it consists of a set of tools for managing a repository of documents, data and processes, including their history and status. These tools enable users to select applications, enter and view data, query the repository, access the thesaurus and manage electronic documents, workflow processes, and task assignments. More importantly, the framework behavior is driven by the repository. The repository captures formal and informal knowledge of the business model. None of the business model is hard coded. The tools consult the repository at runtime. End-users can change the behavior of the system by using high-level tools to change the business model. Thus we separate descriptions of an organization's business logic from the application functionality.

The framework helps us develop applications iterative and interactively. It enables us to build increasingly complete specifications of applications that can be executed at once. The resulting system is easy to adapt to changes in the business. Argo can develop new applications through modeling and configuration, rather than through coding.

1. Context

Argo is a semi-government organization that manages the public schools (non-denominational) within the Flemish community in Belgium. It consists of a central administration and a few hundred semi-autonomous local sites (boards and schools).

The project started early 1994. At the start, Argo was going through an external audit, which made it likely that there would be major changes of procedures, organization structure, accountability rules and delegations of responsibility. Future requests for changes were a given at the start of the project.

So Argo needed a framework to build applications, sufficiently flexible to cope with changing and emerging needs and opportunities, both functional and technological, imposed or desired. It wanted an environment that not only would survive the expected reorganization and decentralization but that would stimulate users to envision better ways of working.

We started with three pilot applications to provide input for initial framework requirements and design:

A documentation center that provided flexible search and retrieval of data and documents such as legislative texts and parliamentary decrees concerning education. Many documents have a complex structure of interrelationships like appendixes, references to subsequent changes and juridical decrees based on multiple laws.

A system to support the central board's decision procedures from initial draft version to the final text, and then to monitor the decisions' implementation by the administration or the local boards and schools. These decisions include motions that have been passed or tabled, contracts with teachers and changes to curricula and budget reports. The path through the administration is not clear-cut and it changes depending on parties involved or on the subject of the decision. The procedures are not always limited to a strict hierarchical structure; some involve temporary workgroups or external committees.

An application to import, index, store and route large volumes of documents sent in by the local boards, such as attendance records, decisions, meeting notes and large addenda.

Halfway through the project the pending re-organization was effectively carried out. Although the nature and extent of it were largely unknown at the onset, we were able to adapt the existing applications through re-configuration without additional coding.

Since then, several new end-user applications have been delivered. We are now in the process of extending the framework; most notably to give schools and local boards access to the central data and information through the Internet. We are converting large existing relational database applications and we added web functionality and an Internet discussion system.

These applications, together with new technological possibilities, are our major sources of requirements for the evolving framework. Within this process we further refined and re-designed many framework components. The persistency component, for instance, has been completely re-implemented. In addition, we created some applications to support the development process, such as bug reporting system with follow-up, training-session management and framework documentation management.

Not only do we build end-user applications with data, document, workflow and Internet functionality. We also use the same framework to develop the tools that we use to build and manage these applications. In this way users can use their interface, known from their business objects, to adapt applications.

End-users at Argo are gradually taking over management of applications and configuration from the development team. Key-users teach and write parts of the documentation. User groups organize workshops to discuss how to put the technology to better use. This process was initially driven by small teams of highly motivated pioneer-users, and has gradually started to embrace the end-user community at large. Given the high degree of computer-illiteracy within Argo at the start of the project, we feel this to be a significant achievement on behalf of the end-users.

Project requirements

1.1. Functional

The technology has to support applications containing a mixture of database e.g. school database, patrimonies, personnel, budget, inventory; electronic documents e.g. educational documentation, research papers, articles, laws and decrees; workflow management e.g. task assignments and follow-up, decision routing, process management ; and has to give local sites access to the central repository of data and documents.

Database

Users have to be able to enter, modify, remove, query, list, browse, report, print, import and export data. They need tools to set up appropriate views, and to store queries for later re-use.

Electronic document management

Users need to create, index, search, edit, annotate, print, export and process documents. These can be either electronic (such as files or E-mail) or scanned paper documents. Structured information, thesaurus keywords and full-text-indexing are used for indexing and searching. Optical character recognition enables users to extract text from scanned documents. Users need tools to manage document versions. They must be able to choose the most appropriate alternative representations of the same document, depending on environment or job at hand. A document created and adapted with a word processor must be frozen once it has passed the decision process and can be presented in PDF format to schools.

Workflow management

Users have to be able to plan and handle incoming tasks, to keep track of outgoing task assignments and to manage workflow processes, which can be more or less well defined or completely ad-hoc. Argo wants workflow functionality to suggest opportunities to its users, guiding rather than restricting them.

Remote access

Argo needs remote access to its applications, data, documents and processes in the central repository in several ways:

- using an off-line version of the system through import / export of documents and data for employees occasionally taking work home for specific tasks like preparing a proposal for the central board;
- using the system on-line by means of a direct dial-up connection, e.g. modem or ISDN for board members and employees working remote;
- using E-mail and Web browsers to access the repository through the Internet for schools and teachers.

1.2. Integrated environment

The applications must be integrated in one environment. All applications must use a common business model. This model provides the cohesion expected in a large administration, providing formal procedures and policies (e.g. hierarchy, responsibilities, indexing vocabulary of documentation, and global validation- and authorization rules), but giving ample scope and freedom to adapt the technology to the most appropriate local or personal working practices. For some legal activities all departments have to abide by strict rules requiring a prescribed number of signatures, while less formal processes are in use for more everyday tasks.

1.3. Configuration and administration tools

There must be tools to define the object model, constraints and behavior, to configure and manage applications, views and stored queries, to define and manage workflow processes, to set up access control, to manage the database, and to configure caching and back-up of documents.

At the end of the project, Argo's users need to be able to apply these tools. Hence the deliveries do not only include end-user applications, but also the necessary tools to empower users to model and configure applications.

1.4. Support for change

To cater for changing needs, the system has to forego hard coding as much as possible: the technology has to support iterative and incremental development through the use of prototypes. Even more, it must act as a catalyst for a learning process when introducing this new technology, thus effectively complementing a Business Process Re-engineering project at Argo.

2. Our framework approach

We aim to combine the high-level modeling power of CASE-tools with the open-ended nature of object-oriented frameworks. Our approach is based on the following observations:

Users tend to describe their requirements in terms of how they currently work. It is difficult for them to picture how new technology can or will influence their future working practices. Hence they need to try out, to learn from and to correct the software delivered. This requires extensive prototyping. Too often, however, prototypes are used to validate the design, rather than help users and developers capture and explore the real requirements. Prototypes can be expensive, as they usually have to be thrown away. This is particularly true when developing many applications, each one requiring several iterations to get the requirements right.

A business model is particular to each organization at a particular point in time [Sto94]. However, functional requirements, such as data-entry, querying, reporting and document management, are generic.

Extensions or enhancements to functionality can often be made into re-usable assets, independent of the actual business model, and vice versa.

Architecture

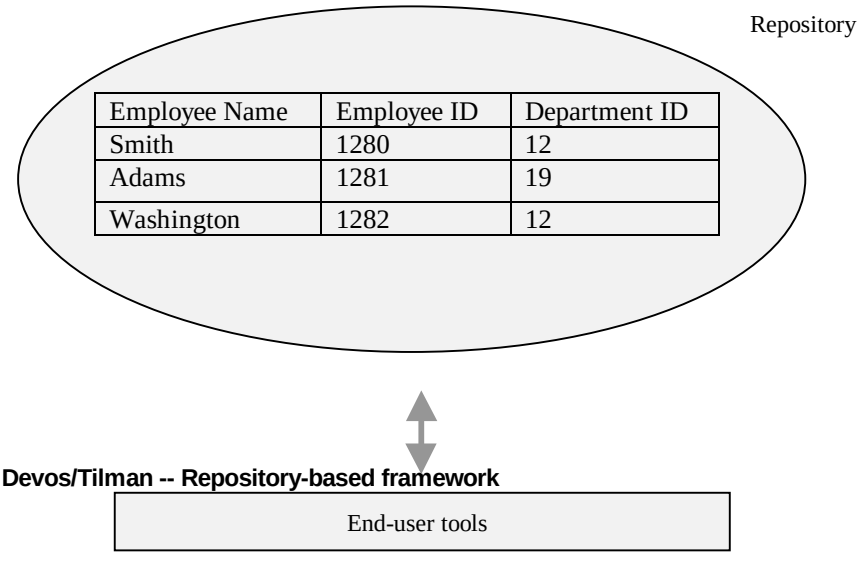
The core of the architecture is a three-level repository that is consulted dynamically by two sets of tools.

2.1. Business objects repository and end-user tools

The framework uses the first repository to capture knowledge of a particular business model, including organization structure, roles, data, documents and processes.

We provide the user with tools to access this repository. These tools enable him to select applications, to enter and view data, to query the repository, to display, print and export the results of a query, to access the thesaurus, to manage electronic documents, workflow processes and task assignments.

This way we deliver the functionality demanded by the end-user: applications for database, document management and workflow and access to all these over the Internet.

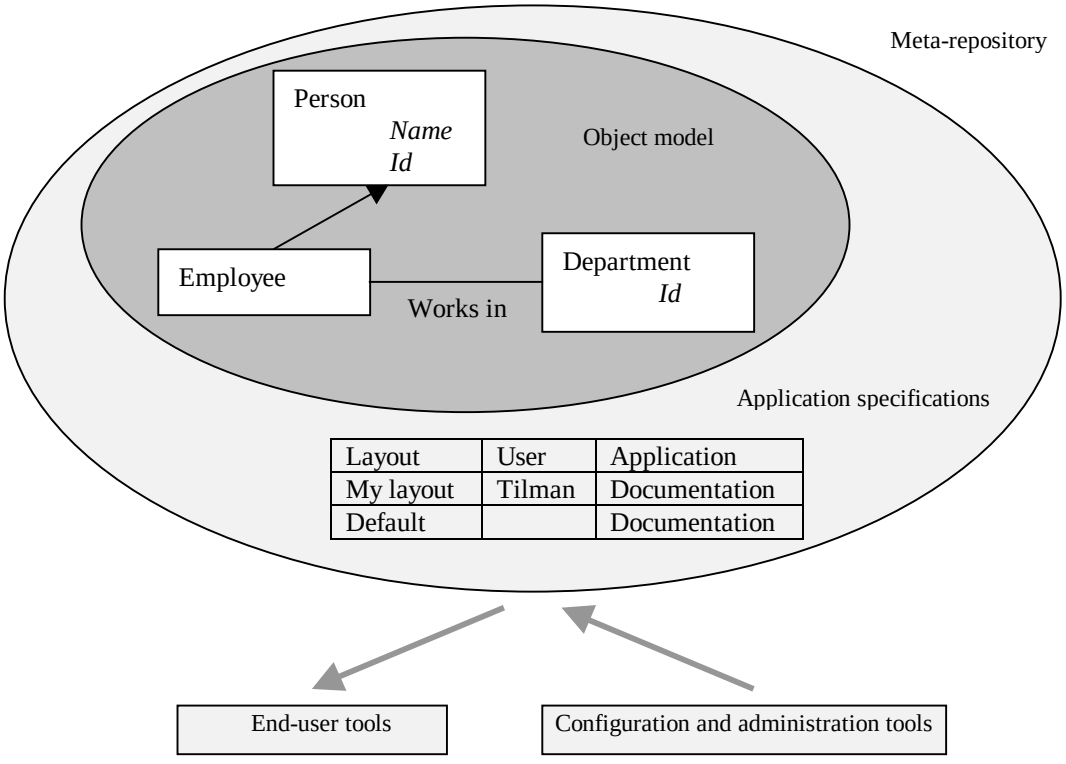


2.2. Configuration tools and meta-information

We want the end-user to be able to tailor, configure and manage these applications. Therefore we provide a second set of high-level tools**.** They define private and shared views on data, store queries for later re-use, edit the object model, set up processes, define authorization and business rules, and add or remove end-user functionality and even applications.

We don't generate the applications. Instead we use a second repository, storing information necessary to specify end-user applications. This meta-repository contains meta-information: information about the structure of applications and business model, user views (lay-outs,...).

Example for usage: We use the meta-level-information to dynamically generate query-screens and forms. This process is based on the object-model. Because we store the meta-information about the objects in the repository, the screens will adapt to changes in the object-model, such as adding properties or associations.



Both sets of tools consult this meta-repository at run time. Changes made to the meta-information e.g. object structure, applications definitions, are immediately available to clients.

All tools are fully operational and provide out of the box both end-user functionality together with the means to configure and fine-tune. They only need the meta-information to get the system up and running.

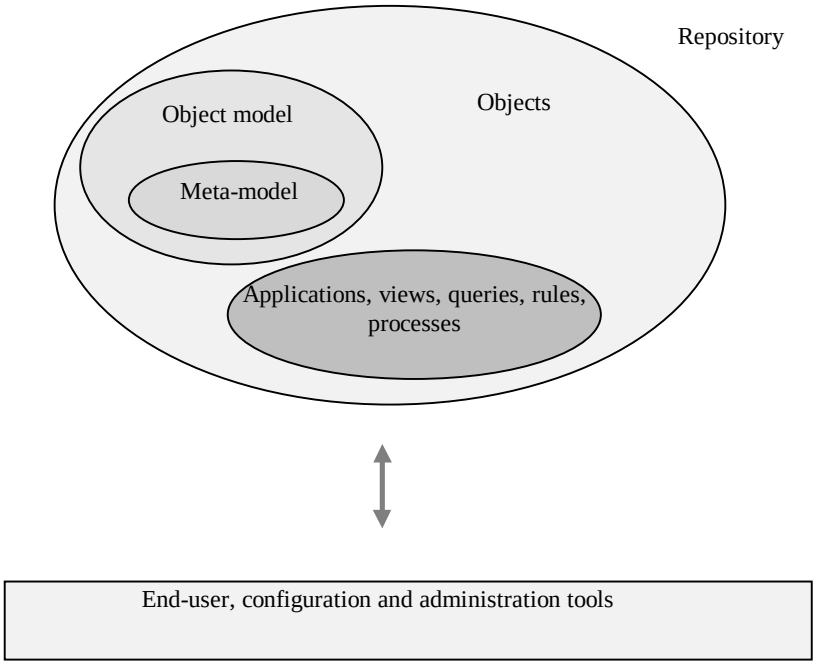
2.3. Secondary goals

To make these tools easily accessible to users, we give them the same interface and functionality as the regular end-user applications. Ideal would be to have one single set of tools, thus enhancing reuse and ease of maintenance.

Support of a bootstrapping process became an important secondary design goal: administration and configuration tools are gradually expressed in terms of the framework itself. We replace most hard-wired administration and configuration tools with configured applications. This leads to a small but powerful kernel of generic and orthogonal tools.

The meta-model used to describe the object model, is in turn expressed in terms of itself and is stored in the repository too. This is a third-level repository, a meta-meta-repository. This started as an experiment to push the limit of flexibility, but proved to be useful. It enables us to extend the semantics of an object model.

Example of usage: In the first version of the framework the semantics of cascaded deletes were hard-wired. Now we define and store rules in the repository that overwrite build-in default behavior. More intelligent delete-rules can now be adapted to cope for particular needs in an application.



In practice the three repositories are one and the same, managed with the same tools.

3. Using the framework tools

We give a short overview of a documentation center application, a decision procedure application and an incoming mail registration application. We will illustrate how the user applies the various tools to build and configure applications.

3.1. Examples

3.1.1. Documentation center

With the documentation center application, users manage authors, publishers, and several types of books, periodicals, editions and other kinds of documentation, including the framework documentation. We offer two applications. Librarians use the first one for administration purposes. Regular end-users use a simpler application.

Some documentation exists only on paper. In this case we maintain references to the physical location. Most documentation is made accessible on-line, in original format or as scanned documents. Part of the documentation is made available on the Internet, in an alternative representation, such as PDF.

For indexing and querying we rely extensively on thesaurus and full-text-indexing. Librarians manage distribution lists to notify users automatically about new documentation. They also keep track of loans. In addition to the standard printing facilities, librarians print documentation in ISBD-specific format.

The object model is complex. To reduce complexity and make the application more dynamic, we use some explicitly typed objects and associations, e.g. typed “replaces” and “is an addendum to” associations between documentation. This is easy to do, since our tools query both meta- and object-level information, which gives us considerable flexibility to design or redesign the model.

3.1.2. Workflow processes

Argo does not use workflow as a means to single-mindedly automate existing processes. Too many exceptions occur in the office to prescribe the flow in most cases. Although some processes follow a strict plan, in general, users need the freedom to deviate from this plan. And in some cases there is no fixed plan.

The central board decision procedure formally describes the process for submitting a dossier for approval by the central board, starting from a department’s manager. The preparation phase prior to this process depends on each department’s culture, which in some cases is very team-oriented and in other cases is inspired by a hierarchical structure. The general manager has the right to advise on the dossier, but he is not part of the actual process flow. To keep him informed of upcoming dossiers, notifications are sent when the process enters a particular phase. When decisions result in a concrete action plan, management needs to check when and if the goals are actually met, by monitoring the state of specific data, documents and processes.

Use of ad-hoc processes is illustrated by the application for handling incoming (paper) mail. Users register these documents in the system and send appropriate task assignments to handle the dossiers. The actual process flow depends on the contents of the documents.

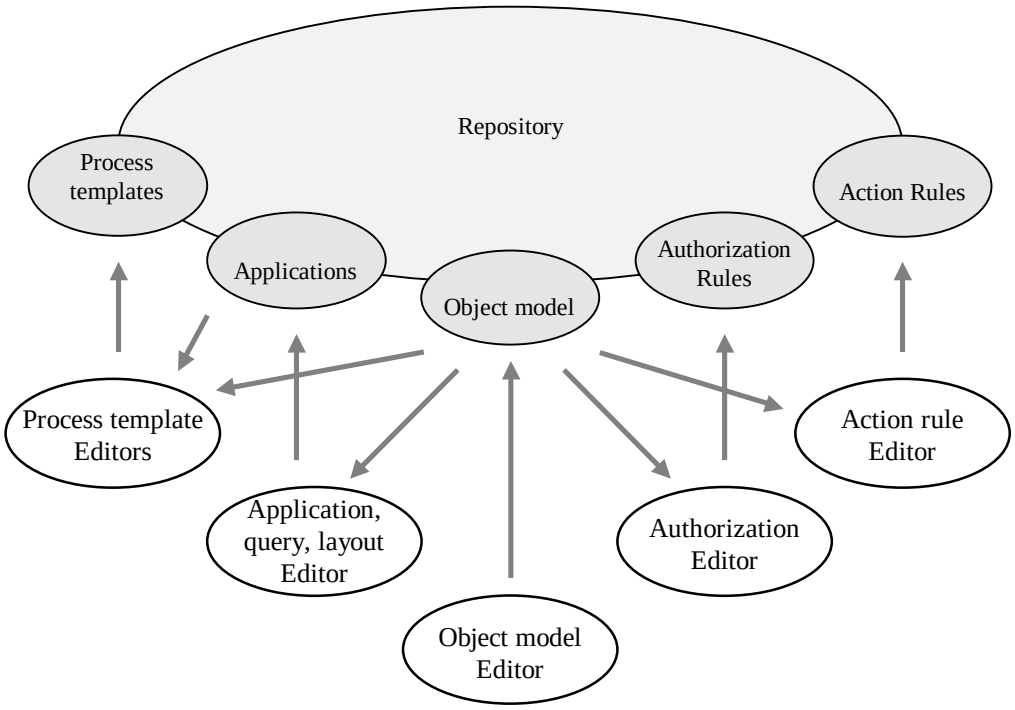
3.2. Building applications

Starting from the model in the repository at a certain moment, we build applications by going through a set of steps. These steps can be performed in any order, even interactively. The development process unfolds incrementally. We can try out and correct ideas dynamically, together with our users.

Starting with a model, which initially contains only meta-model and system objects, developing an end-user application implies that we:

- extend or refine the object model;
- set up application environments;
- extend or refine authorization rules;
- define action rules
- and define workflow process templates.

This process is illustrated in the following picture. Incoming arrows denote dependency for the tools; outgoing arrows specify the products



3.2.1.Extending or refining the object model

We update the model with the necessary object types (classes), attributes, association types (relationships) and global constraints, when new structures or modifications are required. If needed, we add behavior. The object model is shared by all applications and so provides cohesion. We store this object model in the repository to make it dynamically available for the tools. This way, we can set up arbitrary business models without re-coding tools or applications.

With the object model editor we populate the meta-model with the object model, associations and basic constraints, such as uniqueness of value and totality of relationships.

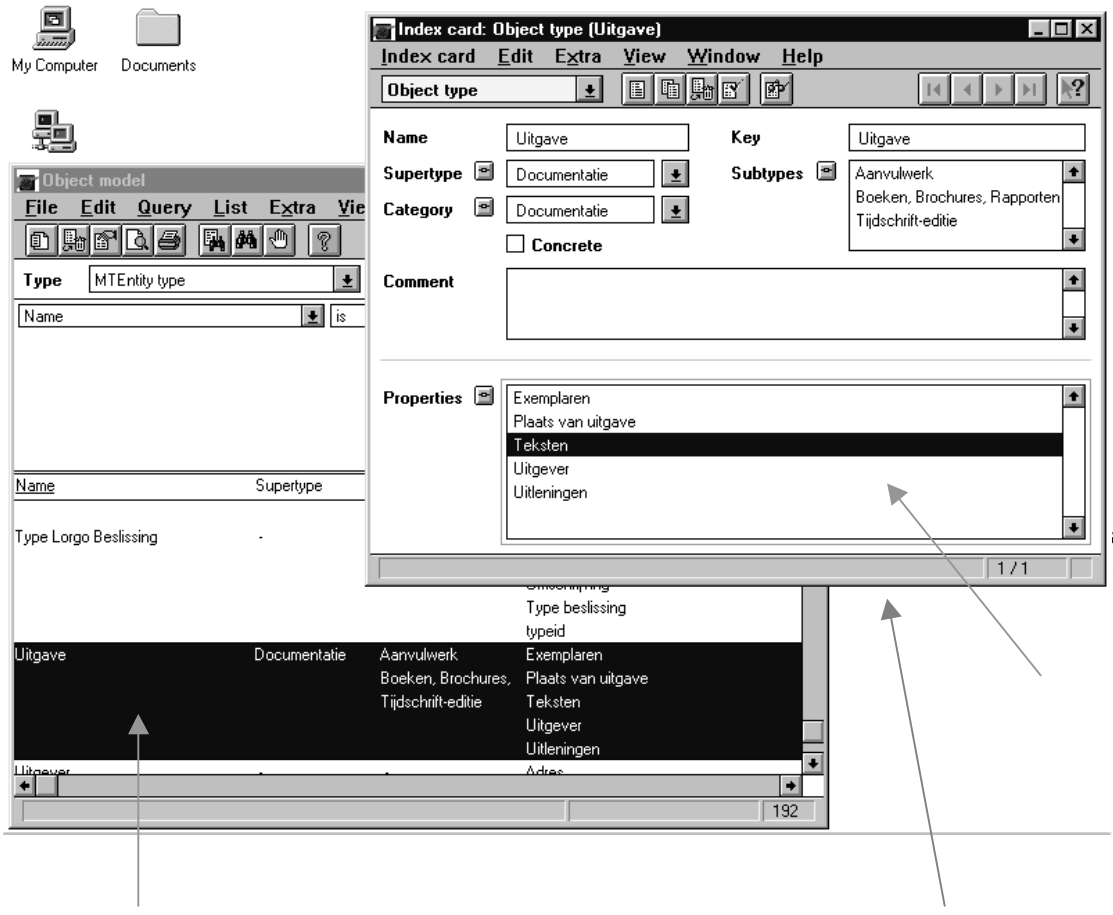
Object behavior and user-defined constraints may be added now or later on, e.g. to support cascaded deletes or to ensure correct initialization. Object behavior and constraints are common to all applications.

The object model editor defines the super-types and subtypes of a new object type. It also defines the properties of a type, which can be either attributes or associations.

In order to bootstrap we first developed a hard-wired editor, which we have now discarded.

The object model can be exported to a UML CASE tool and visualized graphically in the tool. We will use UML-diagrams to edit and design the elements of the model in the near future as well.

When we have defined the model, we generate the necessary database structures. We select the mapping strategies for representing objects in the database. These strategies may be mixed, and even changed later on, to optimize the database mapping and to ease integration of existing databases.



are either attributes o
s.

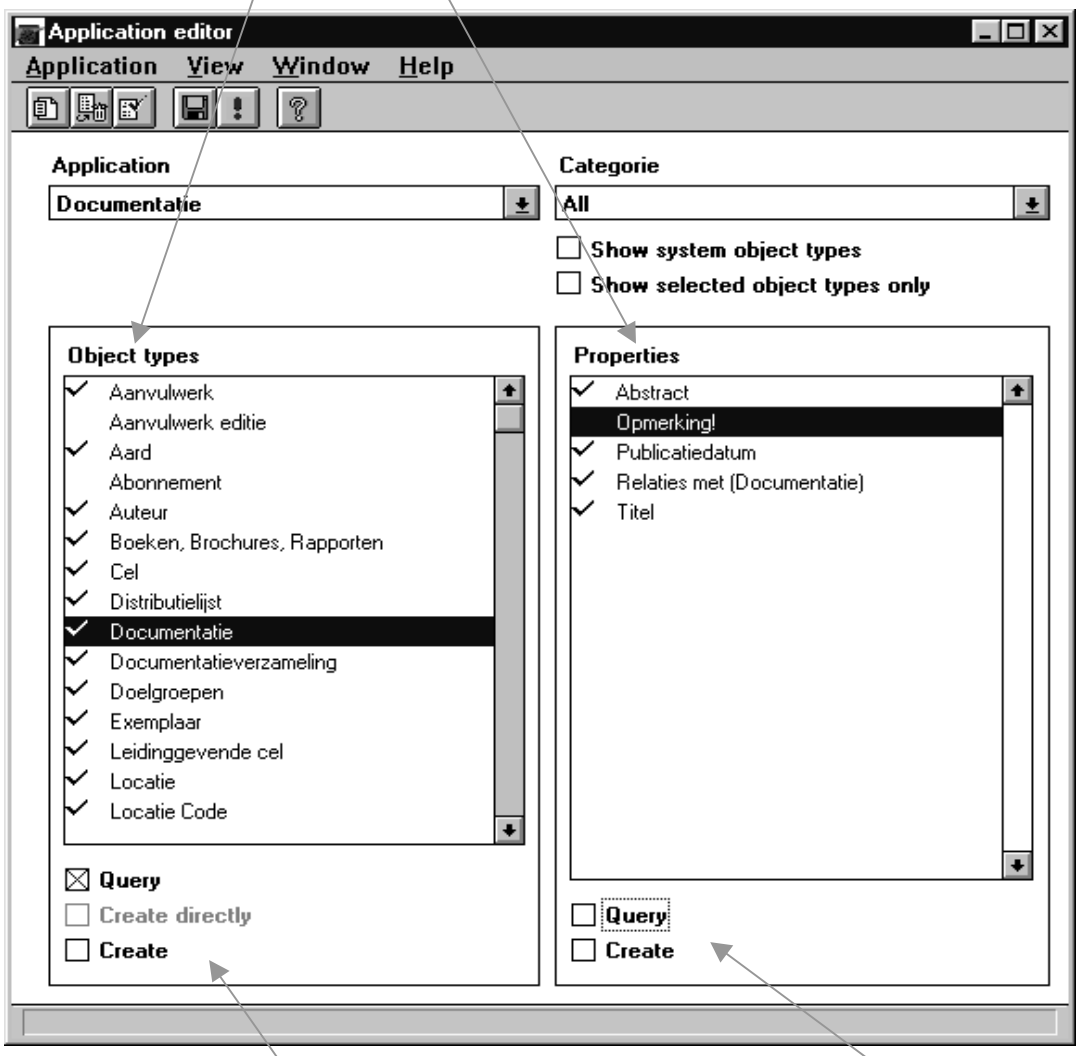
From the list of object types we have opened a form containing detailed information about the selected object type. Note that we are using the same end-user tools for managing the object model, object behavior and constraints as we do for the regular applications.

3.2.2. Setting up the application environment

This step starts with defining a view on the shared business model: objects and properties to be accessed, created and queried within the context of this application. Once this basic environment

has been stored in the repository and access privileges have been set, the user gets a default application that is available for immediate use. The user can log on, choose this environment, enter data, query the repository, list or print results. He can keep track of task assignments, manage electronic documents and access the thesaurus. Functionality can be added or removed. The environment can be further refined for individual or shared use through views, queries and sub-environments.

Object types and properties with a check mark are visible in the application.



For each visible object type we specify whether the user can query the object type, create instances as such (from within the main window) or from within the context of another object (i.e. from within a form).

For each visible property we specify whether the prtoperty can be used in the query editor. This is particularly important for associations, as association properties can be used to build complex queries, exposing the object model to the user

In the example above we define the application “Documentatie”. We can define global filters on the object model, defining a category. We select object types to be used in the application environment. For each object type we indicate accessible properties and how they are used and created.

Having defined the application environment, the user has access to a fully functional default application by logging on in the main application window.

The main window (here on the default mail registration application) presents the user with the query editor and an overview list. It also provides access to other functionality, such as additional query and list windows, forms, document management, the thesaurus browser, workflow processes, in- / out-baskets, preference settings and on-line help.

Queries can be saved for individual or shared re-use within the application. Each query can have its own list lay-out. This list lay-out is selected automatically when the query is executed.

Since associations can be used to browse through the object model, end-users can formulate sophisticated queries themselves. Novice users get simpler applications where most of the object model structure (the chains of associations) is hidden or replaced by logical (virtual)

The top pane is the query editor. It is searching for objects of type IndicateurDocument, but not for its subtypes. Queries can be nested (using and-or-not combinations), and may involve

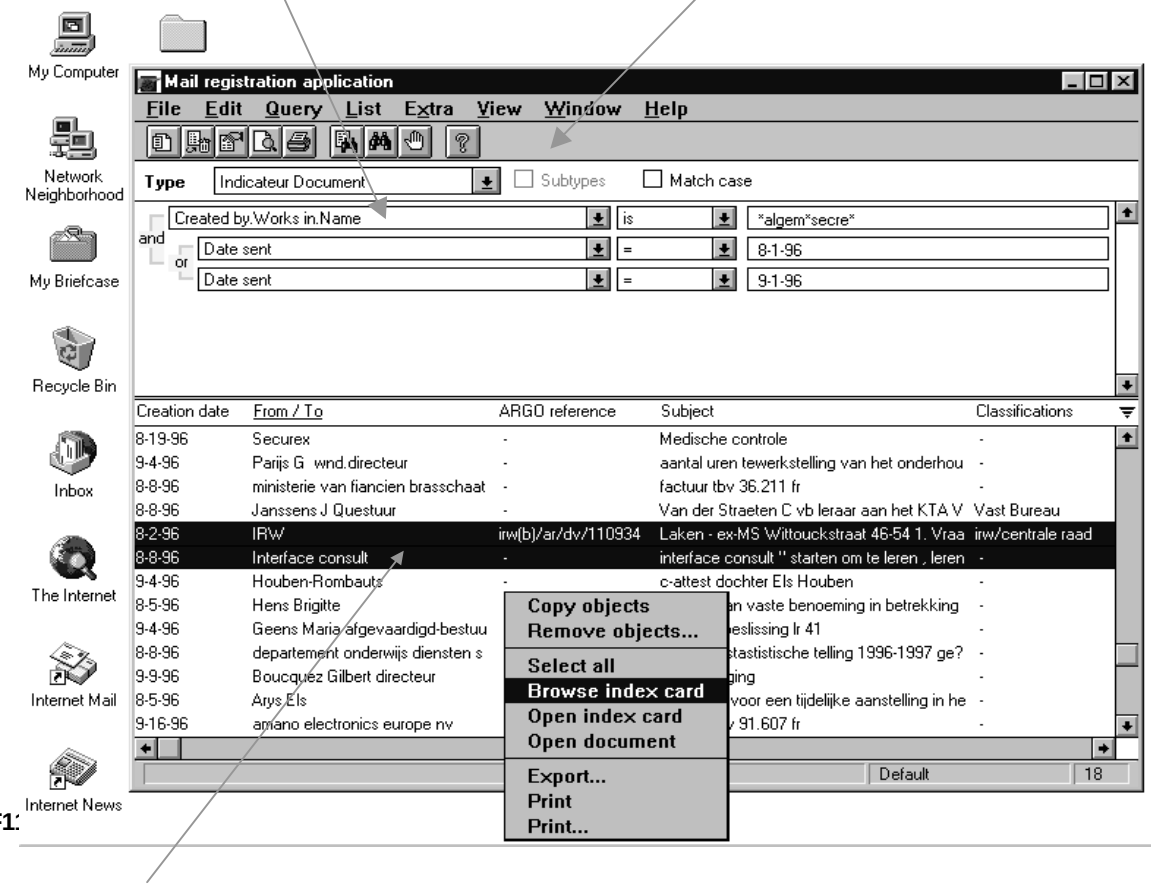
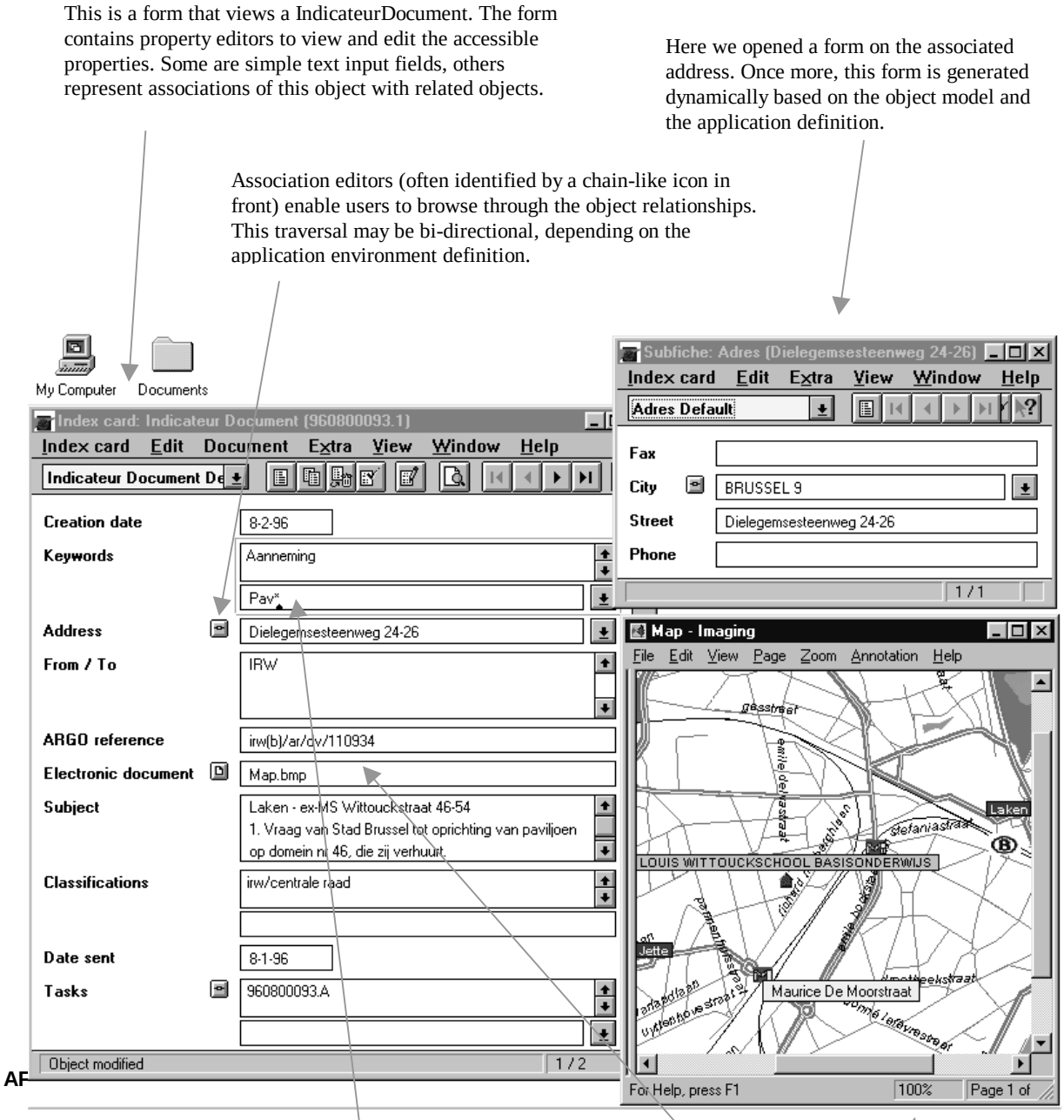


Figure 7 Query editor in an application

Users create, view, edit, print, export and browse objects through the form tools. Each form is generated on the fly, according to a default strategy, subject to the authorization rules. The form generator selects appropriate editors for the individual properties and the object type at hand. In this figure we opened a form on the two elements selected in the overview list.

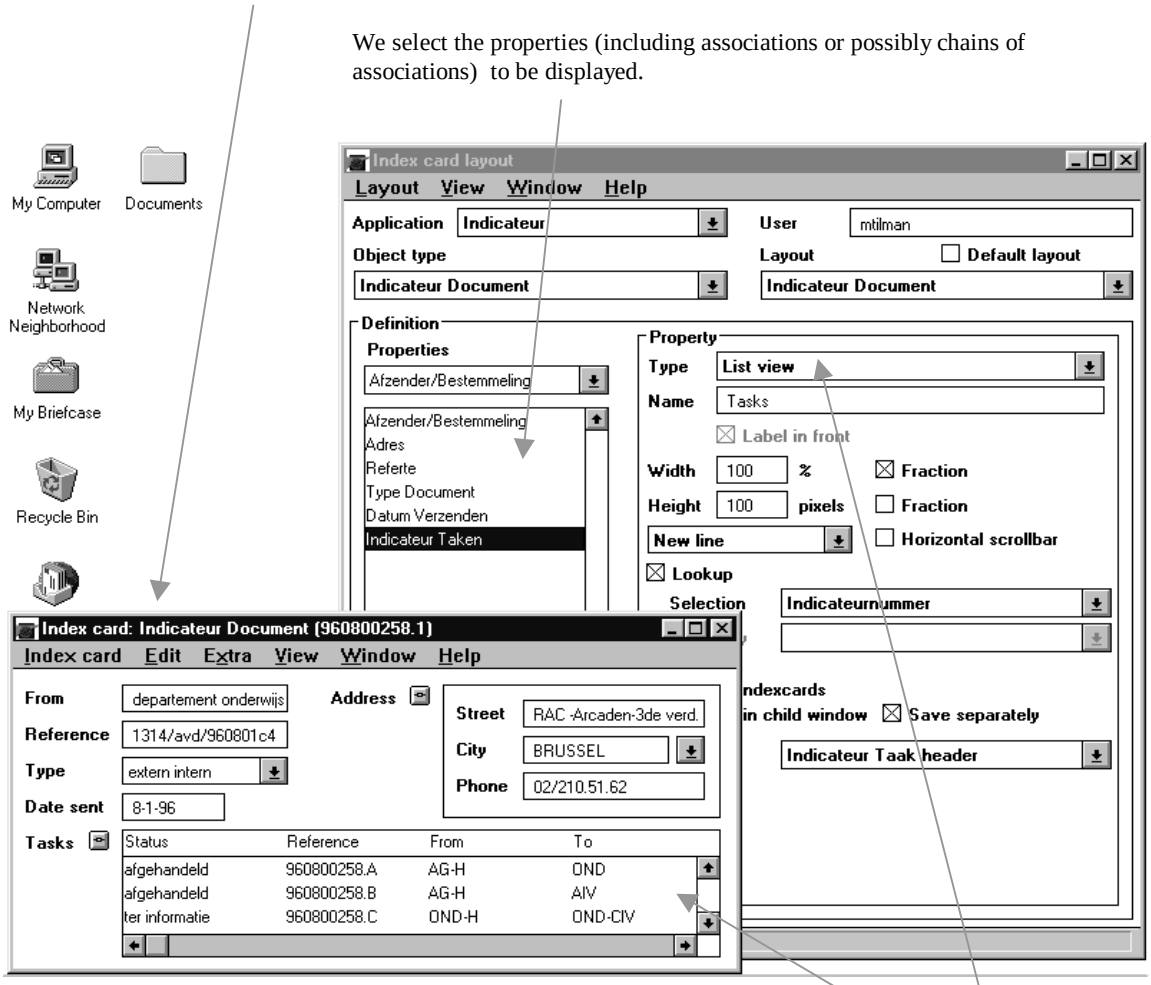


Lay-out editors enable users to define new forms and overview lists and change old ones. The form lay-out editor selects the alternative editor for each property. For instance, association editors may be replaced with embedded list views and forms, enabling re-use of existing lay-outs. In the list lay-out editor, hierarchical relationships can be used to define threaded list views. Furthermore, association chains can be followed when defining the list of properties to be displayed, allowing us to set up de-normalized views. Lay-outs can be the default lay-out for a particular object type within a given application. In that case they will be selected automatically in overview lists and forms for the corresponding object type.

Since lay-outs contain some complex information that needs not be modeled explicitly (such as property information and editor type), the major part of their definitions are stored in an encoded form. Only the elements to query and manage lay-outs are modeled explicitly, such as name, default flag, object type, user (if private) and application (if any). This is sufficient to configure a simple application for listing, reporting and easily sharing lay-outs between users and applications. The actual lay-out definitions are handled by means of the lay-out editors.

This private layout will replace the default form in the Mail registration application.

We select the properties (including associations or possibly chains of associations) to be displayed.



Users can select alternative layouts on the fly. This enables them to interact with the information in the most appropriate way, depending on the job at hand.

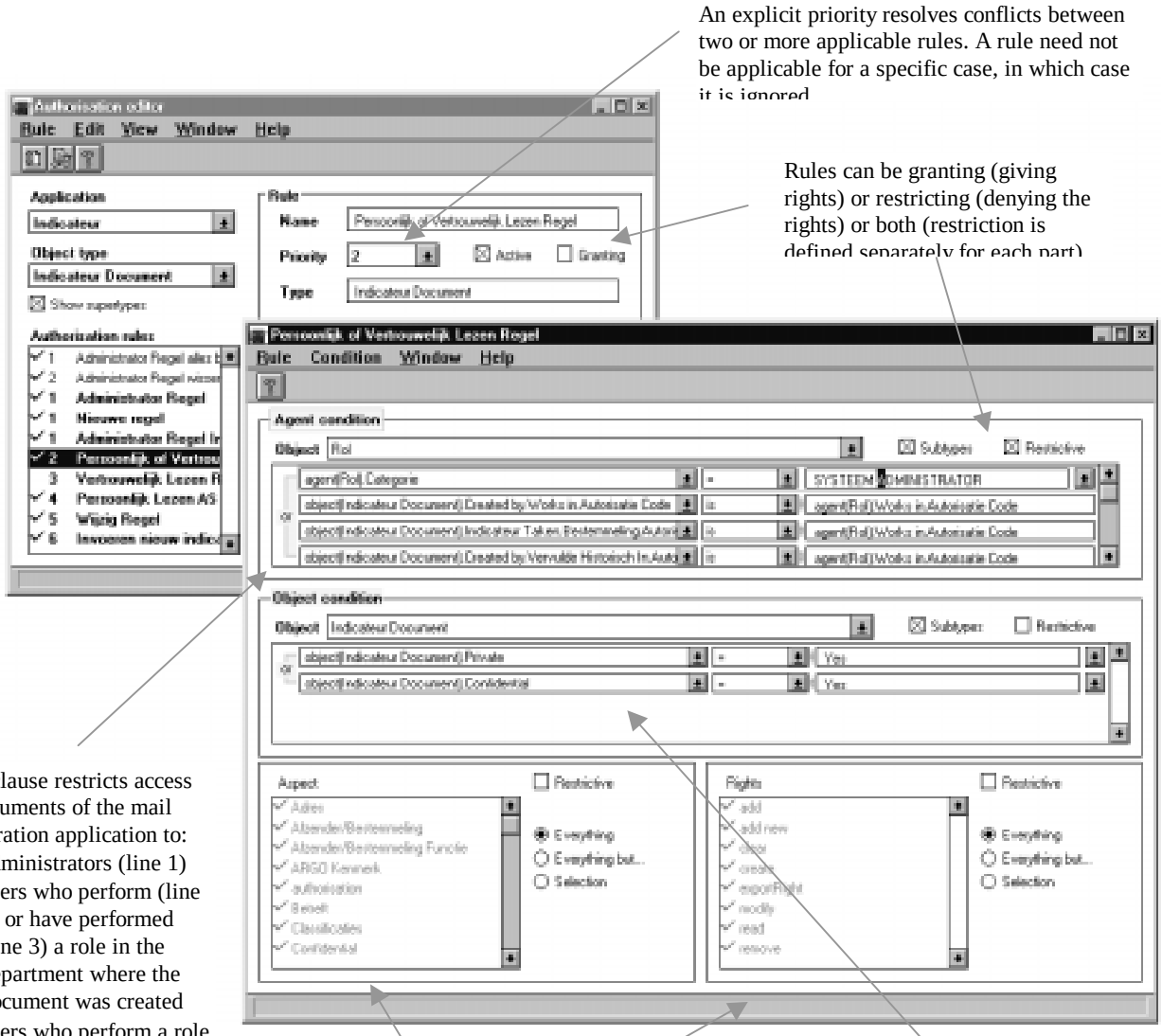
We re-use an existing overview list for this multi-valued association.

3.2.3. Extend or refine authorization rules

Access control is based on a set of authorization rules. These are not tied to any specific model and can be used to set up both very fine- and course-grained access. Authorization rules, just as action rules, can be context- and contents-sensitive. Some rules apply to everyone; other rules apply to groups of people whose members are determined at run-time. This way, authorization rules support cohesion, while giving considerable flexibility in supporting different team cultures.

Authorization rules contain four parts. The agent (= whom we are granting or denying access) and object (= what objects the rule is about) conditions are defined using the query editor component.

The aspect part specifies what aspects (e.g. properties) the rule applies to. The rights part denotes the privileges (e.g. read and export).

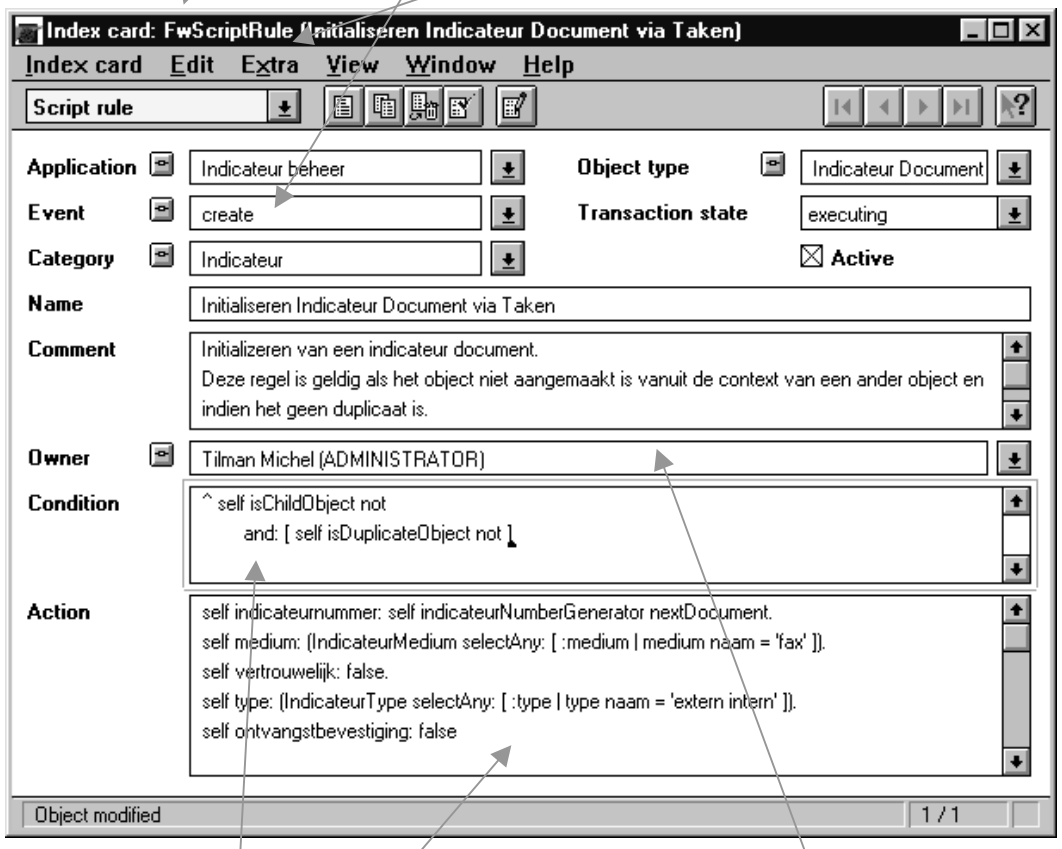


3.2.4. Define action rules

These rules capture business-semantics, set defaults, perform extra validation, impose constraints and add functionality. Action rules can cover an entire organization, particular functions or they can be limited to a specific application environment. Hence action rules give extensive support to autonomous teams while preserving overall consistency. Action rules come in two flavors: high-level template rules for defining workflow processes and script rules for handling more specific cases.

This script rule initializes IndicateurDocuments in the mail registration application. It is triggered immediately whenever an object of this type is created, whether by means of forms or in the scripting language.
Script rules are managed by means of a configured application. At commit time we validate and compile the rule. This, in turn, requires another (meta)-rule. A simple manual bootstrapping process solves the

Events can be system events (create, update, delete) to impose constraints, time events (enabling us to set up automated tasks easily), application events (such as login or application switch events and GUI events. The latter are typically triggered by application- and object-specific menu options and provide context-sensitive functionality. Thus the end-user tools such as the main window and forms tool adapt themselves to the application and object in



Rules consist of a condition and action part. Conditions specify whether the rule is applicable to the object at hand. An event may trigger several rules. The action parts of rules with valid conditions are actually executed.
The scripting language in condition and action parts 'extends' the Smalltalk language with a dynamic accessor protocol, automatic translation of Smalltalk-like select methods into query expressions and implicit access control when accessing or updating an object property.

Access privileges are validated for the login user, unless we define a rule owner. In the latter case we check the owner's privileges when accessing or updating

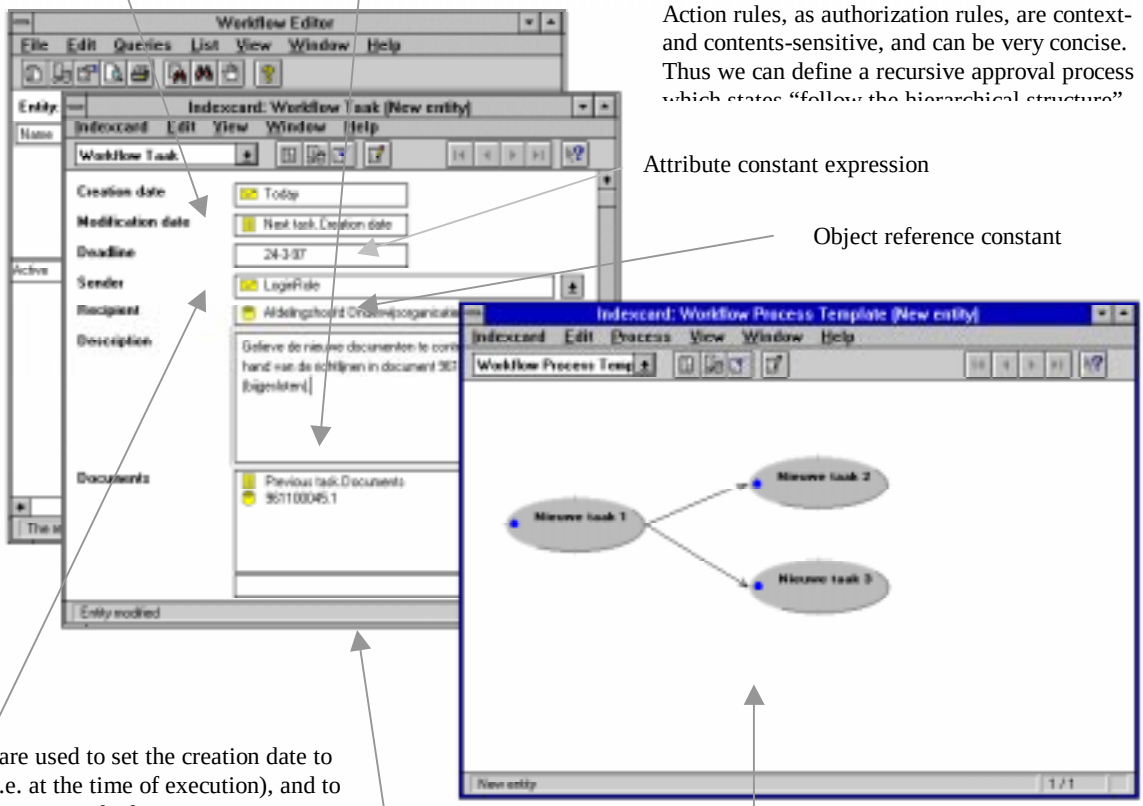
3.2.5. Define workflow process templates

With workflow process templates we specify default scenarios that the user typically follows in the context of a business process. Users can (usually) deviate from these scenarios and snap back into the pre-defined flow later on. Processes can be configured incrementally, e.g. between departments first, and within each department later on. Depending on team culture, these individual sub-processes are more or less strictly defined. In addition, we add automated tasks and dedicated private or shared in- / out-baskets to manage incoming and outgoing work.

This relative expression specifies to include the documents enclosed in the preceding task (property expression) and to add the document with reference number 961100045.1 (object reference constant expression).

This new task's modification date is set to the creation date (relative expression).

We configure script and high-level rules through end-user tools to support business rules or to offer extra functions. These options depend on the job at hand. For instance, when opening a form on an incoming task assignment, the tool presents the user with the list of options to can be performed on that particular task. In an approval process, these might be rejecting (in which case the requester gets notified) or approving (in which case the procedure continues along the default lines) the request. Action rules, as authorization rules, are context- and contents-sensitive, and can be very concise. Thus we can define a recursive approval process which states "follow the hierarchical structure"



Attribute constant expression

Object reference constant

Scripts are used to set the creation date to today (i.e. at the time of execution), and to set the creator to the login user. We use typed methods to select compatible scripts.

The user has opened an extended form on a new (follow-up) task, and is assigning default values to it.

The graphical tool displays workflow tasks, task dependencies, timing (in case of a process history) and status information. Users obtain more detailed information about individual tasks by opening the forms tool. They define queries to filter tasks from the diagram. Queries can also be used to highlight points of interest in the process.

4. Framework components

In this section we present an overview of some of the key elements in the framework architecture:

- the repository: end-user objects, meta-model and system objects
- the persistency component
- the main interactions of the framework components.

We borrow some terminology from the GOF design pattern system [Gam95].

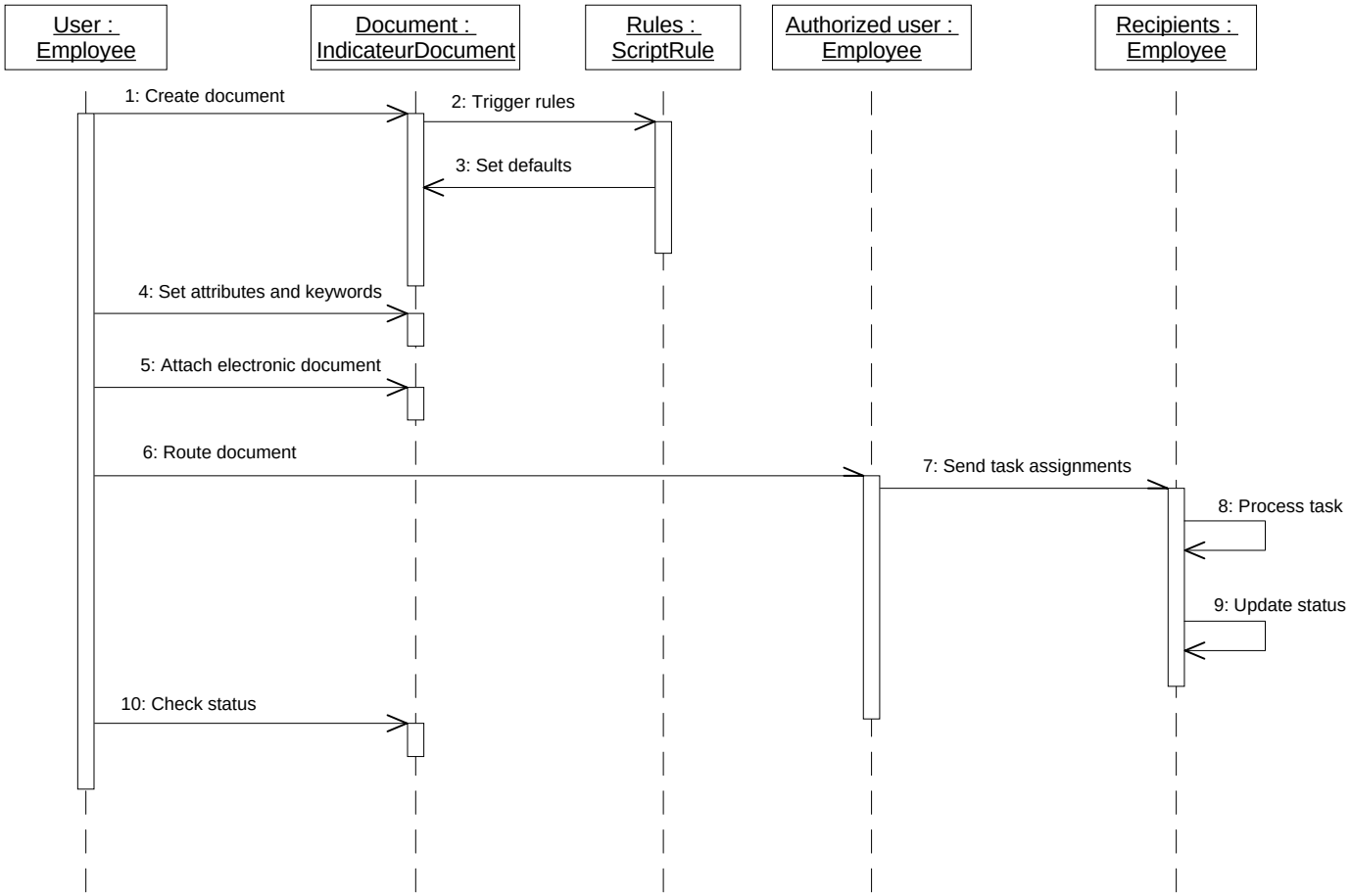
4.1. Example: the mail registration application

The mail registration application is used to register information about incoming documents and dossiers, to assign tasks for handling them and to keep track of their status.

When the user receives a document he creates an instance of the IndicateurDocument object type. He registers date, reference of sender, subject, summary, keywords using Argo's thesaurus etc. He indicates if the information is confidential. Some attributes, e.g. the identity of the employee who treats the document and the creation and modification date, get default values. The actual document is stored in the system using native Windows documents, HTML, PDF, scanned Tiff etc.

The IndicateurDocuments are routed to the person who has the authority to assign tasks and set priorities and deadlines. Depending on the nature of the document, it is routed through the organization for further handling. The recipients of the tasks are notified by E-mail. On completion or when further delegating the task they change the status of the task.

The employee responsible for the dossier keeps track of the status of the tasks in treatment, by querying the database and he reports to superiors if needed.



Handling mail requires several types of objects: IndicateurDocument, ElectronicDocument, Employee, OrganizationUnit (department), ThesaurusDescriptor (taxonomy of keywords and synonyms). We do not write code to represent these objects but model them in the repository. Nor do we code the relationships between these objects. Association types like "IndicateurDocument created by Employee" and "Employee member of OrganizationUnit" are stored in the repository along with their cardinality constraints e.g. "every IndicateurDocument is created by exactly one Employee" and constraints on attributes e.g. "the name of the Employee is mandatory".

We re-use objects from existing applications, e.g. Employee, ThesaurusDescriptor (managed within the documentation application), ElectronicDocument. The IndicateurDocument inherits common attributes from an abstract type (WorkflowDocument).

4.2. Repository

Hotspots should be easy to change. The framework stores hotspot objects in the central repository so that they can be changed without any programming. They can be changed by end-users, configurators or administrators, rather than by developers.

4.1.1. Object model

The repository contains three kinds of objects. The first kind are end-user objects, which are the objects that make up the business model. These are the objects that the end-user talks about. The second kind are system objects, which are objects that the developers or configurators talk about, but that are not part of the business model. For example, authorization rules and the internal

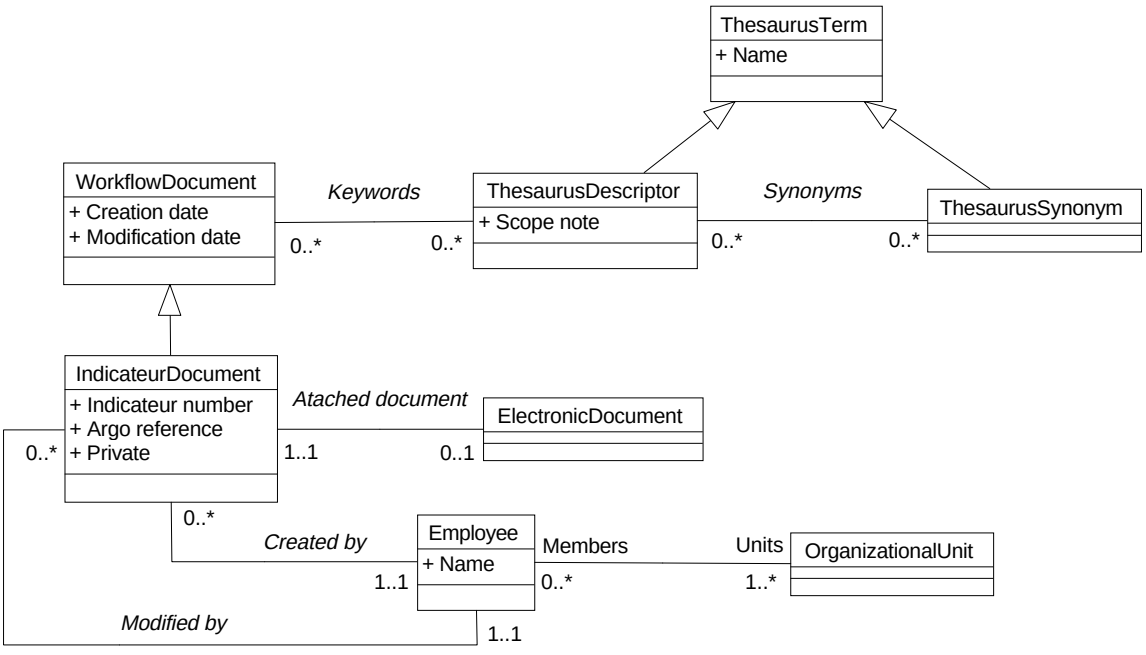
structure of electronic documents, such as versions and various representations, are system objects. The third kind are the objects that describe the business model and system objects (object model).

The object model describes the elements stored in the repository: object structures, associations and constraints. The meta-model describes the structure of the object model. It is expressed in terms of itself and is stored in the repository too.

End-user objects

End-user objects model organization structure, processes, tasks, data and relationships. The tools make no a priori assumptions about these end-user objects. This enables us to construct different business models. In fact, several models of the organization can be made to co-exist, which is useful for simulating future scenarios.

A IndicateurDocument keeps track of its creator and the organizational units its creator works in. There are different kinds of organizational units, for example departments and temporary project teams. Instead of modeling these different kinds of units through different object types, we use the Object Type pattern to classify the units dynamically. This enables us to replace the organization model with another one without having to change the object model. A mere change of repository population suffices. We have the flexibility to do this. For instance, authorization rules depend on the object model, but may also depend on the values of the actual objects in the repository (e.g. the kind of organizational unit).

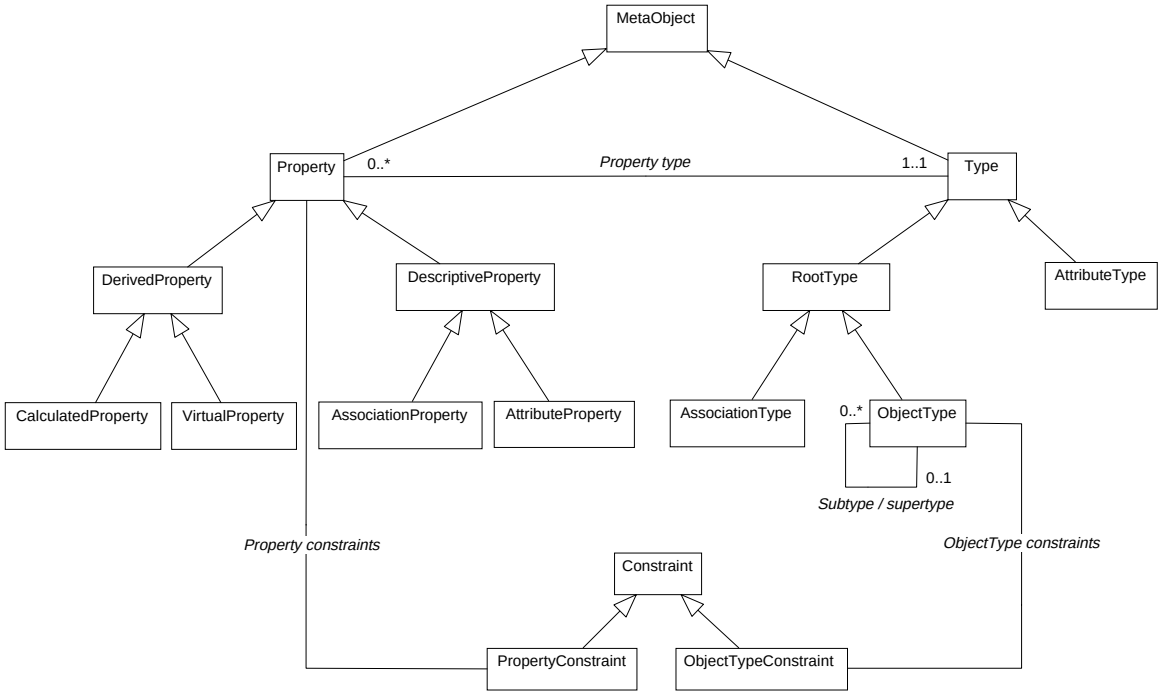


Meta-model

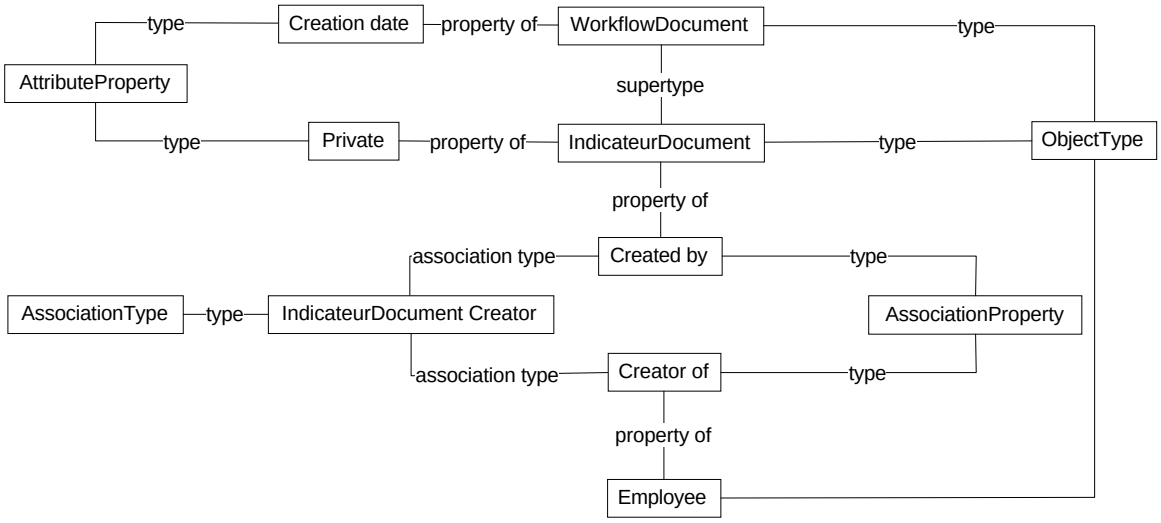
Each object has a specific type. An object type has a name and defines a set of properties. Each property has a name, a type, and also has other attributes such as whether it is mandatory, unique and represents a symmetric relationship.

If an object is of a particular type, than it is allowed to have properties that the type specifies. The values of a property of an object of type T must match the type of the property specified in T. Attribute property values have primitive types, such as dates, strings or numbers, and enumerated values over these types. Association properties reference other objects.

Object types can have subtypes. The property type may be overridden in subtypes. This is particularly useful for specializing abstract associations between two abstract types.



In the following picture, we present a small instance schema of the meta-model.



Our model supports N-ary and attributed associations. We use an example of attributed associations in the documentation center application. A document may be related to other documents. For example, it may be an addendum to, or a modification of other documents. We model this relationship as a symmetric association. The nature of the relationship is represented by means of an attribute of the association. The attribute takes its values from an enumeration set.

Individual properties can be constrained to take values in a subset of the type value set. We also provide object and inter-property constraints, describing conditions to be satisfied prior to committing changes to (groups of) objects. These constraints are specified by means of script expressions (Smalltalk code). Property types and constraints may be overridden in subtypes.

As an abstraction mechanism to hide details of the object model or to de-normalize the user's perception of the object model (e.g. when using the query editor or defining lay-outs in the forms tools) we use derived properties. These are read-only properties representing values or aggregates of properties. They are defined by means of query expressions (derived properties) or script expressions (calculated properties). Derived properties can be used as regular properties in most cases (calculated properties cannot be used in queries).

Our framework treats objects that make up the object model in the same way as objects managed by end-user applications. Thus we can re-use the forms tool to edit the object model. Authorization and action rules apply to this object model editor application. For instance, we can define action rules that are triggered whenever the model changes, e.g. to update database tables and columns, or to invalidate rules. And expressing the meta-model in terms of itself enables us to go one step further: the definition of our kernel meta-model can be extended to specify new types of constraints; the semantics can then be implemented by means of action rules.

System objects

For proper functioning of the end-user and configuration tools, system objects are needed.

We use many of these system objects to specify end-user applications**,** such as application environments, stored queries, action rules and object behavior. These types of objects often link object model and meta-model.

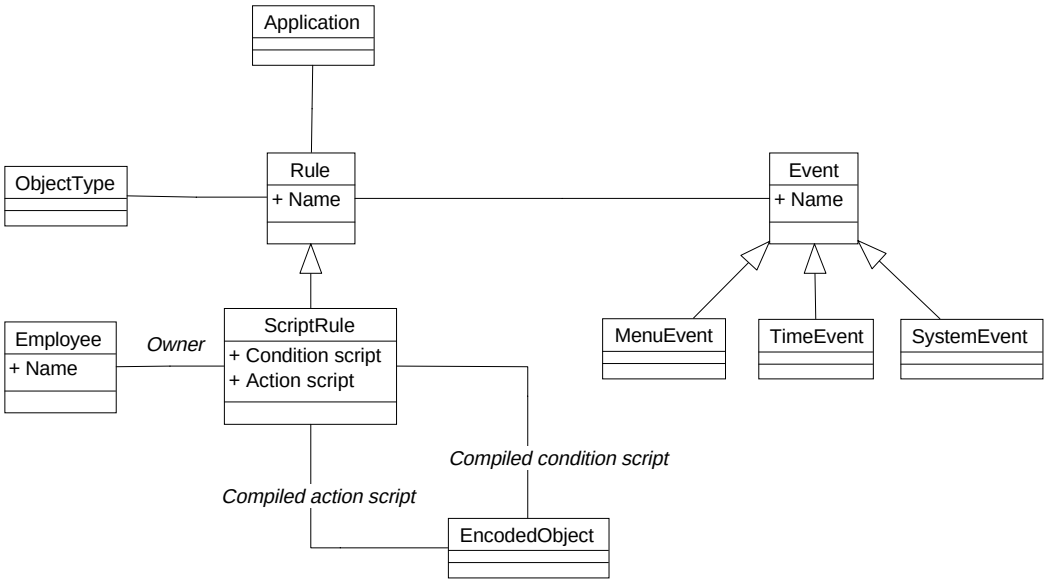
Several types of system objects, such as action rules and stored queries, contain parts that would require complex models and need not be queried or updated separately. These parts are modeled

as encoded objects that provide an interface to encode (decode) Smalltalk objects into (from) a string representation when updating or accessing the object.

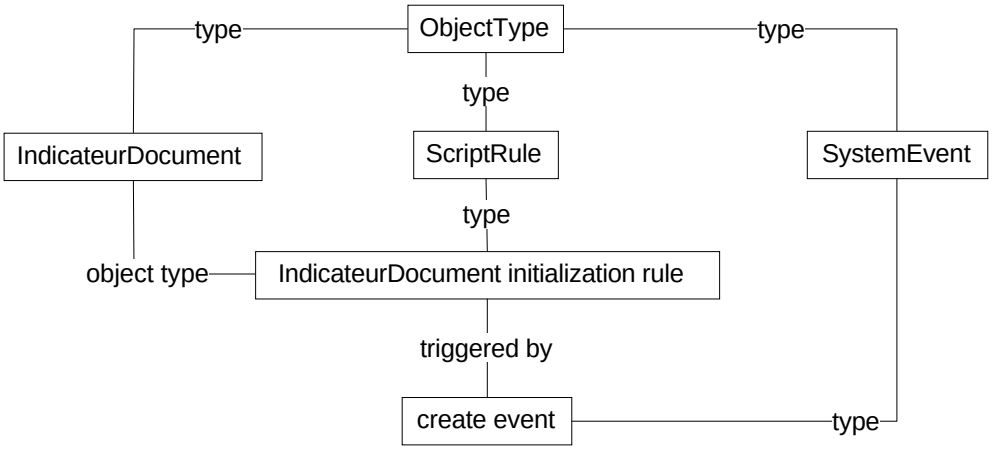
Example:

In the Indicateur example we use initialization rules to set the default values of new IndicateurDocuments e.g. IndicateurDocument is not confidential and the CreationDate is today. The CreationDate rule is defined in the super-type WorkflowDocument and inherited by all its subtypes.

All these rules are modeled explicitly in the repository. Rules have an association with object types. Condition and action scripts of script rules are modeled as strings containing the (Smalltalk) source code. The compiled versions of these scripts are modeled as encoded objects, associated with the rules. When committing changes to a script rule, script rules (acting on instances of ScriptRule) are triggered that validate the script expressions and generate the compiled code.



In the following picture, we present a small instance schema.



Other system objects represent special business objects, such as login-users, electronic documents and thesaurus keywords, that need some dedicated framework components. These objects are usually presented to the end-user through special tools, such as the query editor and viewers for scanned documents. But since system objects are explicitly modeled and stored in the repository, for administration purposes as querying and reporting we don't need additional tools. The generic end-user tools can be used instead to set up management applications.

As the framework itself evolves, and tools are modified or added, changes may be made to the pool of system objects.

Object representation

By default, repository objects are represented at run-time by instances of a Smalltalk class (FwEntity). These instances keep track of the identification (object id and type) of the repository objects they represent. In addition the class provides an interface for accessing and updating the properties. Instances typically contain partial information, i.e. we load only those properties from the database we actually need. Accessor methods are created on the fly when using the scripting language, and object behavior is loaded when needed.

This approach usually suffices for end-user objects. In the case of system and meta-model objects, the framework components often require extra behavior or state. We provide the means to add subclasses to FwEntity and to link these classes to specific types of repository objects.

4.1.2. Application environments

Application environments restrict the view on the repository by specifying what objects can be accessed, queried and created. Environments can be structured hierarchically: nested environments further specialize the view on the shared object model. Action rules, views and queries are inherited.

Standard functions, such as scanning and indexing of documents, or reporting, can be removed from the environment if not required for a given job. This is particularly useful for novice users.

For every object type, views can be configured for use in list-, print-, export- and forms tools. These views are either private or shared among all users of the environment. Default views can be defined. Individual views take precedence over shared ones. Views may be re-used in other views.

In a similar way, queries may be saved for private or shared re-use. Defaults can be specified. Queries can be re-used in other queries. List views can be coupled to specific queries.

Action rules can be defined within the context of the environment, e.g. to set appropriate defaults, trigger a cascaded delete or to add specific functionality, for instance a batch update of a list of objects.

The use of nested environments typically leads to the following strategy: definition of an abstract application with re-usable assets, followed by dedicated sub-applications, such as full management and retrieval applications, and even simplified applications with many pre-defined queries and lay-outs for novice users.

4.1.3. Electronic documents

Electronic documents represent unstructured information and require dedicated tools for creating, viewing, editing and printing. Making electronic documents available in end-user applications is straightforward: we only need to associate documents with the relevant object types in the application. Thus electronic documents are not perceived as stand-alone objects by the end-user, rather they are enclosed in objects having more domain specific meaning.

The apparently simple document structure hides a more complex object model that contains the necessary information for the various tools, such as versions, different representations, storage information and instructions to delete or copy documents.

Versions

Documents are perceived by the end-user as logical documents, which have a name and can exist in one or more versions. One of these is the preferred one. We do not maintain a version tree, as it would needlessly complicate use of the tools, neither are new versions created automatically by default. If necessary, we define rules to override this default behavior.

Document versions come in two flavors:

- basic documents, corresponding to physical documents which have a well defined format and type
- complex documents, containing different representations (basic documents) for the same logical document version.

Basic documents are stored on physical media in one ore more copies, which are managed by the document caching process manager.

Representations

Representations enable users to use different views on the same logical document, e.g. to create a text representation of a scanned document, which can subsequently be stored in a full-text-indexing database. Other examples are alternative representations using a more widely available format, e.g. PDF for Internet, or a TIFF representation of a frozen Word document that may only be annotated.

One of these representations is the master document. Invalidating the master invalidates the other (slave) representations as well.

Document caching process manager

A document can exist in one or more copies on several cache and the optical storage medium. A background process (implemented by means of background processing rules) keeps track of the status of these copies to make sure that the up-to-date copy is stored and retrieved.

Instructions to e.g. cache or delete documents are not necessarily carried out immediately. Instead, they are listed as instructions to be executed by the background process. This condition takes into account model and contents of the objects in which the document is enclosed (the document owner). This enables us to set up very flexible caching strategies.

Whether an electronic document is persistent is ultimately derived from the repository information: an electronic document must be owned by a repository object. Hence, in case of failure, our recovery strategy makes the document base consistent by starting from the information in the database.

4.1.4. In- / outbaskets

In- / out-baskets maintain lists of incoming and outgoing work. What appears in a basket is defined by means of a query expression, hence it is not restricted to task objects only. For each user we configure a generic in-basket containing tasks sent to him or her. In addition, dedicated baskets may be set up, e.g. to follow up particular dossiers.

The in- / out-baskets tool uses a special application environment that re-uses several system objects from other environments, such as lay-outs.

4.1.5. Workflow

Workflow processes represent cause-connected activities to be performed in a (not necessarily unique) particular order to achieve a specific goal, e.g. the central board decision process, ad-hoc processes to register and handle incoming documents, or the process to create and distribute new norms and regulations towards the school.

Activities are modeled by means of task types. Tasks may involve several participants, e.g. the person who issued the task assignment and the recipient of the task.

Pre-defined process templates provide options to guide the user through default process scenarios. Processes can also be triggered automatically by specific events, for instance at the start of each day.

Tasks

A task specifies the interaction between participants necessary to perform a specific activity. Participants can be any object, since object types are independent of the actual rule engine, process editor or in / out basket model.

Some workflow tools, such as the graphical process editor, assume the existence of a minimal set of common properties defined for all task objects:

- the status property defines the various relevant states for a task with regards to commitment of the participants involved in the task;
- the previous task / next task association maintains task dependencies;
- timing information includes date sent, reminder date and deadline.

Other properties can be added as needed in the various subtypes of the Task object type. These properties can be viewed or edited using regular list views and forms, or listed in the graphical process history overview.

Processes

Processes represent the running history of an ad-hoc or pre-defined workflow process (or combination of both), i.e. the tasks performed so far. Tasks are always associated with exactly one process. A process will automatically be created when the user activates a process template from scratch. Within the context of a process, sub-processes may be started by activating process templates.

Process templates

A process template defines default scenarios for a workflow process, by listing the default tasks (more precisely types and default values) to be performed, and, at each step, the options to move

from one task to another. These default scenarios are modeled by means of a collection of action rules.

Whether users may deviate from this default process is ultimately determined by the authorization mechanism. Instead of relying on programmed exceptions, the process template definition gives users the means to let non-default processes snap back into the default flow if some rule condition is satisfied. Alternatively, users can synchronize explicitly with one or more key steps in the template by activating a rule specifically configured for these purposes.

The mixture of ad-hoc / pre-defined processes and tasks / sub-processes, allows us to define process templates incrementally. For instance, a process template may be initially defined at the level of individual departments, and refined later within each individual department. These sub-processes may be more or less strict, depending on each department's working practices.

Implementing other process models

Other workflow process models can be implemented too. We illustrate two existing approaches here:

Workflow processes based on intelligent work objects [Kar90] can be modeled using script rules. The work object contains the necessary data and state information. A script rule describing the routing algorithm is activated whenever the user opens a form on the work object.

Process models based on Speech Acts [Flo93] can be modeled using high-level rules. Conversation objects contain the necessary data and state information. The condition expression refers to a matrix describing permissible state transitions. Default acts on conversations, such as promise, decline or counter-offer can be configured by means of action rules.

4.1.6. Authorization rules

The authorization mechanism makes no a priori assumptions about a particular model. It allows to set up simple user / user group access controls, as well as much more sophisticated, fine-grained context- and contents-sensitive access privileges. It relies on a knowledge-base of authorization rules. Each rule either explicitly grants or denies rights to users for certain aspects (mostly properties) of objects, or tells us nothing at all (i.e. the rule may be irrelevant or undecided). Thus the authorization mechanism revolves around a strategy to find the most appropriate rule that conclusively answers the question 'does this user have these rights for these aspects of this object' with yes or no. The main idea is to capture the default cases in rules with a large scope, and to add rules catering for exceptions in a piecemeal fashion. The actual solution depends on the problem domain (e.g. do we restrict access in the default cases and explicitly grant access in the exceptions, or vice versa), but often reflects global policies within the organization regarding access control and privacy of information.

Example:

- All users may read all documents (default case for all users, aspects and types of documents, granting)
- No user, except administrators, may update documents (default case, denying in the former case, granting in the latter case)
- The colleagues of an employee may update a document he or she created (specific case, granting)
- In the mail registration application we use documents that may be flagged as private. Only members of the same department as the document creator or employees higher up in the hierarchy have access to private documents (specific case, denying).

Since rules may conflict, it is necessary to provide conflict-resolution strategies.

Definitions

Each authorization rule consists of the following 4-tuple:

- the agent set (Ag) specifies the users to whom access is granted or denied
- the object set (O) specifies the subjects of the rule
- the object aspects (As), e.g. a selection of properties
- the rights (R), such as read, write, add, remove and export.

These sets (called dynamic sets) can be dynamic, based on a condition that specifies the elements belonging to the set. Conditions make use of query expressions defined with the end-user query editor, or, in some cases, Smalltalk expressions. In addition, static sets can be defined by explicit enumeration. In practice, agent and object set are usually dynamic, the aspects and rights static. Dynamic sets can refer to context variables, such as the login-user.

Each of the sets can be made *restrictive* (denoted as [S], S a dynamic set). A rule is restrictive if at least one of its sets is restrictive. A rule can also be *granting*. Rules can be both granting and restrictive. Using these rules, we explicitly grant or deny access control.

Rule semantics

The following example illustrates the semantics of authorization rules.

Applying a rule {Ag, [O], [As], R} to an agent-object-aspect-right tuple {ag, o, as, r} will answer true, false or undecided according to the following strategy:

- true if ($ag \in Ag$) and ($o \in O$) and ($as \in As$) and ($r \in R$) and (R is granting)
- false if ($ag \in Ag$) and ($o \notin O$) and ($as \notin As$) and ($r \in R$) and (the type of o is a subtype of the type of the object set O, i.e. the rule must be *relevant*)
- undecided otherwise.

If we substitute Ag = {Department heads}, O = {Proposals}, As = {Visa property} and R = {Edit}, then this rule translates into: “Department heads may only edit the visa property of proposals”. This rule is not relevant for objects of types other than proposals.

Combining rules

Rules can be inherited according to the object set type hierarchy. Rules defined for a particular object type have precedence over inherited rules. Rules defined within the same object type are explicitly prioritized to resolve conflicts.

As rule evaluation may be undecided for a particular tuple {ag, o, as, r}, at least one default rule is needed that answers true or false for all tuples. This can be a rule defined at the common supertype of all object types.

Using authorization rules

Authorization rules can capture complex requirements succinctly.

Example: “A user has access to all documents created within a workflow process if at least one of these documents has been created by a member of his organizational unit or a unit higher-up in the hierarchy”. This rule can be defined in the configuration tool without any scripting. Rules like these will typically use context variables, for instance to refer to the login- user.

Rules can be applied to all elements in the repository, including system objects such as document annotations, applications and even authorization rules.

The authorization mechanism is very expressive, but managing the rule-base may become less evident if number and complexity of the rules grow too large. Two techniques can be applied to simplify maintenance of access control:

- Reification of hidden elements in the rule definitions.

  Example: to specify the applications a user has access to, a simple end-user application has been configured. This application maintains explicit associations between users and environments. Several authorization rules have been reduced to one meta-rule checking this information. Note that this rule applies to the application itself.

  Similar techniques can be used to manage e.g. large numbers of users.

- Modeling additional, simple authorization rule types to be managed through end-user tools. Transformation methods are needed to convert these rules into system-level rules.

Thus we can configure specific authorization applications, targeted towards a particular use with a minimal effort.

Optimizing performance

By default, authorization rules are executed locally. In some cases, this generates considerable overhead when many objects are retrieved from the repository, only to find out that the user has no access. And even if the user has access to a particular object, the authorization rules may require additional associated objects to be downloaded in order to verify the access privileges. To optimize performance, rules are combined with database queries if appropriate.

Design

The authorization mechanism presents one aspect of the framework [Xer96]. At first sight, its functionality should be woven together with the object store's to ensure consistency. However this behavior is not always required nor even wanted. Performance issues take precedence in some cases. The Facade-pattern provides a clean mechanism to have different framework components access the object store in different ways, for instance, with or without access control. Ideally, however, this requires context-sensitive access control for objects [Ric92], whereby only trusted components may access the object store directly.

4.1.7. Action rules

Objects obey constraints defined in the model and exhibit behavior that is used across all applications. End-user applications sharing these objects may however require additional, often context- and contents-sensitive functionality and semantics (business rules) [Gra94]. Action rules capture these requirements. While action rules allow semantics to be different across applications or workflow processes, they do not violate the global constraints defined in the model.

The main elements of action rules are events, conditions, actions and rule scope. The type of event determines whether the rules implement some business rule or provide some additional functionality to the user.

Examples:

- When creating documents we set the creation date and creator. When committing a modified document, we update modification date and remember the last author. Documents in the mail registration application need extra initialization to generate a unique id based on the context: is the document the start of a new workflow process or is it created within the scope of an existing process?
- When removing objects, we delete attached electronic documents (if any) by default.
- We add extra functionality to perform operations not directly provided by the generic tools (e.g. a particular report), or to support repetitive work. In the context of workflow processes

we often need to synchronize the state of different related objects and guarantee the overall consistency; instead of complicating the tasks of both users and configurators, we provide extra functionality to perform the necessary operations in a controlled way. All these rules are typically triggered by menu events corresponding to menus that only appear in the relevant context (application environment and object type).

- Several tasks must be performed automatically, often at regular times. Rules can be triggered by time events generated by the clock.

Events

Action rules are triggered by events. These may be generated by the system or by explicit user actions.

Rules triggered by system events generally affect the semantics and include: creating, duplicating, saving, deleting, locking or modifying an object or handling exceptions. User actions either correspond to specific menu events providing extra functionality or generic events when using the tools, such as switching applications, selecting an object or lay-out.

A single event can trigger several rules, which enables us to add behavior in a modular way. All enabled rules (i.e. rules for which the condition is satisfied) will be activated.

Activating rules

Action rules have access to context-variables, such as the login-user or the interface component that triggered the rule. To activate a rule we apply it to a particular object (the receiver) in a given context. If the condition is satisfied, the action will be executed.

A rule is scoped: it can only be applied to objects of a given type (including subtypes). The scope can be further limited to a particular application or workflow process.

Types of rules

We provide essentially two types of rules: script rules and high-level rules

- **Script rules**

Script rule condition and action expressions require our scripting language (basically Smalltalk with dynamic object accessor methods, implicit access control and high-level query access to the repository), hence they are not targeted towards the average end-user. The scripting language has access to the context-variables, and can re-use object behavior (the latter uses the same scripting language, but has no access to the context-variables, making it context-independent).

- **High-level rules**

To allow end-users to add their own rules, we provide high-level rule types. High-level rules explicitly model common script practices in the repository, such as:

- initializing objects to default values (constants or results of query expressions)
- updating an object at commit time or when a property has been modified
- creating a follow-up task in a workflow process, based on certain conditions, and updating the state of the current task.

Rules components

Conditions are specified by means of query expressions. These can be created by the query editor. The action part uses one or more *update expressions* to specify new values for object properties. The first expression refers to the receiver. Executing the expression updates the receiver. Additional expressions specify type and initial values for new objects to be created when the rule is activated.

Examples:

- Rules with one expression are typically used to set default values for a newly created or duplicated object, or to update an object's properties as the result of a user action. The latter is often used in batch procedures to update a series of objects.
- Rules with two expressions are used to support definition of workflow processes. Given a task (the receiver), a rule can be defined to create a follow-up task and update the current task state. We define a workflow process consisting of a set of tasks by describing for each task the options (and conditions) to start other tasks. Each option then corresponds to a rule.

High-level rules can model recursive processes, e.g. to implement an approval procedure that mirrors the hierarchy.

Update expressions

Update expressions can be edited by means of extended forms. No scripting is needed. These expressions behave similarly to regular objects, i.e. they have a type and properties, but the latter accept expressions in addition to values.

The following expressions can be used:

- constant expressions, such as a date value (attribute property) or a reference to another object (association property)
- query expressions,
- property expressions (see persistency component),
- scripts (using typed methods).

For multi-valued properties, individual elements may be flagged for addition or removal.

Example: The following approval rule (in pseudo-language) illustrates the idea:

```
event
        select 'forward' menu item
condition
        currentTask.approved = true
current task expression (type ApprovalTask)
        currentTask status: finished
new task expression (type ApprovalTask)
        newTask recipient: (Manager selectAny: [ :mgr | mgr division = 'Finances' ])   (query)
        newTask documents: currentTask documents
        (relative)
        newTask approved: false                                                        (constant)
        newTask dateSent: self dateToday                                               (message)
```

This can be read as: "When the user has approved a request (condition) and has selected the 'forward' menu item (event), the current task will be flagged as finished (current task expression). A new request for approval will be sent to the Finances manager. The documents enclosed in the current task will be attached and the date will be set to today's date (new task expression)".

Stepwise structuring

We will formalize other common practices in the future. Operations on objects such as saving, deleting or locking often require that the same operation be performed on related objects

(cascaded saves, deletes and locks). A variant of the high-level rules can be used to model these practices. Extended forms can equally well be used to edit these new types of rules. This incremental process of formalizing practices in the repository is an example of stepwise structuring [Häg89].

Authorization

When executing rules, it may be necessary to override the current user's privileges. Hence, each rule can be given an owner, whose privileges will be used instead. In addition, the user must have the necessary privileges to execute the rule.

Managing action rules

Whenever the user switches applications, the action rule manager retrieves and caches the rules relevant for that particular application. For each rule triggered by system events we register this dependency, as well as the transaction state in which the rule should be triggered.

Example: Triggering rules when a particular property value changes will typically be performed at once, whereas more global object updates are usually performed at pre-commit time. User-defined constraints are managed in a similar way.

4.1.8. Background processing rules

Background processing rules specify activities to be performed automatically on elements in the repository. Typical uses are process managers handling requests to migrate document copies to a particular storage medium or to convert notification requests generated by the system into E-mail messages, and automated tasks within the context of a particular application or workflow process.

Background processing rules used to be dedicated objects and framework components. Now they are implemented by means of action rules triggered by time events. Time events enable us to set up elaborate scheduling strategies, including recurrent events.

The action rule manager automatically schedules any background rules defined in a specific application whenever the user selects the application. Whether the background rules run within the context of an end-user application on his or her client, or on a dedicated client PC is solely a matter of configuration. Thus we can easily set up appropriate applications on separate clients in order to achieve the right degree of load balancing.

4.2. Persistency component (object store)

The object store component bridges Smalltalk objects to the repository, adding persistency in an abstract and orthogonal way. Transactions and locking provide support for interactive applications, even when used off-line. The component also contains a bridge to a full-text-indexing engine.

The object store provides strategies to fine-tune the database performance.

4.2.1. Bridge-pattern

The bridge actually uses two layers. The uppermost bridge maps Smalltalk objects, transaction operations, locking and abstract query expressions on the abstract database layer. The lowermost bridge maps abstract database operations on the physical layer.

Example: In the case of relational databases, object store operations are translated into operations involving abstract relational columns and tables (upper layer). The lower layer translates these operations into database-specific SQL statements.

This two-level schema allows us to change the physical database without impacting the upper bridge. The lower bridge may have to implement abstract operations that can not be mapped

directly onto the physical database, such as recursive queries in the case of relational databases. We deal with the database type, e.g. object-oriented vs. relational, in the upper and lower layer.

4.2.2. Transaction operations

We provide the necessary behavior to create, update and remove repository objects. All these operations must be performed within the context of a repository transaction that keeps track of local changes to one or more objects. At commit-time the logical transactions are translated into the underlying database operations and executed (typically in a database transaction). Locks, constraints and action rules give us the means to implement the necessary integrity.

To avoid concurrent update of the same property of an object within two repository transactions, properties are locked automatically (at the client side) by the transaction performing the update.

Traditional transactional systems are too much targeted towards a programmatic use. To better support interactive, multi-window applications, the framework provides transactions that can be nested hierarchically. These sub-transactions need not be committed strictly sequentially.

A basic premise is the possibility to commit partial results of transactions [Nod91]. If a sub-transaction fails, the state will be retained. Committing the main transaction later on will result in an attempt to commit sub-transactions, including previously suspended sub-transactions. Nested transactions inherit the locks set by their parent transactions. This gives the freedom to commit partial results, while still retaining overall consistency with regards to the transaction hierarchy.

Committing the transaction does not end the transaction: this happens when the transaction is explicitly released.

4.2.3. Locking

In addition to locking individual properties, which is purely a local operation, we need to provide (global) locking of objects that are visible for all clients. For this we rely on persistent locks, which are better suited towards highly interactive applications and remote (off-line) use.

When setting a lock we either specify whether the lock should be released at the end of the transaction or whether the lock should survive the transaction. In the former case, the user can e.g. start editing an object and saving intermediate results, while still retaining existing locks. In the latter case we provide a timestamp specifying when the lock expires. This enables the user to lock some objects for a few days, export the data and process the data off-line, before importing changes and freeing up the locks.

Action rules can be used to capture more complex locking schemes, whereby locking a group of related objects is presented as one logical locking operation.

4.2.4. Query expressions

In order to query the repository we provide an abstract interface for building query expressions. Query expressions are defined in terms of the object model and describe the instances of a particular type (possibly including all its subtypes) to be retrieved. A condition expression specifies the criteria to be matched by attribute and association properties.

Query expressions can be described in the following pseudo-language:

Figure 20 Query expressions

queryExpression = “select” objectType “where” conditionExpression

conditionExpression = “true” | “false” | simpleExpression | conditionExpression “and” conditionExpression | conditionExpression “or” conditionExpression | “not (”conditionExpression“)”

simpleExpression = propertyExpression operatorExpression valueExpression

propertyExpression = property {“.” property}

operatorExpression = “is” | “like” | …

constantExpression = “true” | “false” | integerExpression | dateExpression | objectReferenceExpression

valueExpression = propertyExpression | constantExpression | conditionExpression | contextVariable.

Operators include relational operations (e.g. documents with the modification and creation date) and membership of sub-queries. Property expressions are either properties of the object type being queried, or chains of properties starting from the object type. In the latter case, all but the last property must be an association property. In addition, context variables can be used (such as the current date or the login-user).

Example: To retrieve all documents in the mail registration application created by members in the user’s department, we use *select IndicateurDocument where createdBy.units.members includes: loginUser*. In this example we follow the association between employees and the organizational units in both directions.

We make extensive use of the interpreter and composite patterns in the query expression component.

Selecting properties

To fully support the use of partial objects, we provide the means to specify the properties or chains of properties retrieved (if none is provided, we default to retrieving the object and type reference). To this end we extend the query expression with a list of property expressions (this is a simplified case: we may also need tree-like expressions to specify we want to retrieve more than one property of associated objects in case the association is multi-valued).

Example: In the example above we might want to retrieve the names of the employees who created the documents, thus adding the *createdBy.name* property chain to the query expression.

Caching associated objects

When retrieving properties of associated objects, as in the example above, we do not cache the associated objects themselves by default. Instead we dynamically create virtual properties containing the values retrieved. If we want to load the associated objects too, we specify this explicitly by enumerating the list of property expressions to be cached.

Using query expressions

The query expression ‘language’ gives us a considerable degree of freedom to tweak access to the database with regards to both memory footprint and query response time. For example we combine (parts of) authorization rules with the query expression and we set up strategies for loading the properties to be displayed in the forms tool.

4.2.5. Object-database mapping strategies

How objects in the repository are mapped on the (abstract) database elements is specified through mapping strategies. To fine-tune performance or to map the object model onto existing databases, different strategies must be available. The following strategies for relational databases are provided and can be mixed:

- a table for the proper properties of an object type
- a table containing proper and inherited properties of a type
- a table containing all properties of a type and all its (recursively enumerated) subtypes.

The last strategy is very useful when subtypes are required without additional structure, but with different behavior. This often makes modeling cleaner.

Additional strategies specify the mapping of associations (separate tables or foreign keys) and enumerated attributes (a separate table containing the possible values and references to this table vs. in-place storage of the values instead of the references).

These strategies are explicitly defined when editing the model. Conversion modules restructure the database when changing the mapping strategy.

4.2.6. Persistent caching

Access to some objects in the repository is often required on a more permanent basis. This includes the part of the object model that is needed for a particular application as well as systems objects (such as lay-outs, rules, queries and preferences) in the scope of the application. Without persistent caching, these objects would be reloaded each time the user logs in and selects an application. This is particularly relevant for remote (on-line) use of the tools.

The cache for these system objects is structured as a dependency graph of sub-caches. For instance the query and lay-out caches depend on the application and login-user. A consistency mechanism using timestamps updates the part of the cache that has been invalidated. Any dependent sub-cache is recursively invalidated and reloaded. We simply save the cache with the Smalltalk image. Less needed application / user combinations can be purged from the cache in order to keep the footprint down.

4.2.7. Selective caching

The default caching strategy purges objects from the cache that are no longer referenced. However, it is often useful to cache or pre-fetch often-used query results for the duration of the login-session or within the context of an application. Action rules triggered by login or application switch events enable us to set up such strategies.

4.3. Main interactions

When the user logs in and selects an application, we do not start a generated or hard-coded application. Instead, the framework tools and components interpret the specifications of object model, constraints, access control and application environments (apart from our scripting language, which uses compiled code). Users can even select and combine several application environments at run-time, e.g. the action rules configuration application together with the ‘target’ application we are defining rules for.

In order to get some understanding about the way the framework components interact, we describe how the framework tools act upon the application specifications stored in the repository.

4.3.1. Log on

In order to bootstrap application start-up, a minimal object model is loaded at all times in the client, such as the kernel meta-model and the object model of the system objects. At login, we retrieve kernel authorization rules (in particular the rules pertaining access to the applications), and application-independent action rules and constraints, as well as the user's preference settings. For many system objects most of the details of the specifications are stored in encoded form. As this code is fairly large and usually not needed immediately, we retrieve these encoded properties lazily in most cases. Other application-independent system objects such as in/out baskets and thesaurus are loaded purely on demand basis.

4.3.2. Selecting an application

When the user selects an application, we load the authorization and action rules, object behavior, stored queries and list and forms lay-outs for that user / application combination. Since authorization rules and object behavior are application-independent, and the scripting language does not prohibit use of object types outside the application environment, we rely on a two-level schema for retrieving these objects. We load the rules and methods defined on the object types in the application. If any others are needed as well, we retrieve these lazily.

4.3.3. Main window

The main window presents a view on the object types (and properties) that can be queried and created. This information is retrieved from the application's definition.

Selecting an object type in the query editor results in installing a suitable view in the query result list. Therefore the list view requests a default lay-out. This is either a pre-defined lay-out or, if none exists, a lay-out generated on-the-fly. In case the user selects any of the pre-defined queries, the attached list lay-out is selected. Note that the lay-out is a specification, not a user interface component: the latter is constructed at run-time based on the lay-out definition.

When listing the query results, the list view filters out unauthorized information.

4.3.4. Forms tool

When the user opens the forms tool on an object, we install an appropriate view based on a lay-out selection strategy, similarly to the list view. The forms tool, however, offers more configuration options: depending on the property, different editors may be selected, each one with a corresponding set of preferences. This leads to a more elaborate lay-out generation strategy. The forms tool further processes this lay-out definition to identify unauthorized and read-only properties and to adapt the settings of the property editors accordingly. This process is repeated for every object, as authorization rules may depend on individual objects and even property values.

We provide a kernel set of functionality, suitable for all objects. Depending on the type of properties displayed in the forms tool, we extend the options, e.g. with electronic document management and thesaurus functionality. In addition, we install application- and object-dependent options defined by means of menu-driven action rules.

The forms tool acts directly on transactions. Hence updates performed within the forms tool trigger the necessary action rules and constraints, similarly to any other component. Thus we support the same behavior when manipulating objects, whether working programmatically or via interactive tools. Execution of action rules and constraints may generate itself additional update events.

4.3.5. Central core component

We rely heavily on caching application specifications and computed results, in addition to the regular object store cache. The various system objects are managed by dedicated caches or

managers. Setting up and releasing these caches is managed by the *core* component. Upon installing the action rule and constraint caches we also register these as depend of the object store, in order to link transaction operations to the execution of action rules and validation of constraints.

Another important role of the core component is the identification of services requested by the various tools (and other components). For instance, a tool needing access to object model and object store retrieves references to the appropriate components from the core. Thus we avoid the need for managing elaborate centralized API's.

5. Bootstrapping process

The bootstrapping process is driven by our observations that most configuration tools essentially manage specific types of objects in the repository, to which we need to add some extra functionality, validation, consistency rules and specific caching. Often, an alternative plug-in property editor is all that is required to access these objects easily through end-user tools.

Examples

To bootstrap the system, we initially developed a hard-wired object model editor. About one year into the project we were able to configure end-user tools for accessing the object model. Now we are enhancing this application to support the functionality offered by the new persistency component.

While access to in- /out-baskets is still managed through a dedicated application, baskets are defined through regular forms. We applied a bit of white-box re-use to make the query editor fit in the property editors scheme, and we modeled the basket definition explicitly. Its properties include a query expression specifying the basket's contents.

Most of our latest tools are configured applications, or are developed with configuration in mind. Examples of the former are the object behavior and action rules applications. The graphical workflow on the other hand is designed as an alternative property editor for use in forms.

Reflection

We are increasingly using existing tools to build new (typically configuration) tools. Reflection [Foo96] is a useful technique to support this kind of approach. Reflection, however, is typically considered too obscure and difficult, and is often associated with the run-time behavior of applications. Yet, given the framework approach, this idea of bootstrapping is very natural: the system already contains so much tools to build applications, that we would usually take a step back by not trying to re-use this functionality. The locality of change that can be achieved with reflection is also a very important aspect.

6. Framework evolution

We started the framework initially building three applications. At this moment we have built and maintain many more. We tuned, adapted and extended our framework and business model, based on what we learned from the needs and shortcomings of our applications and from feed-back from configurators and end-users.

Since the business objects, object model, script and application specifications are all stored in the same repository we use it as a sort of central knowledge base. Some working procedures and processes can be recognized as recurring patterns and are formally modeled using status and relations in the repository. Other informal business practices are less clearly recurring at the start and often come to surface, appearing in custom scripts. When we identify these patterns, we try to make them more explicit either in the repository or in the tools. In several places, we needed

to list objects in an hierarchical way. We extended the overview lists with a hierarchical display option.

When developing applications with the framework we identify:

- **Working practices specific to the organization**. We create re-usable assets in the repository, re-factor the object model, or we develop more specific end-user and configuration applications using the existing tools.
- **Practices which transcend applications**. We change the framework functionality, and re-factor the framework [Foo95,Rob96].
- **Functionality specific to a particular application**. Depending on its nature, we change the functionality of the repository, the framework or both. For instance, some applications require a simple procedure to automatically generate standard reply letters based on repository information. In this case, we store extra behavior in the repository. If a new external tool is required, we usually add a component to the framework.
- **Technological needs and opportunities**. We use sub-frameworks for components that are strongly coupled to technology, such as the persistency layer and the document storage. These sub-frameworks can easily be replaced.
- **Additional functionality**, to support new types of applications, e.g. Internet applications.

This way, the framework explicitly supports evolution and re-use at different levels [Til96].

7 .Conclusion

The main goal of the framework design was a small kernel of generic components acting dynamically upon the repository. Hard-coding was to be avoided as much as possible. Initially, we focused on achieving this goal for end-user applications. We developed hard-wired administration and configuration tools to help us bootstrap the system. As the framework evolved, and the tools became more flexible, we re-used components of end-user tools in some of our administration tools. In a third phase, we started to replace some of the hard-wired tools with applications configured in the system.

The framework configuration functionality will be enhanced, in order to further increase the expressiveness of the tools, and to allow end-users to adapt the tools to their own needs even better, thus obviating the need for developers to a larger degree.

Some additional components are being developed, most notably to make the repository Internet-aware. The main component will act both as a client of the repository and as a generic Internet application server. This will empower Argo to develop Internet applications through modeling and configuration too.

At first sight, one might wonder whether this approach does not make the design more complex, or affects performance negatively. We do not feel that the design is more complex. In fact, the framework goals help us focus more clearly on the responsibilities of the various components, in particular with regards to re-use and evolution. As is often the case, reifying implicit responsibilities actually makes the design simpler. And the bootstrapping principle is an additional asset in validating the design.

Although we initially encountered some performance problems, these were not essentially related to the approach as such, but to the use of a persistency component which was not really suited for our purposes. We must be aware, however, that the flexibility of the system allows end-users to build e.g. equally good and bad queries. Proper training is of primary importance, as is the need to hide more complex elements of the tools from novice users.

After three years, we are convinced that the approach is a valid one. End-user applications can be developed iterative and incrementally, even interactively. We do not code (apart from some custom scripting) and we do not generate end-user applications. Neither do we use or need throw-away prototypes. Instead, we build increasingly complete specifications of end-user applications. These specifications are available for immediate execution. In this way, we help close the gap between specification, development and use of the applications. This way we can afford to stimulate the users to think and rethink their processes and redesign them.

References

[Dev96] Martine Devos and Michel Tilman, Design and Implementation of a Business Modeling Framework using Smalltalk, Object Technology'96, 1996

[Foo95] Brian Foote and William F. Opdyke, Lifecycle and Refactoring Patterns that Support Evolution and Reuse, Pattern Languages of Program Design, Addison-Wesley, 1995

[Foo96] Brian Foote and Joseph Yoder, Evolution, Architecture, and Metamorphosis, Pattern Languages of Program Design, Addison-Wesley, 1996

[Flo93] Fernando Flores, Michael Graves, Brad Hartfield and Terry Winograd, Computer Systems and the Design of Organizational Interaction, Readings in Groupware and Computer-Supported Cooperative Work, Morgan Kaufmann, 1993

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995

[Gra94] Ian Graham, Object Oriented Methods, Addison-Wesley, 1994

[Häg89] S. Hägglund, Iterative Design and Adaptive Maintenance of Knowledge-Based Office Systems, Proceedings of the IFIP WG 8.4 Working Conference on Office Information Systems: The Design Process, 1988, North-Holland

[Kar90] B. Karbe, N. Ramsperger and P. Weiss, Support for Cooperative Work by Electronic Circulation Folders, Proceedings of the ACM OIS'90 Conference, 1990

[Nod91] M.H. Nodine, A.H. Skarra, S.B. Zdonik, Synchronization and Recovery in Cooperative Transactions, Implementing Persistent Object Bases, Morgan Kaufmann, 1991

[Ric92] J. Richardson, P. Schwarz, L.-F. Cabrera, CACL: Efficient Fine-Grained Protection for Objects, Proceedings of the ACM OOPSLA'92 Conference, 1992

[Rob96] Don Roberts and Ralph Johnson, Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks http://st-www.cs.uiuc.edu /users/droberts/evolve.html

[Sto94] Frank Stowell and Duane West, Client-Led Design, A systemic approach to Information System Definition, McGraw-Hill, 1994

[Til96] Michel Tilman and Martine Devos, Object-Orientation and Evolutionary Software Engineering, Position paper for the OOPSLA'96 Workshop on Object-Oriented Software Evolution and Reengineering, 1996

[Xer96] Xerox Parc Aspect-Oriented Programming Project, A position paper on Aspect-Oriented Programming, position paper for the ACM Workshop on Strategic Directions in Computing Research, Working Group Object-Oriented Programming, MIT, June 14-15 1996.

Glossary

Action rules: Rules triggered by events that conditionally perform some action. Action rules are used to add extra semantics (business rules) or to add extra functionality.

Authorization rules: Access control relies on a rule-base of authorization rules.

End-user objects: Represent the specific business model.

Indexcards: Forms.

In- / out baskets: Baskets maintain lists of incoming and outgoing work.

Meta-model: Describes structure and constraints of an object model.

Object model: Describes objects, associations and constraints.

Object store: The persistency component.

OCR: Optical character recognition. The process of extracting text out of e.g. scanned documents.

POP: Post Office Protocol. Protocol for accessing a mail server.

Properties: Describe attributes and associations (references to another object) of an object.

Property expressions: Describe chains of properties, by following associations.

Query expressions: Represent in an abstract way queries to be executed. A query (expression) can be stored and re-used.

SMTP: Simple Mail Transfer Protocol. Protocol used for sending Internet mail.

System objects: Represent objects in the repository required for proper functioning of the framework.

Thesaurus: Indexing vocabulary having its own semantically meaningful structure, such as a generic / specific hierarchy and synonyms.

Acknowledgments

We would like to thank the people in our team: Rudy Breedenraedt, Hilde Deleu, Jan Geysen, Els Goossens, Wouter Roose, Fred Spiessens, Du Thanh-Son, Danny Ureel and Leen Van Riel. We would also like to thank the Programming Technology Lab at the Vrije Universiteit Brussel, for the many good ideas. Finally, we would like to thank the pioneer-users at Argo who helped us make this framework happen.