

# A Framework for Adaptive Design

Martine Devos (mdevos@argo.be), Michel Tilman (mtilman@argo.be), Fred Spiessens (fred.spiessens@argo.be)

Argo, Belliardstraat 12, 1040 Brussels, Belgium  
Tel: 32-2-5051931  
Fax: 32-2-5051930

## Abstract

Organization thinking is coming to accept change as a constant force, and is learning how to deal with it. The software development process has yet to adapt to this new perspective. The result of its effort is often not what the customer expected or is outdated before it has had a chance to produce its expected return on investments.

This paper presents our framework, supported by and giving support to an evolutionary software development approach.

The framework uses a repository to capture both formal and informal knowledge of the business model and of personal and shared work practices. High-level end-user and administration tools consult the repository at run-time, querying the meta-model for dynamic behavior. Changes to the repository can be made at-runtime and are immediately available to clients.

End-user applications can be developed interactively and incrementally. They are not hard-coded, neither are they generated. Instead, we build increasingly complete specifications of end-user applications that can be executed immediately.

**Keywords:** Adaptive design, Evolutionary software development, Frameworks, Patterns, Reflection, Turbulence

## 1. Introduction

"When we decided to introduce computer technology to support our business processes, we were facing some serious challenges. A major re-organization was pending, but when and how was not known. Our end-user had little, if any, knowledge of how computer technology could support his or her work, even less how it could affect current work practices or lead to new insights about the way to do work more effectively. Another unknown was the fast-pacing technology itself, in particular framework technology and the Internet."

This statement is not unique. It could have been made by any number of organizations. What is new, however, is that organizational thinking is coming to accept change and uncertainty as they are: facts of life. If the expression of the learning organization has any meaning at all, it requires firms to confront turbulence rather than avoid it, to absorb uncertainty rather than reduce it [Boi95].

Software engineering however, still makes the unreasonable demand for system requirements to be fully specified in advance. Too often it views organizational objectives and work practices as given: things that can be cleanly objectified and solidified.

Software development can no longer afford to ignore change or deal with it as an afterthought. It requires a new approach, which draws on right brain thinking as well as on advances in software technology. This new perspective affects the development process as much as it affects the software architecture.

## 2. Challenges and related work

### 2.1 Diversity

Each organization is unique. Hence the analysis results of one particular organization cannot simply be treated as a model for another [Sto94], neither do the results remain valid when the organization evolves. Even in one organization support for divergent thinking is needed.

Supporting a diversity of views will generally serve as a corrective to the obstructive over-automation, precisely because such support is predicated on a requirement to minimize the prescription of how procedures should be performed or how data may be viewed and combined. Of course, such support involves a tension between providing flexible support for the individual user and retaining the integrity of data and enforcement of procedures that organizations need for legal, security and other reasons [Cro96].

There can never be a once-and-for-all model of a business organization [Sto94].

### 2.2 Turbulence and user-led dissemination

Self-organization and self-managing teams are likely to become the norm in a learning environment.

A new system must first establish itself in a limited region and then invade the whole organization. In these smaller entities local creativity can bloom, and weaker signals get a better chance to be strengthened and to lead to renewal. In a larger bureaucracy, these signals would be stabilized away in 'competition' with legislation or a 'this-is-how-we-do-things' mentality [Bro92].

New systems need the flexibility and diversity to support dissemination of new practices from small kernels and teams to the larger organization.

### 2.3 Interactive process-oriented adaptive design

Traditional requirements gathering techniques are mainly product-oriented. People like to believe that systems that match their specifications as is are a perfect support for their job.

Design documents are often ineffective vehicles for communicating the customer's vision of how the system should work [Cop95].

In recent years, more emphasis has been placed on approaches which try to capture working processes, and increase user involvement, e.g. through scenarios (Use Cases) and role playing games (CRC). System envisioning workshops help the user to find out what the impact of technology can be through future perfect thinking [Hoh97], opportunity scouting, use of metaphor, storytelling and other creative techniques.

'Soft' system thinkers are still concerned with problem identification and solution, but they place increasing emphasis upon the process of problem appreciation. This process of appreciation can best be facilitated by providing a framework for organizational inquiry and learning [Sto94].

All successful systems, including business, go through a process of learning better rules. An organization needs the freedom to experiment. A system needs a cross-over process by which new rules can be learned and the ability to unlearn, to let go of weak rules so it does not get frozen by too many rules [Mai96].

### 2.4 Frameworks

Frameworks are appropriate vehicles for designing re-usable large-scale software. The past years has seen a lot of research activity on how to increase re-use and quality of design through techniques as (re)factoring [Foo95], reflection [Foo96], how to support design and documentation through patterns [Gam95], how to support the development process through re-use contracts [Ste96] and how frameworks evolve [Rob96].

Yet few results show how to capture the simple observation that what may be a stable model or piece of software in one organization, may evolve very soon or even continuously within another one. Likewise some components can be re-usable across different organizations, while other ones may be highly re-usable within one specific organization only. Acknowledging this diversity and differentiating the framework architecture according to this perspective is an important asset to get the required flexibility [Dev96].

## 3. Framework

### 3.1 Context

Work on the framework started 3 years ago at Argo, a semi-government organization that manages the public schools (non-denominational) within the Flemish

community in Belgium. Argo consists of a central administration and a few hundred semi-autonomous local sites.

Three pilot applications, each one emphasizing different aspects, were developed, providing the initial input for the framework design. Since then, several new applications have been developed. This allows us to continuously refine, extend and re-design the framework. Since the start of the project, the business model has been revised due to internal re-organization and change of legislation. Yet no additional coding was required, only re-configuration.

In the near future schools will have access to the central repository through the Internet. Internet applications are configured in a similar way to applications at the central site.

#### 3.1.1 End-users

Not only did users give feedback to the development team, some of them actively took on the role to disseminate their own knowledge and experience towards a larger audience within the organization.

A mail registration and follow-up application has demonstrated the effectiveness of this client-led approach, starting with a small group of key-users, pulling each other along, setting up a larger user-group, editing and publishing a small newsletter and assisting newcomers in tailoring the application.

### 3.2 Framework goals

Our approach aims to combine the high-level modeling power of CASE-tools with the evolutionary nature of object-oriented frameworks. The framework design is driven by the following goals:

- The framework must allow us to model different kinds of organizational, data and workflow process structures, rules and work practices, in order to capture an organization's unique requirements. End-users and teams must be able to adapt the tools to their own skills and needs to allow for divergent team behavior, without violating overall consistency.
- The framework must allow us to develop end-user applications through modeling and configuration, rather than through coding. Application development must support an incremental process of adaptive design.
- To support changing needs, as few assumptions as possible about a particular business model must be hard-coded. Instead, knowledge about the model must be dynamically available from a central repository. This allows the business model to evolve, with limited impact on the application logic and vice versa.

Another important goal of the framework is to support a bootstrapping process, whereby administration and configuration tools are gradually expressed in terms of the

framework itself. This process must lead to a small, but powerful kernel of generic tools.

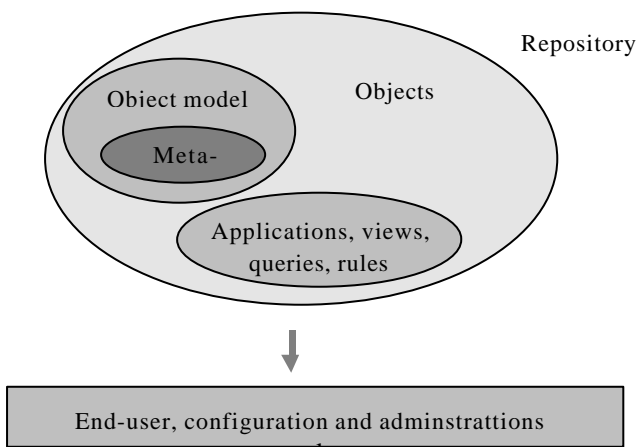
### 3.3 Approach

The framework uses a repository to capture knowledge of organization structure, roles, data, documents, processes, applications, rules and personal and shared work practices. Part of this knowledge is explicitly modeled in the object model, other work practices are less formally specified, e.g. through custom scripts.

To model, configure, tailor and manage the repository, we provide the user with high-level tools:

- End-user tools allow users to select applications, to enter and view data, to query the repository, to display, print and export the results of a query, to manage electronic documents, to access the thesaurus and to manage processes and task assignments.
- Configuration and administration tools allow users to define private and shared views, to store queries for later use, to edit the object model, to set up processes, and to define authorization rules and action rules.

These tools have no hard-coded knowledge of a particular business model. They consult the repository at run-time, querying the model for dynamic behavior. Changes made at runtime are immediately available to clients.



To support this behavior, the repository contains both object- and meta-level knowledge. The meta-model, used to describe the actual object model, can be expressed in terms of itself and is stored in the repository too.

Example: The forms tool uses the following information in the repository:

- the object model
- the application environment
- the available views
- authorization and action rules.

### 3.4 Building applications

Starting from the model existing in the repository at a certain moment, building an application consists of a set of steps. These steps can be performed interactively, in any order. No coding is required (except to add specific custom behavior), neither are applications generated. This allows the development process to unfold incrementally, and to try out and correct ideas dynamically.

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

- **Extend or refine the object model.** If new structures or modifications are required, we update the model with the necessary object types and association types and global constraints. If necessary, we add behavior. The object model, shared by all applications, provides the necessary cohesion. We store this object model in the repository to make it dynamically available for the tools. This allows us to set up arbitrary business models without re-coding tools or applications.
- **Set up application environments** This step starts with defining a view on the shared business model: objects and properties to be created and queried. Once this basic environment has been stored in the repository and access privileges have been set, it is available for immediate use. The user can login, choose this environment, enter data, query the repository, list or print results. He can keep track of task assignments, manage electronic documents and thesaurus. At this point functionality can be added to or removed from the environment and it can be further refined for individual or shared use through views, queries and action rules.
- **Extend or refine authorization rules.** Authorization rules are not tied to any specific model and can be used to set up both very fine- and course-grained access. Some rules may apply to everyone, others are only relevant to a group of people whose members are determined at run-time. This way, authorization rules support cohesion, while allowing considerable flexibility in supporting different team cultures.
- **Define action rules.** These rules capture business- semantics, set defaults and perform extra validation, filter information. Action rules can cover an entire organization, particular functions or they can be limited to a specific application environment. This way, they can give extensive support to autonomous teams while preserving overall consistency.

### 3.5 Development process

When developing applications with the framework we identify

- Practices specific to the organization. We create re-usable assets in the repository, re-factor the object

model, or we configure more specific end-user tools or administration tools using the existing tools.

- Practices which transcend applications. We change the framework functionality, and re-factor the framework.
- Specific functionality to a particular application. Depending on its nature, we change the functionality of the repository, the framework or both.

Example: Some applications require a simple procedure to generate letters based on repository information. In this case, we store extra behavior in the repository. If some additional external tool is required, we will 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 lower persistency layer, document storage and E-mail. These sub-frameworks can easily be replaced.

This way, the framework explicitly supports evolution and re-use at different levels [Til96]. As meta-model and tools are part of our framework, they evolve as well to capture additional generic functionality, e.g. to formalize new types of constraints.

### 3.6 Implementation

The framework is developed in VisualWorks\Smalltalk in a Windows / Novell / Oracle environment. It relies on some external components, such as a Kofax scanning and rendering library, a Calera optical character recognition-module, a Mires full-text-indexing engine and HP jukebox technology.

### 3.7 Detailed component overview

#### 3.7.1 Object model

The object model describes object structures, associations and constraints. Objects, behavior, model and meta-model are stored in the central repository and are used by configuration- and end user tools. Tools query the model at run-time.

The objects stored in the repository are classified as follows (see figure):

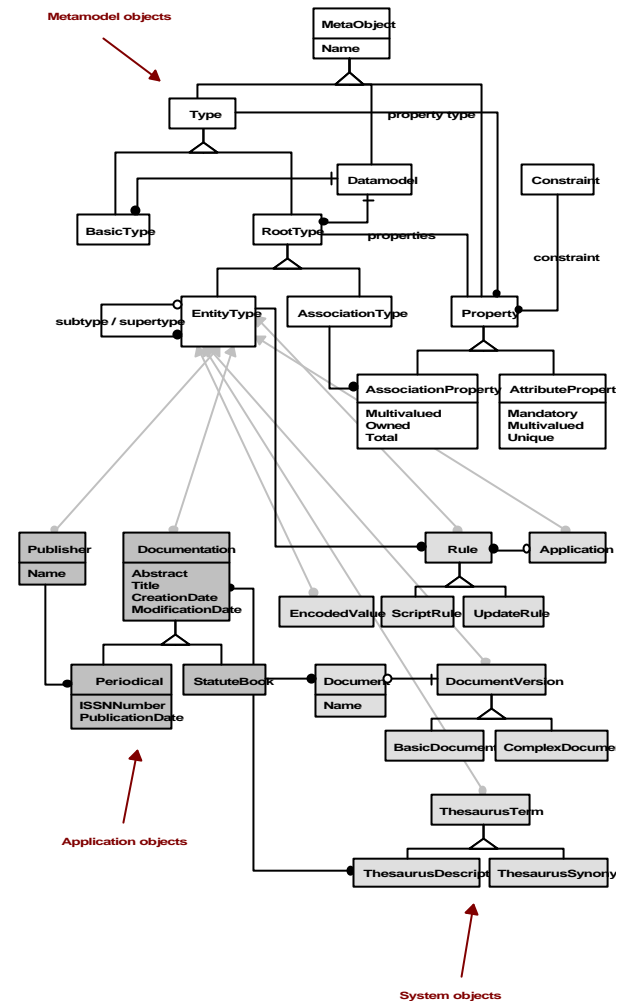
- the meta-model
- system objects that are needed for the proper functioning of the framework, such as login users, business and authorization rules, stored queries and electronic documents
- end-user objects, making up the actual business model.

##### 3.7.1.1 Meta-model

Each object has a specific type and consists of zero or more typed properties. Object types can have subtypes.

Attribute properties have primitive types, such as date, string or number, and enumerated values over these types. Association properties reference other objects. Properties

have several attributes denoting mandatory, uniqueness, arity and symmetry constraints.



Simplified OMT schema, showing some object types and associations

Our model supports n-ary and attributed associations. N-ary associations are useful abstractions, in particular when we need to integrate existing databases into our object model. Dynamically-typed associations can easily be modeled by enumerated attributes.

Properties can be constrained to take values in a subset of the type value set. Constraints are specified by means of query expressions or Smalltalk expressions. Property type or constraint can be overridden in subtypes.

As an abstraction mechanism to hide details of the object model, we use virtual properties. Virtual properties are read-only properties representing values or aggregates of properties. Values can be calculated by means of a query or Smalltalk expression. Virtual properties can be used as regular properties.

Example: Virtual properties are used to avoid the common problem that definition of complex queries requires

exposure of the user to the complete details of the object model.

As the meta-level is explicitly modeled, no dedicated CASE tool is required. We use the forms tool to edit the object model. Authorization and action rules can apply to the object model as well.

Example: Rules can be defined that are triggered whenever the model changes, taking appropriate actions to update the database tables and columns. As all object types are instances of the same meta-object type, conditions can be used to differentiate rules acting on different object types.

### **3.7.1.2 End-user objects**

End-user objects model organization structure, processes, tasks, data and relationships. The tools make no a priori assumptions about the end user objects. This allows construction of flexible business models.

Instead of using a separate object type for each kind of organizational unit, like department, workgroup, sub-department, we use generic units in our applications. This allows to change or replace the original organization model by a new one without having to define new object types. A mere change of repository population suffices.

We have the flexibility to choose either approach, because we can define authorization and action rules in terms of the model and in terms of its population.

### **3.7.1.3 System objects**

For proper functioning of the tools, system objects are needed: views, queries, authorization and action rules, thesaurus elements and electronic documents. To increase uniformity, we model these objects explicitly and store them in the repository.

These objects are usually presented to the end-user through special tools, such as a dedicated tool to define queries, and viewers for scanned documents. But since system objects are explicitly modeled and stored in the repository, for administration purposes as reporting, we don't need additional tools. The generic end-user tools can be used.

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

### **3.7.1.4 Object behavior**

Repository object types are mapped onto regular Smalltalk classes. The meta-model and system objects need minimal behavior to support the framework. For these object types Smalltalk classes are permanently available in the image.

Lightweight classes and accessor methods are generated on the fly for the object types in the active application, and cached. Additional repository methods are installed into the corresponding method dictionaries. This ensures a dynamic system, running at maximum performance, while minimizing space overhead.

Not all object types are completely modeled. The internal structure of queries, for instance, is too complex to be modeled explicitly, neither is it necessary. These objects are stored as encoded, anonymous data.

### **3.7.1.5 Object substitutes**

Object substitutes provide lazy access to repository objects. They are defined as chains of association properties starting from a particular object. This is especially useful when the last property in the chain is multi-valued.

Example: The expression "department.members" (the dot refers to properties), when applied to an employee, can be treated as the set of members working in the same department as the employee. Elements may be added to or removed from this collection. Changes will be propagated to the database.

We use a similar technique to query the repository.

Example: the expression

*Employee select: [:each | each salary > 10.000 ]*

uses substitutes to translate this high-level expression into a query expression.

### **Future work**

*Dependencies between objects, for instance to enforce cascaded deletes, are expressed as action rules. We will formalize common practices as they are discovered.*

### **3.7.2 Application environments**

Application environments allow to restrict the view on the repository, by specifying what objects can be accessed, queried or created.

Groups of functions, such as scanning and indexing of documents, or printing reports, can be defined. The functionality in the environment can be specified by assembling the appropriate functions required for a given task.

For every object type, views can be configured for use in list-, print-, export- and forms tools. These views can either be 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, private or shared queries may be saved. 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, for instance to set appropriate defaults or to perform some extra functions, for instance batch update of a list of objects.

Private or shared task lists keep track of job assignments. Specific types of task lists can be defined by means of queries.

Environments can be structured hierarchically. Nested environments further specialize the view on the shared object model. Action rules, views and queries are inherited.

### Future work

*The need to assemble application functionality out of groups of functions has become increasingly manifest during project implementations. We will elaborate the interactions with other components of the framework in the future.*

### 3.7.3 Authorizations

The authorization mechanism makes no a priori assumption 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 rule base of authorization rules.

#### 3.7.3.1 Authorization rules

Each authorization rule R 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), for instance a selection of the object properties
- the rights (R), such as read, write, add, remove, print.

These sets can be dynamic, based on a condition specifying the elements that belong to the set. Conditions make use of query expressions defined with end user tools, or, if necessary, 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 set). A rule R is restrictive if at least one of its sets is restrictive. A rule R can be *granting*. Rules can be both granting and restrictive. Access control must be explicitly granted or denied.

#### 3.7.3.2 Rule semantics

The following example illustrates the semantics of authorization rules.

Applying a rule R {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 definition:

- true if (ag ∈ Ag) and (o ∈ O) and (as ∈ As) and (r ∈ R) and (R is granting)
- false if (ag ∈ Ag) and (o ∉ O) and (as ∉ As) and (r ∈ 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”. Note that this rule is not relevant for objects of types other than proposals.

#### 3.7.3.3 Combining rules

Rules can be inherited according 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 rules can be undecided for a particular tuple {ag, o, as, r}, at least one rule is needed, which answers true or false for all tuples. This can be a rule defined at the common supertype of all object types.

#### 3.7.3.4 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 as document annotations, applications or even authorization rules.

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 was configured. This application maintains explicit associations between users and environments. Several authorization rules have been reduced to one meta-rule checking this information. This rule applies to the application itself.

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

- Configuration and management using the end user tools for simple authorization rule types. Transformation methods are needed to convert the rules into a basic-level rule.

These techniques allow to configure specific authorization applications, targeted towards a particular use.

#### 3.7.3.5 Optimizing performance

By default authorization rules are executed locally. In pathological cases, this generates considerable overhead when many objects are retrieved from the repository, only to find out that the user has no access. To optimize performance, rules are combined with database queries if appropriate.

### 3.7.3.6 Design

The authorization mechanism presents one aspect [Xer96] of the framework. 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. However, this requires context-sensitive access control for objects [Ric92], whereby only trusted components may access the object store directly.

### 3.7.4 Action rules

Objects have constraints defined in the model and have behavior that is used across all applications. End-user applications sharing these objects may however require additional functionality or even additional semantics (business rules) [Gra94]. Action rules capture these requirements.

While action rules allow semantics to be different across applications or processes, they do not violate the global constraints as defined in the model.

The main elements of action rules are events, conditions and actions.

#### 3.7.4.1 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.

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

#### 3.7.4.2 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.

#### 3.7.4.3 Script rules

Script rule condition and script rule action expressions require Smalltalk code.

#### 3.7.4.4 High-level rules

To allow end-users to add their own rules, high-level rule types are provided.

High-level rules formalize common script practices in the current applications.

Conditions are defined using query expressions. The action part uses one or more object templates to specify new values for object properties. The first template refers to the receiver. Additional templates specify type and initial values for new objects to be created when the rule is activated.

Rules with one template 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 to set up a batch procedure to update a series of objects.

Rules with two templates are used to support workflow processes. Given a task (the receiver), a rule can be defined to create a follow-up task. This allows to set up 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.

##### 3.7.4.4.1 Object templates

Object templates can be edited by means of extended forms. No scripting is needed. These templates behave similar to regular objects, but their properties can accept expressions in addition to values.

The following expressions (see example) can be used:

- constant expressions, depending on the type of the property, such as a date value (attribute property) or a reference to another object (association property)
- query expressions
- property expressions, relative to the rules templates
- message sends, using typed methods, with receiver one of the rule templates.

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

Example: the following approval rule (in pseudo-language, the dot referring to properties) illustrates the idea:

```

event          = select 'forward' menu item
condition      = currentTask.approved = true
current task template (type ApprovalTask)
    currentTask.status := finished
new task template (type ApprovalTask)
    newTask.addressee := Manager select [: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 the request sent to him and when he selects the ‘forward’ menu item, the current task will be flagged as finished. A new request for approval will be sent to all managers in Finances. The documents enclosed in the current task will be attached and the date will be set to today’s date.”.

### 3.7.4.5 Modeling different process models

Different workflow process models can be implemented. We illustrate two 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 containing the routing information 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 action rules

#### Future work

*Rules and events are explicitly modeled. The end-user tools have been used to configure an application for managing and editing rules. As the configuration of this application itself requires rules, we use a limited form of manual bootstrapping.*

*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 one-template high-level rule can be used to specify these related objects.*

### 3.7.5 Repository

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

#### 3.7.5.1 Bridge-pattern

The bridge actually uses two-layers. The uppermost bridge maps Smalltalk objects and abstract query expressions on the abstract database layer. In the case of a relational database this will describe abstract columns and tables. The lowermost bridge maps abstract database operations on the physical layer.

This two-level schema allows to change the physical database without impacting the upper bridge. The lower bridge may have to implement abstract operations that can not always be mapped directly onto the physical database, such as recursive queries in the case of relational

databases. Changing the type of database, e.g. object-oriented instead of relational, can be dealt with in the upper layer.

#### 3.7.5.2 Mapping strategy

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 (separate tables or separate columns for each enumeration value).

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

#### 3.7.5.3 Query expressions

Query expressions allow users to build abstract queries, with the inclusion of relational operations and subqueries. Queries allow to retrieve instances of a given object type (possibly including all its subtypes) satisfying constraints over attribute and association properties. Query expressions can be specified using the following pseudo-language:

```

query expression = objectType expression
expression = simpleExpression | constExpression |
expression “and” expression | expression “or” expression |
“not (” expression “)”
constExpression = “true” | “false” | integerExpression
dateExpression | objectExpression
simpleExpression = variable operator value
propertyExpression = property {“.” property}
value = propertyExpression | constant | expression |
contextVariable.

```

Operators include recursively-defined operators and membership of subqueries. Property expressions are either properties of the object type being queried, or chains of properties. In addition, context variables can be used (such as the current date or the login user).

#### 3.7.5.4 *Selective caching*

Access to specific objects in the repository is often required on a more permanent base. Query expressions define which objects must be prefetched. The scope can be limited to an application, to the login-session (for instance the authorization rules) or even persistent across sessions. A consistency mechanism using timestamps updates that part of the cache that has been invalidated.

#### 3.7.5.5 *Locking*

Locking relies on persistent locks, which is better suited towards highly interactive applications and remote (off-line) use.

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.

#### 3.7.5.6 *Transactions*

Traditional transactional systems are too much targeted towards a programmatic use. Notifications and dirty reads are essentially building blocks that leave the responsibility up to the programmer.

To better support interactive, multi-window applications, the framework provides transactions that can be hierarchically nested. These subtransactions need not be executed strictly sequentially.

A basic premise is the possibility to commit partial results of s [Nod91]. If a subtransaction fails, the state will be retained. Committing the main transaction later on will try to commit previously suspended transactions.

### 4. **Future work**

The main goal of the framework 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 completely replace some of the hard-wired tools with applications configured in the system, which has turned out to be relatively easy for the object model editor. Using the existing tools to model and edit new types of action rules, to implement an evaluator and to connect the evaluator with the existing components proves to be more of a challenge.

Reflection is a useful technique to support this kind of approach. Reflection is however typically considered too obscure and difficult, and it 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 functionality to build

applications, that we step back by not trying to re-use this functionality.

We will explore the use of reflection in more detail in the future. The locality of change that can be achieved with reflection is a very important aspect.

### 5. **Conclusion**

After three years, we are convinced that the approach is a valid one. End-user applications can be developed interactively and incrementally. 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 a sense, the framework and its built-in tools act like an interpreter for the specifications.

In this way, we help close the gap between specification, development and use of the applications.

### 6. **Acknowledgments**

We would like to thank the people in our team: Rudy Breedendraedt, Hilde Deleu, Els Goossens, Du Thanh-Son, Danny Ureel. 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.

### 7. **References**

- [Boi95] Max Boisot, Preparing for turbulence: the changing relationship between strategy and management development in the learning organization, Developing Strategic Thought, Bob Garratt ed., McGraw-Hill, 1995
- [Bro92] Gerrit Broekstra, Chaossystemen als metafoor voor zelfvernieuwing van organisaties, Gamma Chaos, Cor van Dijkum en Dorien de Tombe ed., Aramith Uitgevers 1992
- [Cop95] James O. Coplien, A Generative Development Process Pattern Language, Pattern Languages of Program Design, Addison-Wesley, 1995
- [Cro96] Malcolm Crowe, Ricard Beeby, John Gammack, Constructing Systems and Information, McGraw-Hill, 1996
- [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

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

[Hoh97] Luke Hohman, Journey of the Software Professional, Prentice Hall, 1997

[Joh88] Ralph E. Johnson and Brian Foote, Designing reusable classes, Journal of Object-Oriented Programming, 1(2):22-35, June-July 1988

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

[Mai96] Arun Maira and Peter Scott-Morgan, The accelerating organization, McGraw-Hill, 1996

[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>

[Ste96] Patrick Steyaert, Carine Lucas, Kim Mens, Theo D'Hondt, Reuse Contracts: Managing the Evolution of Reusable Assets, Proceedings of the ACM OOPSLA'96 Conference on Programming Systems, Languages and Applications, 1996

[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-Oriented 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

## **Addendum: 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.

### **Bridge-pattern**

Decouples an abstraction from its implementation.

### **Configuration and administration**

Define object model, applications, rules, private or shared views and queries.

### **End-user objects**

Represent the actual business model.

### **Facade-pattern**

Provides a unified interface to a set of interfaces in a subsystem. A facade defines a higher-level interface that makes the subsystem easier to use.

### **Framework**

A framework is a set of cooperating classes that make up a reusable design for a specific class of software [Joh88]. The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control [Gam95].

### **Meta-model**

Describes structure and constraints of an object model.

### **Object model**

Describes objects, associations and constraints.

### **Property**

Describes an attribute or reference to another 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.

### **System objects**

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