

***Dart*: A Meta-Level Object-Oriented Framework for Task-Specific Behavior Modeling by Domain Experts**

Reza Razavi¹, Jean-François Perrot², Ralph Johnson³

¹University of Luxembourg – FSTC
LUXEMBOURG
razavi@acm.org

²Université Pierre et Marie Curie – CNRS – LIP6
Paris – FRANCE
jean-francois.perrot@lip6.fr

³University of Illinois at Urbana Champaign,
Illinois – USA
johnson@cs.uiuc.edu

Abstract

We propose an object-oriented framework for complex behavior modeling by domain experts. It is set in the context of *Adaptive Object-Models* and *Flow-Independent* architectures. It is an evolution of Dragos Manolescu's *Micro-Workflow* architecture. We add to it several abstractions needed to reify a number of concerns common to modeling by domain experts. Our aim is to make it easier for domain specialists to write behavior models in their own terms, using sophisticated interfaces built on top of the framework.

1 Introduction

An increasing number of object-oriented applications that we call Adaptive Object-Models (AOMs) [YJ02, RBYPJ05], integrate a Domain-Specific Modeling Language (DSML) [Tolvanen05] for behavior modeling. This language is dedicated to domain experts and available at run-time. In general AOMs deploy a DSML to cost-effectively and programmer-independently (1) cope with rapid business changes; (2) create a family of similar software; and (3) provide modeling and model operating functionality. More specifically, we focus on Flow-Independent AOMs (FI-AOM). A flow-independent architecture keeps the control flow outside the application domain, and thereby avoids intertwining process logic and application code, which is a hindrance to the software evolutive maintenance. The DSML embedded in an FI-AOM shares many characteristics with workflow languages and architectures [WMC99, LR2000]. They support both defining and executing processes in terms of a coordination of primitive tasks and their distribution between processing entities. What distinguishes such a DSML from a classical workflow language is that the processing entities tend to often be objects and not humans and applications. Both processing entities and primitive tasks belong to the domain's concept and task ontologies. The language targets domain experts. To emphasize these important differences and avoid confusion with standard workflow languages, we propose to call this class of DSMLs *expert languages*.

Both in industrial and academic settings, we have studied and also developed many successful FI-AOMs [AJ98, DT98, Raz00, GLS02, CDRW02, YBJ01]. Unfortunately, these applications required developing a custom expert language. In our opinion, the best approach for creating FI-AOMs is the *Micro-Workflow* architecture by Dragos Manolescu [Manolescu2000, Manolescu2002], hereafter denoted by “MWF”. Several characteristics distinguish MWF from traditional workflow architectures, notably, a lightweight, modular architecture, and dealing with processes in a pure object world, i.e., all workflow processing entities are objects. This feature is crucial when developing expert languages for FI-AOMs.

However, MWF mainly targets developers. Manolescu explains the choice of the activity-based process modeling methodology by the fact that there is resemblance between the modeling abstractions and control structures in structural programming languages. Our goal is to take advantage of the extensibility and modularity of the MWF architecture in order to propose a core process modeling component which targets also business *experts*.

In the following subsections we explain our solution, called *Dart*, based on operating two separations of concerns through refactoring [Opd92], and some other amendments. Dart stands for Dynamic ARTifact-driven class specialization. It provides more flexibility and more desired features for programming by domain experts than MWF, while remaining compatible with it, but is harder to learn. We describe the design of Dart, as well as the reasons behind our design decisions using patterns (figure in *slanted fonts*). We use UML as a standard notation for presenting the *meta-level* abstractions that define Dart, as well their meta-level relationships.

We postpone the description of our motivations and also achievements to section 4. Section 2 is dedicated to the presentation of an example, and section 3 to an overview of the MWF core. Section 4 exposes our solution. Section 5 discusses our results, before concluding in section 6.

2 A simple example

For illustration purposes we adopt from [RTJ05] a simplified version of a banking system for handling customer accounts like checking or savings accounts. The system contains a class called `SavingsAccount` which provides a field called “`interestRate`” that provides the interest rate for the account, as well as other fields that are value objects, like “`Money`” and “`Percentage`”. Each day a method called `accrueDailyInterest` is executed and the interest is added to the account’s balance. The underlying algorithm is illustrated by Figure 1. The computation comprises five *steps*: (1) the current saving account is explicitly designated and named *Account*¹; (2) and (3) the interest rate and the balance for that account are computed (could be done in parallel), and called respectively *Balance* and *Interest Rate*; (4) the daily interest is computed and called *Daily Interest*, and (5) finally, the daily interest is deposited on the selected account. No object is produced in this last step (result called *Void*).

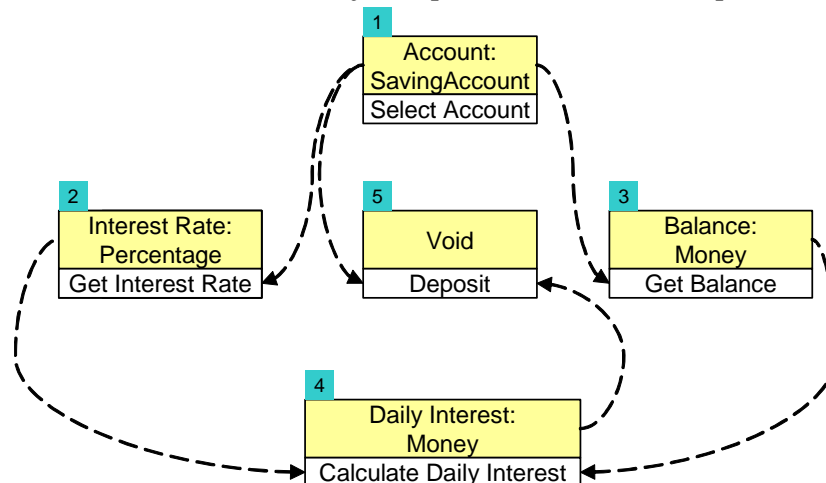


Figure 1: A visual representation of the *Accrue Daily Interest* computation.

The graphical notation for steps corresponds to the association of two rectangles. The lower rectangle (in white) symbolizes the operation and the upper one (in yellow) its result (which

¹ Could be what-ever else; the names are strings with no special semantics from the computation’s point of view.

is a *part* of a whole product). The arrows are directed from parts towards operations, and denote the data dependency between steps. For instance, the arrow from the step 4's part to the step 5's operation denotes the fact that the execution of the step 5's operation requires the availability of the part of the step n° 4. This graphical notation is chosen since it reflects (1) the type of graphical interface that Dart supports (typically a spreadsheet interface), and (2) the cognitive approach of users when modeling by a Dart-based DSML (*grosso modo*, programming is done by relating together domain-specific operations and parts).

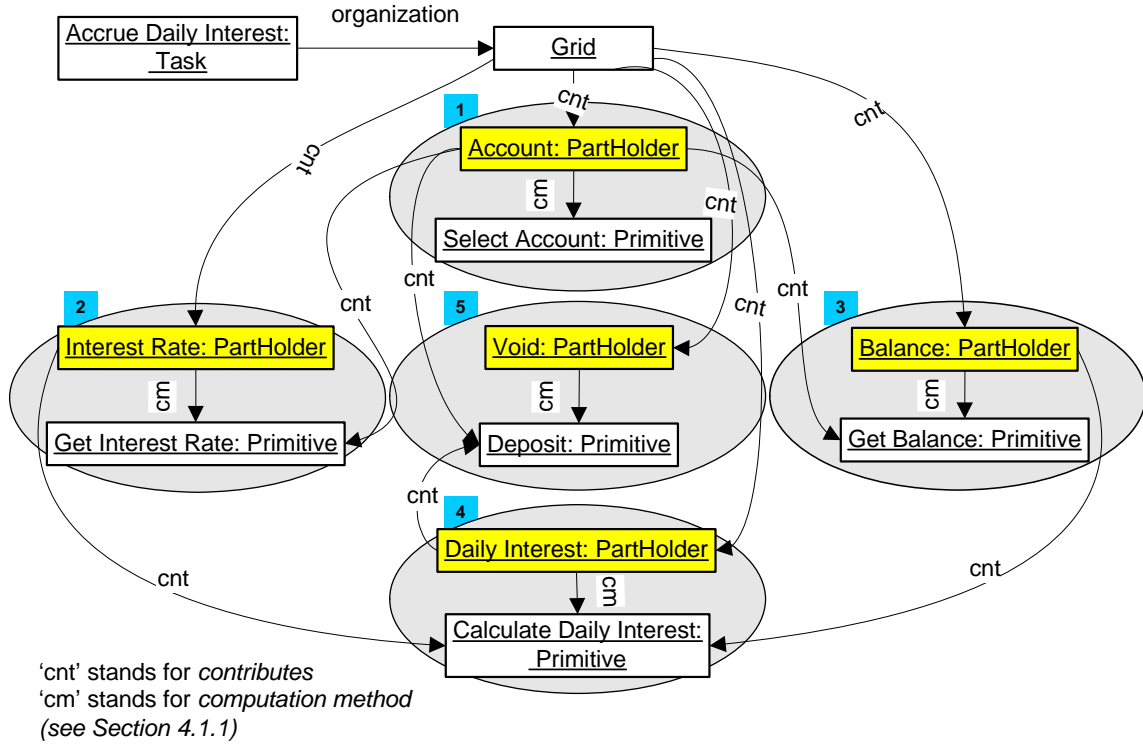


Figure 2: The Dart representation of the example in Figure 1.

The Dart representation of the same algorithm is given by the object diagram of Figure 2. The *type* of objects in this diagram is *Task*, *Grid*, *Part Holder*, and *Primitive*. These are Dart abstractions that we describe in the following sections. A secondary goal of this diagram is to illustrate the resemblance between the internal representation of algorithms by Dart, and their visual rendering on the screen (Figure 1). Figure 3 illustrates the same model represented according to the MWF through abstractions such as *Sequence*, *Ordered Collection*, and *Primitive*. We further describe and compare MWF and Dart abstractions in the following sections.

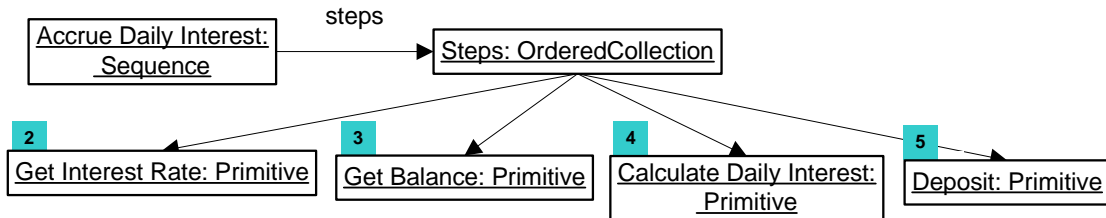


Figure 3: The MWF representation of the example in Figure 1.

3 The Micro-Workflow core

The MWF architecture leverages the object technology to bridge the gap between the type of functionality provided by current workflow systems and the type of workflow functionality required to implement processes within object-oriented applications. At the focal point of the architecture, the MWF core provides functionalities for workflow definition and execution.² This section explains the underlying design based on Manolescu's thesis [Manolescu2000].

3.1 Representation of workflow definitions

A *workflow definition* specifies the activities that the workflow processing entities must perform to achieve a certain goal. From a theoretical point of view, MWF has adopted the *activity-based* process modeling methodology, where workflows are represented in terms of activity nodes and the control flow between them. The whole constitutes a directed graph called *activity network*, which captures how process activities coordinate. This representation places activities in the network nodes and the data passed between activities on the arcs connecting these nodes, showing the data flow between activities [GPW99].

From the framework design point of view, MWF represents the nodes of the activity map corresponding to the process definition with a set of *Procedures* (e.g. Figure 3). The MWF employs several procedure subclasses that together provide a range of procedure types. Our focus here is on core abstractions, i.e., *Procedure*, *PrimitiveProcedure*, and *Sequence* (Figure 4). *PrimitiveProcedure* enables domain objects to perform application-specific work outside the workflow domain.

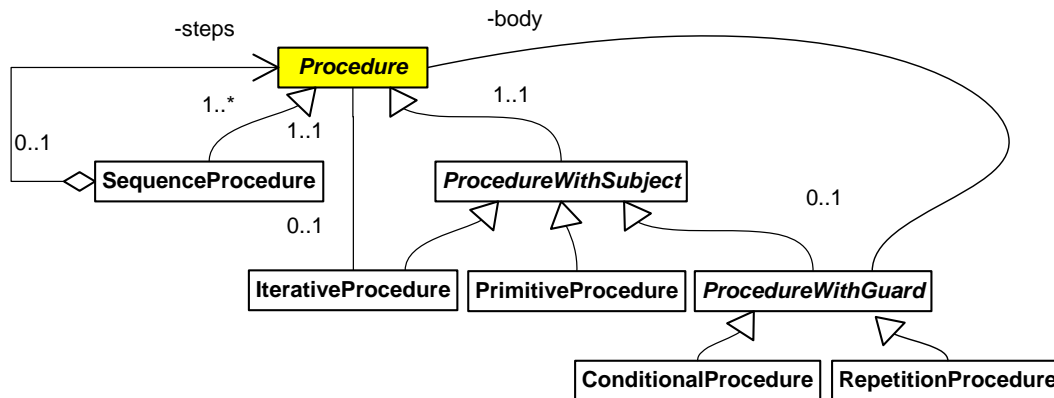


Figure 4: The MWF process component (comprises also Fork and Joint abstractions) [Manolescu2000, page 187]

As illustrated by Figure 5 using the *Smalltalk* language syntax, the implementation language of both the MWF and Dart, a primitive procedure is specified by providing the name of the method to invoke at runtime, the name of the receiver of the message, the name of the arguments, if any, and the name of the result. Names correspond to keys for storing the resulting objects and storing the arguments in the current execution context (a hash table). In this example, the message sent is called `calcDailyInterest:with:`, and the names are respectively called: `balance`, `interestRate`, `myAccount`, and `interest`. *SequenceProcedure* allows developers specifying sequences of activities by aggregating successive procedures. It is a *Procedure* subclass that has a number of steps, each of which is another procedure. *Conditional* and *Repetition* provide a means to alter the control flow. *Iterative* works on composite objects. Finally, *Fork* and *Join* spawn and synchronize multiple threads of control in the workflow domain.

² Other components are added by extension to support history, persistence, monitoring, manual intervention, worklists, and federated workflow.

```
PrimitiveProcedure
  sends: #calcDailyInterest:with:
  with: #(balance interestRate)
  to: #myAccount
  result: #interest.
```

Figure 5: Instantiation of a primitive procedure in MWF.

The (simple) activity graph in Figure 3 is defined by creating a `SequenceProcedure` which holds an ordered collection of four primitive objects.

3.2 Representation of workflow executions

Procedure execution relies on the interplay between a `Procedure` instance and a `ProcedureActivation` instance. There are two ways to trigger the execution of a procedure object. The `execute` message allows clients from the application domain to fire off a procedure. Typically they send this message to the root node of the activity map representing the process definition. The second entry point `continueExecutionOf:` serves the workflow domain. Composite procedures send this message to execute their components. A procedure reacts to the `execute` message by sending itself the `continueExecutionOf:` message. The control reaches the internal entry point. Next the procedure checks its `Precondition` by sending the `waitUntilFulfilledIn:` message with the current activation as the argument. In effect, this message transfers control to the synchronization component. The `waitUntilFulfilledIn:` message returns when the precondition manager determines that the precondition associated with the procedure is fulfilled. Next the procedure creates a new instance of `ProcedureActivation`. Then it transfers control to the new activation by sending the `prepareToSucceed:` message. On the workflow instance side, the activation handles the data flow. First it initializes the local variables from the initial context of its type. The first `forwardDataFlowFrom:` message moves data from the procedure initial context to the activation. Then the new activation extends its context with the contents of the current activation. Finally, it returns control to its `Procedure` object, on the workflow type side. The `ProcedureActivation` is here responsible for managing data flows. At this point, the procedure has all the runtime information and sends the `executeProcedure:` message to complete execution. However, `Procedure` is an abstract class and doesn't implement `computeStateFor:` and `executeProcedure:.` Execution within the `Procedure` class ends here, and each of its concrete subclasses implements these messages in its own way. Thus inheritance allows all procedure types to *share the same execution mechanism*, while polymorphism enables them to augment this mechanism with the behavior specific to each type.

4 Refactoring the Micro-Workflow core

In the following subsections we explain our solution based on operating two separations of concerns, and also some other amendments.

4.1 Separation of structural and semantic aspects

MWF procedures combine simultaneously two important roles. First, they serve as building blocks for constructing the activity graph. Second, they hold information about the semantics of the operation. For instance, the procedure in Figure 3 is constructed by interconnecting primitive and sequence objects. The operational semantics of each step of the procedure is also held by each of these objects. We propose to separate these two roles by applying the *Bridge* [GHJV95] pattern. The result is that each step in the workflow is specified using two distinct abstractions as follows.

4.1.1 Representation of *part holders*

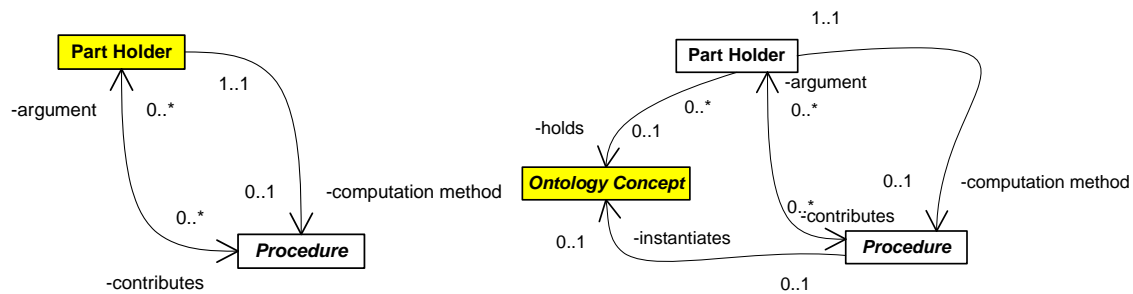


Figure 6: Design of steps in Dart.

Figure 7: How procedures and part holders relate to the ontology.

As it is illustrated by Figure 6, we propose to delegate the structural role of the procedures to an abstraction called *Part Holder*. The specification of a step in a workflow, called *task* in our context (cf. the next subsection), is now achieved by associating a *part holder* ('what') to a *procedure* ('how'). The association is twofold. On the one hand, part holders are related to the (new) procedures by a relation called *computation method*. A part holder is associated to at most one procedure. For instance, as illustrated by Figure 2, step 2 in Figure 1 is represented by associating a part holder named *Interest Rate* with a primitive called *Get Interest Rate*. On the other hand, part holders are related to procedures by a relation called *contributes*. Primitive procedures may in fact require arguments. A part holder may contribute to the computation of a primitive by providing 'its' part. The inverse relation is called *argument*. A part holder may in effect serve as argument to the definition of zero or more procedures. An argument for such a procedure is selected amongst the part holders associated to other steps in the task. For instance, the part holders called *Balance* and *Interest Rate* in Figure 2 contribute to the computation of the primitive called *Calculate Daily Interest* (that computes the value of a part holder of type *Money*, called *Daily Interest*).

Further, for practical reasons it is important to be able to associate to the primitive nodes of a task definition to the object that results from their execution. This can for instance serve when fine-tuning the workflow definition by simulation. Or, when the workflow engine is used like a spreadsheet with two modes: showing the formula associated to a cell or the result of its execution. We therefore add a new abstraction, called *ontology concept*. Figure 7 illustrates how part holders and procedures relate to the *ontology*. Ontology concepts are instantiated by primitive procedures, and held by part holders. The domain ontology provides a specification of the target business product and its parts and their relationships. We assume here that the target system is provided with an explicit representation of the domain ontology, which is crucial for DSMLs in general. Dart allows expressing how a full product can be computationally obtained through partial computation of its parts.

Now, we can explain in more detail the abstractions that underlay the object diagram in Figure 2. Each part holder (yellow rectangle) is connected to a primitive by a link called 'cm' that is an abbreviation for the 'computation method'. Part holders are also connected to the primitives with the 'contributes' link. The association of a part holder and a (primitive) procedure creates a *step*. Furthermore, a grid structure contains the part holder of each step (link called 'cnt' for content). As explained in the next subsection, the grid is an example of organization and visual layout media for the steps of a task.

4.1.2 Representation of tasks

The notion of task refers to a logical grouping of steps, defining a meaningful, but often partial, business procedure. Part holders and procedures already maintain two relationships called computation method and contributes (see the previous subsection). These relationships interconnect steps together. For instance step 4 in Figure 2 is connected to step 5 of the same figure, since the part holder of the former contributes to the procedure of the latter. This implicit organization *de facto* represents a task. However, it is not sufficient for a neat representation of tasks. This issue is addressed by the notion of a task, which allows explicitly organizing steps.

To represent *tasks*, we first apply a variant of the *Composite* pattern [GHJV95] to the design presented in Figure 6. The result is two new abstractions (see Figure 8). The common abstraction is called *Process-Conscious Product*, and the composite is called *Task*. A task aggregates one or more steps by pointing to their part holders. In this design, steps are sequentially ordered (like in MWF). The relationships of part holders with ontology concepts and procedures remain as in Figure 7.

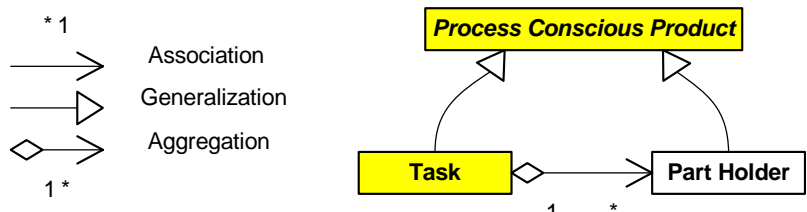


Figure 8: Preliminary representation of *Tasks* in Dart.

This design imposes an overspecification of tasks by sequentially ordering and executing their steps. For optimization and business-related motivations, steps may be organized in different structures. For instance, the steps of the task in the example of Figure 2 can indifferently, from the operational semantics point of view, be visually organized in a list, grid or free shape. Therefore, we modify the design of tasks to separate the two step-organization and step-grouping aspects (see Figure 9). Now, a task aggregates one or more steps by pointing, indirectly, through its *organization* link, to their part holders. The order of steps in a task is by default irrelevant. The full definition of a business procedure is obtained by aggregating a set of task definitions into a *Behavior* definition (see also Figure 9).

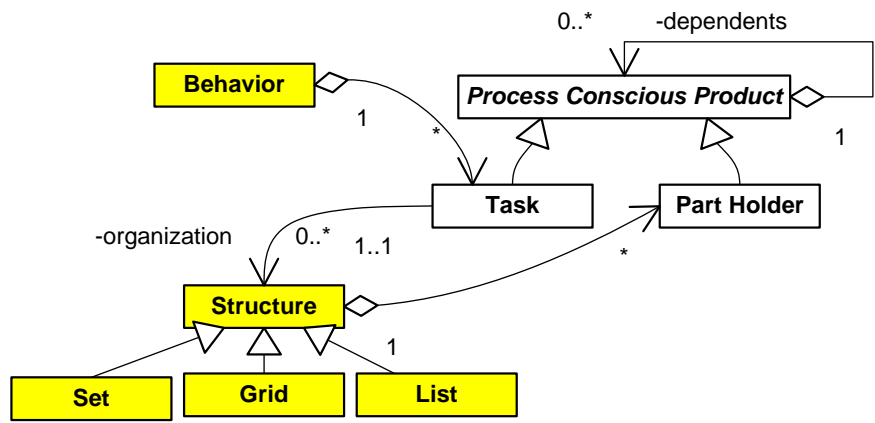


Figure 9: Design of *tasks* in Dart.

4.2 Separation of the computation description from the execution strategy

As explained in Section 3.2, MWF procedures are deeply involved in both (1) the description of the expected computation; and (2) the implementation of the execution technique for that procedure. For instance, the primitive in Figure 5 (1) holds the information about its purpose which is calculating the daily interest; and (2) also implements the rules that govern the realization of that computation (a method invocation). By applying the *Strategy* [GHJV95] patterns, we propose to further split the *semantic* role of the procedures into two distinct roles.

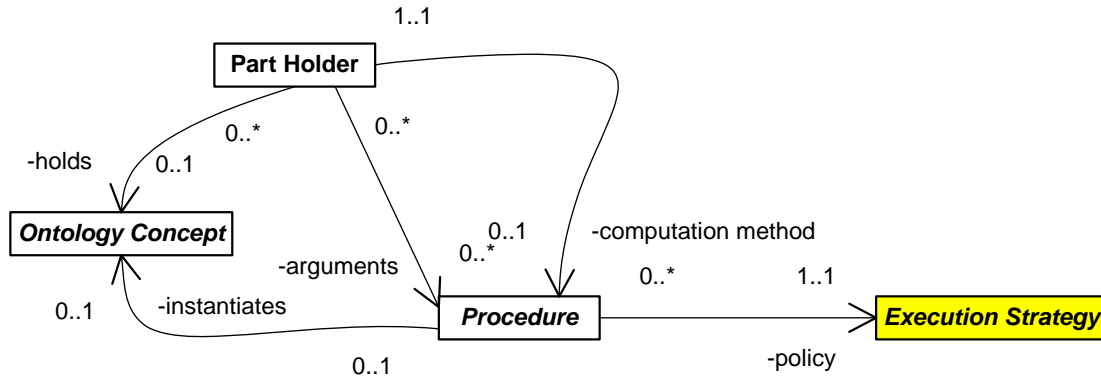


Figure 10: Design of execution strategies in Dart.

A new abstraction called *execution strategy* is added to Dart (see Figure 10). Procedures have now only the role of representing the computation. The operational semantics of the `executeProcedure` method from the MWF changes now to give the control to the execution strategy which is currently associated to the procedure (and can dynamically change). The execution strategies that we have currently identified and implemented are summarized in Table 1. Execution strategies are also associated to tasks. The default behavior consists in launching the execution of task steps taking into account their organization.

Table 1: Different execution strategies currently identified in Dart.

Construct	Execution strategy
Primitive	Invocation of a method with its arguments.
Factory	Invocation of a static method.
Getter / Setter	Invocation of a getter/setter method.
Control Structure	Execution of the pre-condition behavior and accordingly the action behavior.
In-Pin	Fetching the ontology instance (business object) which should be hold by the associated part in the execution environment.
Constant	Returning the cached constant value.
Component	Invocation of the associated behavior definition.

4.3 Contracts

We further suggest associating to procedures, and especially to primitives, a new abstraction called *Contract* (see Figure 11). The idea consists in enriching the modeling system with a set of metadata about the modeling primitives. Contracts hold in particular information about the signature of the primitives (default name, and when pertinent, name and type of parameters and the result). For instance, the fact that our DSML for a banking system has a primitive called *Calculate Daily Interest* that needs two arguments of type *Percentage* and *Money* is

stored in a contract. Table 2 provides the list of all contracts associated to the operations used in this example. Each line corresponds to a contract for a *construct* of type primitive. Contracts can further hold metadata about the medium and execution mode (the type and amount of hardware required, the name of the runtime library, etc.). The exact type of metadata hold by contracts is however application specific.

Table 2: Description of the *contracts* used in specifying the *Accrue DailyInterest* task.

Name	Method	Inputs	Outputs
Calculate Daily Interest	calcDailyInterest	Money, Percentage	Money
Get Balance	getBalance	N/A	Money
Get Interest Rate	getInterestRate	N/A	Percentage
Deposit	deposit	Money	N/A
Select Account	selectAccount	N/A	Account

We consequently apply the Mediator pattern [GHJV95] to the design in Figure 9 to link the procedures to their execution strategy by the mediation of the contracts (see Figure 11). A specific type of contract should be designed for each specific type of procedure, execution context and strategy.



Figure 11: Associating procedures to execution strategies by mediation of *contracts*.

Table 3: Description of the different constructs of *Dart*.

Construct	Description
Primitive	Allows specifying a step whose value is computed by calling a 'primitive' function, e.g., a method, a function in a library, even a task.
Factory	Allows specifying a step whose value is computed by instantiating/selecting a specific business object.
Getter	Allows specifying a step whose value is computed by fetching the value of an attribute of a given business object.
Setter	Allows specifying a step that sets the value of an attribute of a given business object.
Control Structure	Allows specifying a step that carries an iteration or a conditional.
In-Pin	Allows creating a step whose value is received as argument. In-Pins are used in conjunction with components, in the sense that the behavior associated to a component contains steps of type In-Pin whenever some values should be passed to it at run-time. For instance, in the example illustrated in Figure 1, step 1 could be an In-Pin, allowing to the workflow to operate on any account received at run-time as argument. Such a behavior could then be <i>wrapped</i> as a reusable component and called by any part willing to 'accrue the daily interest' for a given account.
Constant	Allows creating a step whose value, a string, date, number or any other business object, is provided at definition time and will not change at runtime.
Component	Allows creating a step whose value is computed by executing a behavior specification.

4.4 Constructs

Now that we have modified the design of MWF procedures, we must face the challenge of adapting other MWF modeling constructs, such as the control structures, to the new design philosophy, and also adding new constructs such as parameterized tasks. Recall that our ultimate goal is a system which targets *both* developers and domain experts. Adapting and adding new constructs should therefore keep the system easy to reuse and extend by programmers, and also easy to learn and to use by domain experts.

We have achieved this goal by adopting ideas from the formula languages investigated by [Nardi93], where notably control structures are used seamlessly like primitives (an iteration or conditional is defined in the same way as an addition or an average). For space reasons we cannot describe the details of our design. Table 3 roughly describes the constructs that we have added. Figure 12 puts them in the context of our class diagram. As an example, the step 1 in the example in Figure 2 uses a *factory* construct. Other steps use a primitive one.

From the design point of view these constructs are added by specializing Procedure by a new abstraction called Construct. All modeling constructs of Dart correspond then to specializations of Construct. Figure 12 provides an abstract view of the final design.

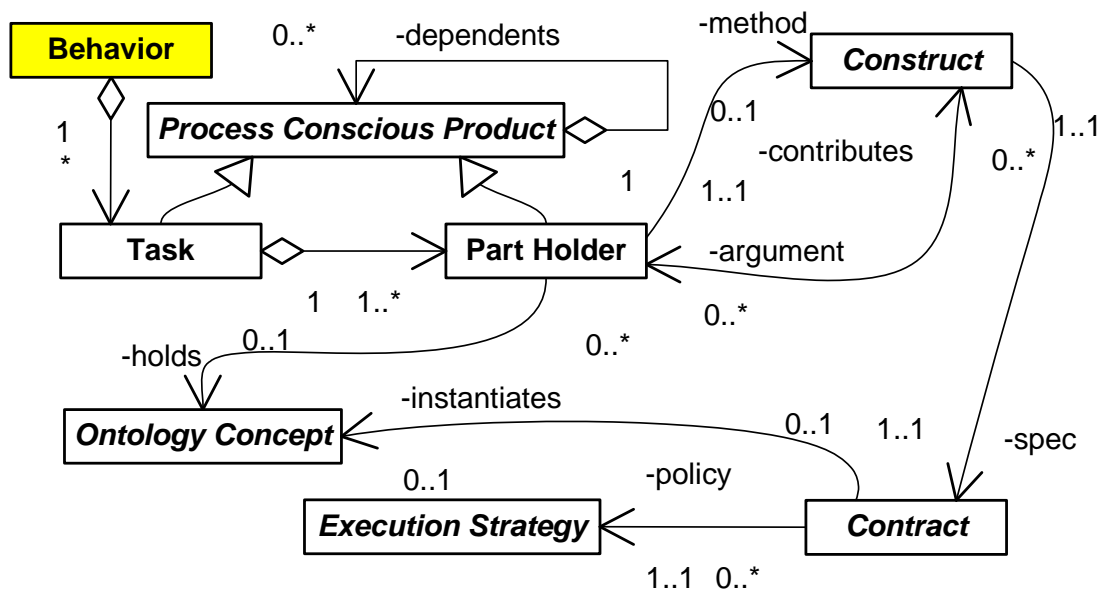


Figure 12: The Dart process meta-model.

5 Putting It All Together

The two separations of concerns that we operated by refactoring the MWF core are essentially motivated by the necessity to extend it to support End-user Programming [Nardi93]. These refactorings, together with the addition of *contracts* and *constructs* lead to a design (Figure 12) that is more consequent in terms of number of abstractions and their relationships than the MWF core (Figure 4). It is consequently harder to learn. The counterpart is that Dart provides more flexibility and more desired features for programming by domain experts, as follows.

5.1 End-user programming and adaptivity

Contracts allow an explicit description of the language constructs and primitives in a human-readable format. Conforming to the analysis of B. A. Nardi concerning task-specific languages, it becomes possible to package business knowledge in a set of well-described primitives and then present them to experts in a neat and structured manner (cf. e.g. [GLS02]). Contracts can also serve for guiding experts at modeling and model fine-tuning at execution time. For instance, it becomes possible to automatically identify that the *Calculate Daily Interest* primitive requires two arguments of types `Money` and `Percentage`. By coupling a type system to *Dart*, it becomes possible to filter choices and to avoid type mismatches when selecting arguments. Contracts allow further automating the generation of graphical interfaces for editing primitive instances. Material for online help can also be associated to contracts. At runtime, contracts help automating type checking on effective arguments and produced results. They can also automate the selection of better execution strategies according to the primitive's resource consumptions and the actual execution environment. We currently take advantage of this feature in developing ambient systems [RMSAP06].

Modeling steps by combining part holders with procedures (instead of uniquely procedures) brings several new possibilities. It allows developing modeling environments with a spreadsheet look & feel, well-known for their accessibility to domain experts [Nardi93]. Domain experts can model complex behavior by simply (1) selecting amongst the contracts, the primitive to instantiate, (2) selecting the grid cell to which the *instance* of the primitive should be *attached*, and (3) selecting the arguments for the primitive amongst other cells in the sheet. We have successfully tested this idea by developing a Web-based and Dart-compliant graphical interface for a research prototype called *AmItalk* [CRZ05].

Additionally, following the *Observer* pattern [GHJV95], *Dart* couples a dependency mechanisms with the reification of arguments (cf. the `dependents` link in Figure 12). If the part holder P1 serves as argument to the primitive that computes the value of the part holder P2, then P2 is automatically made dependent of P1 so as it computes its value upon to any *significant* change in the definition of the primitive associated to P1. This feature, also present in spreadsheets such as Excel, is also very appreciated by domain experts. It prevents them from manually keeping track of the consequences of a change in a primitive definition or value.

Adding execution strategies which are used through the mediation of contracts, allows changing the execution policy at runtime, which is no more structurally attached to the task definition. It also allows deploying task definitions on non-object execution platforms. We are also exploiting this possibility in implementing ambient systems that feature runtime adaptivity to a changing execution context. From the domain experts' point of view, this feature is appreciated, since Dart dissociates the operational semantics of the task from their definitions. In conformance with the DSM approach, it becomes possible for the domain experts to focus on the expression of the business logic in terms of an (object) workflow or task. The platform then transforms the definition and deploys it, based on the contextual data.

Our industrial experience with FI-AOMs shows that experts use both artifact and activity-based modeling. It is often a question of perspective for them, and they need to be able to switch between these two perspectives. In effect, experts need to analyze both the products and the actions, for instance from cost and resource-effectiveness point of view. Thanks to the contracts, the reification of parameters, the management of dependencies and business objects types, it becomes also possible to recursively guide experts for finding the write

sequence of actions for achieving a specific product. Dart supports then the two activity modeling methodologies.

Last but not least, Dart provides a full reification of behavior modeling abstractions. Even complex control structures are fully reified. This allows domain experts defining complex procedures without low-level programming.

5.2 Ities: expressivity, modularity, reusability and extendability

The MWF primitives model also the formal arguments. However, an argument is represented as a symbol and not a full-fledged object. In Dart, arguments are represented by the part holders. This allows in particular designating as argument virtually any complex interpretable structure (*Interpreter* pattern) that implements the `value` protocol. We have used this feature in a successful metrology application [Raz00] to allow experts *embedding* mathematical expressions as arguments to other primitive calls.

Consequently, it becomes possible to hierarchically structure a computation, while keeping the same spreadsheet-like programming look & feel. For instance, an `ifElse` conditional can be represented as a “primitive” that takes two arguments which are themselves workflows. At runtime, the predicate-workflow is executed first, and the action-predicate is executed only if it returns true. Programmers can relatively easily extend *Dart* to add specific control structures, adapted to the business domains and domain experts. The *Mobidyc* system [GLS02], which reuses an implementation of *Dart*, has taken advantage of this possibility to implement a variety of control structures.

From the framework design point of view, having separated the different roles in the design of procedures makes the architecture more flexible by allowing the evolution of one aspect without being limited by the constraints imposed by the other aspect. In other terms, Dart decomposes the process model component of the MFW into several reusable, extensible and finer-grained components.

6 Conclusions and perspectives

An extension to the MWF core component dedicated to workflow definition and execution is proposed. We show that the goals of a workflow architecture that targets both developers and domain experts is achievable. Many enhancements and more flexibility (including new hooks for dynamic adaptivity) are possible.

To experimentally validate Dart, we have developed an object-oriented framework using *VisualWorks* Smalltalk, which we first used in an ecology simulation system [GLS02]. This prototype is being reused in a project related to the *Ambient Intelligence* and *Ubiquitous Computing*, where we are further deploying this architectural style for developing a macro-programming environment for *Wireless Sensor-Actuator Networks* [RMSAP06].

7 Acknowledgements

This work has been partially funded by the University of Luxembourg, in the framework of the *Ámbiance*, a project in the field of Ambient Intelligence (R1F105K04). We would also like to acknowledge the valuable collaboration of G. Agha, I. Borne, A. Cardon, N. Bouraqadi, Ch. Dony, B. Foote, V. Ginot, Ph. Krief, M. Malvetti, D. Manolescu, K. Mechitov, S. Sundresh, and J.-W. Yoder.

8 References

- [YJ02] Yoder JW, Johnson R. The adaptive object-model architectural style. In: Bosch J, Morven Gentleman W, Hofmeister C, Kuusela J, editors. Third IEEE/IFIP conference on software architecture (WICSA3). IFIP conference proceedings 224. Dordrecht: Kluwer. p. 3–27. 2002.
- [RBYPJ05] Razavi, R., Bouraqadi, N., Yoder, J.W., Perrot, J.F., Johnson, R.: “Language Support for Adaptive-Object Models using Metaclasses”. In the Elsevier Int. journal Computer Languages, Systems and Structures. Bouraqadi, N. and Wuyts, R. (Eds.) Vol. 31, Number 3-4, ISSN: 1477-8424, October/December(2005).
- [Tolvanen05] Tolvanen, J.-P.: Domain-Specific Modeling for Full Code Generation. Methods & Tools - Fall 2005.
- [WMC99] The Workflow Management Coalition. Process definition model and interchange language. Document WfMC-TC-1016P v1.1. October 1999.
- [LR2000] Frank Leymann and Dieter Roller. Production Workflow—Concepts and Techniques. Prentice-Hall, Upper Saddle River, New Jersey, 2000.
- [AJ98] Francis Anderson and Ralph Johnson. "The Objectiva telephone billing system". MetaData Pattern Mining Workshop, Urbana, IL, May 1998.
- [DT98] Martine Devos and Michel Tilman. A repository based framework for evolutionary software development. MetaData Pattern Mining Workshop, Urbana, IL, May 1998.
- [Raz00] Razavi, R.: “Active Object-Models et Lignes de Produits – Application à la création des logiciels de Métrologie”. In proceedings of OCM’2000, 18 - May 2000, Nantes, France, pp 130-144 (2000)
- [GLS02] Ginot, V., Le Page, C., Souissi, S.: “A multi-agents architecture to enhance end-user individual-based modeling”. Ecological Modeling 157 pp.23-41 (2002).
- [CDRW02] Caetano H., De Glas M., L., Rispoli R, Wolinsky F.: The Importance of Fragility in the Realisation of Seismic Simulators: The Urban Risks Example. Assembleia Luso Espanhola de Geodesia e Geofisica (2002).
- [YBJ01] Yoder J, Balaguer F, Johnson R. Architecture and design of adaptive object-models. SIGPLAN Notices;36(12):50-60, 2001.
- [Manolescu2000] Manolescu, D.: “Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development”. PhD Thesis, University of Illinois at Urbana-Champaign, Illinois (2000).
- [Manolescu2002] Manolescu, D.: “Workflow enactment with continuation and future objects”. Proceedings of the 17th OOPSLA Conference. ACM Press, ISBN 1-58113-471-1. Pages 40 – 51. Seattle, Washington, USA (2002).
- [Opd92] William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [RTJ05] Dirk Riehle, Michel Tilman, and Ralph Johnson. “Dynamic Object Model”. In: Pattern Languages of Program Design 5, Addison-Wesley (2005).
- [GPW99] Dimitrios Georgakopoulos, Wolfgang Prinz, and Alexander L. Wolf, editors. Proceedings of WACC99, volume 24 of Software Engineering Notes. ACM, March 1999.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns---Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [Nardi93] Nardi, B. A.: “A Small Matter of Programming: Perspectives on End User Computing”. MIT Press, Cambridge, MA (1993).
- [RMSAP06] Reza Razavi, Kirill Mechitov, Sameer Sundresh, Gul Agha, Jean-François Perrot: Ambiance: Adaptive Object Model-based Platform for Macroprogramming Sensor Networks. Poster session extended abstract. OOPSLA 2600 Companion October 22–26, 2006, Portland, Oregon, USA (to appear).
- [CRZ05] Stéphane Célet, Reza Razavi et Pouryia Zarbafian : “AmItalk: towards MDE/MDA Tool Support for Ambient Systems”. Communication to the ESUG Innovation Technology Awards (2005).