Dynamic Object Model and Adaptive Workflow[*]

Dragoş A. Manolescu and Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{manolesc,johnson}@cs.uiuc.edu

1 Introduction

One of the problems with current workflow systems is their limited support for dynamic environments and evolving product and process models. A 1997 study of trends in workflow management concluded that exception handling (i.e., dynamically modifiable processes) and object-oriented views of workflow definitions deserve "serious attention" from researchers [Moh98]. But why are dynamic models so important for adaptive workflow?

In [HC93], Hammer and Champy identify three factors that characterize modern businesses: (i) customers take charge, (ii) intense competition, and (iii) constant change. In workflow terms, customers in charge means support for ad-hoc processes. Constant change translates into dynamic process models. In a world of intense competition where "nothing is permanent except change itself" [Cha96], support for ad-hoc processes and dynamic process models are requirements for a succcessful business.

Scientists and engineers who work in computerized environments use "scientific" workflow systems. They also require flexible models for process definition and execution [BBG$^+$98]. The outcomes of scientific processes evolve as the experiment unfolds. It is difficult to determine in advance the structure of the entire process. Scientific workflow systems must allow workflows and product data to evolve at runtime.

2 Dynamic Object Models for Changing Environments

Computer scientists have long recognized the importance of adaptive, dynamic systems. Winograd and Flores identify two clear objectives for software design: anticipating the forms of breakdown and providing a space of possibilities for action when they occur [WF86]. Their ideas also provided the catalyst for the use of computers to manage and coordinate activities.

The object-oriented community is trying to provide solutions to the challenges associated with building software for dynamic environments. In the remider of this section we introduce one of these solutions, the Dynamic Object Model [Joh]—also known as Active Object Model or User Defined Product.

Most object-oriented systems employ a static object model. The system architect defines the object model during the design stage and then programmers translate it into code. A static object model is fixed and doesn't change at runtime. In contrast, the Dynamic Object Model (DOM) architecture stores its object model in configuration data and *interprets* it at runtime. Changing the object model will immediately result in a change in behavior. Since the object model is represented as data, it is usually easy to change.

[*]Additional information about this research is available on the Web at `http://www.uiuc.edu/ph/www/manolesc/Workflow/`

DOMs strive to represent knowledge about a domain as relationships between the objects that model the domain. They avoid hard-coding this knowledge into the code, since usually code can't change at runtime. Even when it does, the people who can perform these changes are a scarce resource. One of the fundamental ideas of this architectural style lies in the ability to configure and manipulate its objects as any kind of data.

DOMs usually emerge from mature domain-specific frameworks. As developers evolve a framework from white-box to black-box [RJ97], they gain a better understanding of the domain. This enables them to recognize the "hot spots" of the design—the parts that are likely to change. Next, developers factor out these parts into configuration data. What's left is a generic system that interprets the configuration data at runtime. Changing the behavior doesn't require code changes any longer.

This approach yields a dynamically modifiable system, where the data contains the configuration information. Users can modify the system without programming and can defer configuration decisions until runtime. DOMs expose only those aspects of the problem domain that they need to change.

However, these characteristics have their price. This style is complex and systems that use it are usually developed by a small group of highly experienced developers. Although this architectural style enables a small team develop complex applications, it usually takes them a long time to develop the framework, and less experienced developers find the applications harder to debug, maintain, and understand. Additionally, when you let users modify the system, you also have to deal with their mistakes. For this reason, DOM applications require user support tools.

The DOM architectural style has been successful in domains where the business rules change often. It works best with intellectual (abstract) products. We have studied this type of architecture in the insurance framework at the Hartford [OJ98], the Objectiva telephone billing system [AJ98] and the Argo school administration system in Belgium [DT98].

Manufacturing concrete products also changes, but not quite as often. For example, automobile makers release new models every year. Manufacturing can benefit from DOM, but the benefit might not be worth the cost.

3 The Structure of the Dynamic Object Model

A few key ideas form the foundation of a Dynamic Object Model architecture. These ideas have been studied and documented as design patterns [GHJV95] (printed in *slanted fonts* throughout this document).

The most important is *Type Object* [JW97], which separates an Entity from an EntityType. Entities have *Properties* and *Type Object* is used a second time to separate Property from PropertyType. *Strategy* objects [GHJV95] define the behavior of an EntityType.

A few other patterns provide the support for Dynamic Object Model architectures. *Metadata* [FY98] supplies the mechanism that allows the architecture to represent the knowledge about the domain as configuration information. *Visual Builder* [RJ97] provides a means for the now-empowered users to interact with the object model.

3.1 Properties

Objects encapsulate state and behavior. State corresponds to attributes which are usually implemented with instance variables. Changing the instance variables of an object requires changing its code.

A dynamically modifiable object model needs a scheme that does not require code changes. We want a way to add or remove attributes on the fly. The solution is to implement attributes in a different way. Instead of having an instance variable for each attribute, the *Property* pattern uses an instance variable that holds a collection of attributes. Each attribute is associated with a unique key. Users use these keys to access, alter, modify or remove attributes at runtime.

For example, consider a workflow system for leasing real estate properties [DGSZ94]. The system automates the creation of lease contracts. Creating a contract begins with the production of a set of documents that will be assembled into the final lease contract. This is followed by a series of legal and economic checks. If the checks succeed, the system generates a lease contract. The workflow system has different leasing processes depending on the type of the lessee. For private individuals, the legal check consists of contacting the previous lessor and the economic check requires a bank statement. In contrast, commercial institutions have different checks.

The leasing processes have attributes that store the starting date, the name of the person who entered the initial data, etc. A possible implementation is to use instance variables for all these types of information. For a DOM we use the *Property* pattern to represent the attributes. With this solution, all process instances replace their instance variables with a single variable that stores their properties. Figure 1 shows the UML [FS97] class diagrams corresponding to the two implementations.

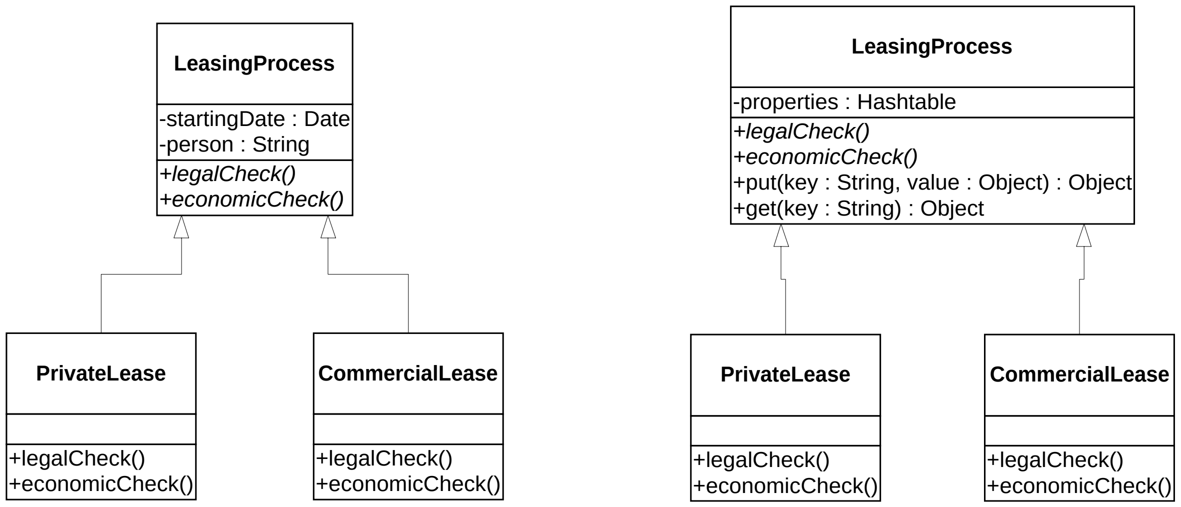

Figure 1: Converting from static to dynamic attributes—class diagrams. The left diagram shows the attributes implemented with instance variables, while the right diagram uses the *Property* pattern.

The *Property* pattern makes it possible to change attributes without code changes. Users can now access, modify, but also *add* and *remove* attributes at runtime. For example, let's assume that the real estate agent opens multiple branches. Each process requires now an additional attribute that stores the name of the branch where it was started. How can the system accommodate the change? This will be quite a production for a system that uses instance variables. First, end users cannot make the change. They have to rely on a programmer who is familiar with the code. Second, after making the change, the programmer will have to re-deploy the new system. This will probably require the import of any processes that were running on the old system. With *Property*, this change doesn't require a programmer. Users can add the extra attribute at runtime, without stopping the system.

3.2 Strategy

In an object-oriented programming language, methods or virtual functions define the behavior of objects. Programmers define an object's methods by writing code. Along with state, these methods are part of the features that make objects so powerful. However, most languages do not allow objects to control their own methods.

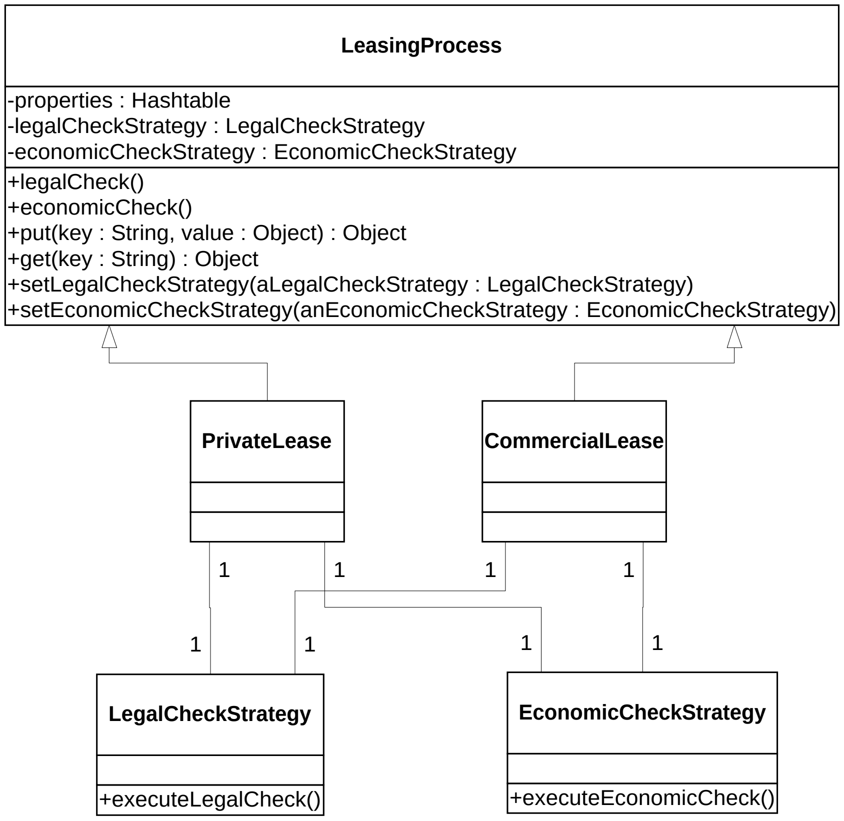
LeasingProcess
-properties : Hashtable
-legalCheckStrategy : LegalCheckStrategy
-economicCheckStrategy : EconomicCheckStrategy
+legalCheck()
+economicCheck()
+put(key : String, value : Object) : Object
+get(key : String) : Object
+setLegalCheckStrategy(aLegalCheckStrategy : LegalCheckStrategy)
+setEconomicCheckStrategy(anEconomicCheckStrategy : EconomicCheckStrategy)
PrivateLease
CommercialLease
1
1
1
1
1
1
1
1
LegalCheckStrategy
+executeLegalCheck()
EconomicCheckStrategy
+executeEconomicCheck()

Figure 2: The application of the *Strategy* pattern to the leasing process workflow—class diagram.

Although this scheme usually works very well, it is not sufficient for a dynamically modifiable object model. We want to be able to control an object's behavior. We need to represent behavior with a user-level mechanism. A *Strategy* is the reification of behavior into objects. The *Strategy* pattern defines a standard interface for a family of algorithms. Clients can work with any algorithm of a given family. If an object's behavior is defined by one or more strategies then that behavior is easy to change.

For example, in the leasing real estate workflow, the legal and economic checks are part of the behavior. In a static object model, the typical implementation is to have a method for each of these operations. The alternative is to have a *Strategy* family for every type of check, `LegalCheck` and `EconomicCheck`. With this solution, users can configure the type of checks for each kind of process. Figure 2 shows the UML class diagram from Figure 1 (right) after we applied the *Strategy* pattern. The `legalCheck()` and `economicCheck()` methods in the base class delegate to the corresponding *Strategy* objects.

What happens if the agency wants to adopt a set of different legal checks for commercial institutions? For a system that implements this behavior in methods, this change requires code modifications and the re-deployment of the entire system. In contrast, with a DOM architecture, the user just plugs in a different `LegalCheckStrategy` object.

3.3 Type object

The majority of object-oriented languages structure a program as a set of classes. A class defines the structure and behavior of objects. Most object-oriented systems use a separate class for each kind of object.

Introducing a new kind of object requires making a new class, which requires programming. When there is little difference between objects, they can be generalized and the difference described by parameters.

So far the implementation of the real estate leasing process has a different subclass for each kind of leasing process, e.g., `PrivateLease` and `CommercialLease`. However, the only differences between these subclasses are their *Property* and *Strategy* configurations (see Figure 2). Further, the same person can have two contracts, one as a private individual, and one as a commercial entity. A lessee object needs to refer to its leasing processes. Some languages (e.g., C++) make it hard to have an object point to a class or to create an object from a class with a particular name. The alternative is to have a `LeaseType` object corresponding to all the leasing processes. A `LeaseProcess` will have a reference to a particular `LeaseType`. With this solution, it is no longer necessary to make a separate class for each kind of leasing process. Make a class `LeaseType` and create instances of `LeaseType` instead of subclasses of Lease. Figure 3 shows how the application of *Type Object* modifies the diagram from Figure 2.

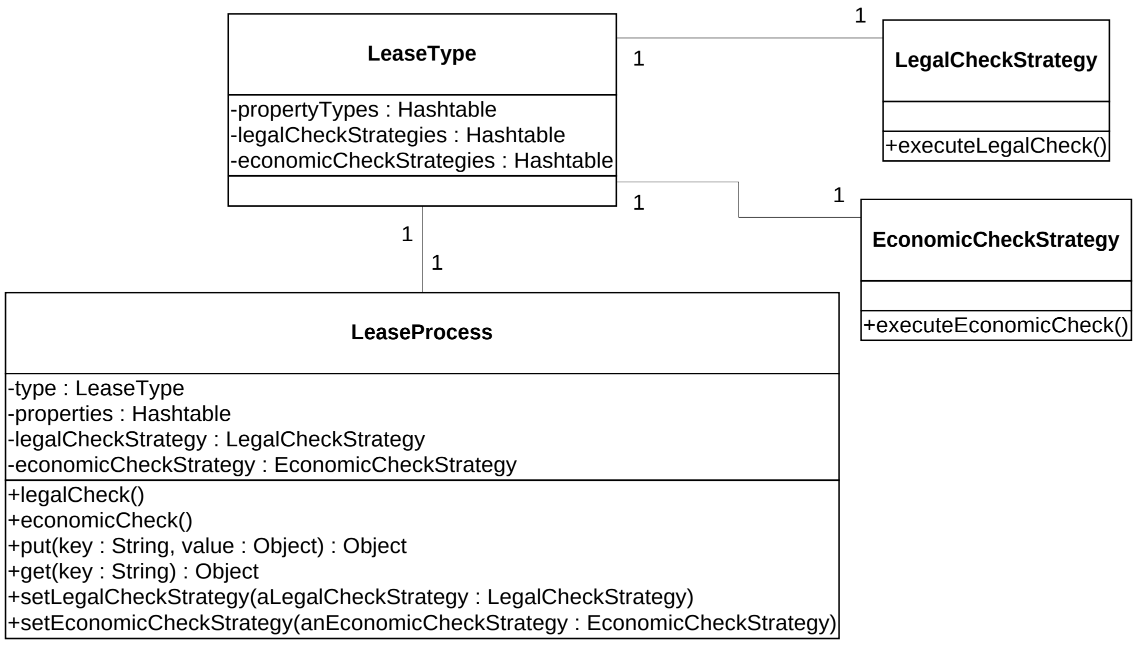
LeaseType
-propertyTypes : Hashtable
-legalCheckStrategies : Hashtable
-economicCheckStrategies : Hashtable
LegalCheckStrategy
+executeLegalCheck()
EconomicCheckStrategy
+executeEconomicCheck()
LeaseProcess
-type : LeaseType
-properties : Hashtable
-legalCheckStrategy : LegalCheckStrategy
-economicCheckStrategy : EconomicCheckStrategy
+legalCheck()
+economicCheck()
+put(key : String, value : Object) : Object
+get(key : String) : Object
+setLegalCheckStrategy(aLegalCheckStrategy : LegalCheckStrategy)
+setEconomicCheckStrategy(anEconomicCheckStrategy : EconomicCheckStrategy)
1
1
1
1
1
1

Figure 3: The application of the *Type Object* pattern to the leasing process workflow—class diagram.

The *Type Object* pattern makes the class-instance classification relationship explicit. Instances of the type class (`LeaseType` in our example) replace subclasses of the original class (`PrivateLease` and `CommercialLease`). Users have full control over this relationship. They can even modify it at runtime. For example, what happens when the type of lease changes while the process is executing? A person may decide that she would like to move her business office in the apartment that she applied for as a private individual. Some languages permit the class of an object to change (e.g., Smalltalk), but most do not. With *Type Object*, the classification relationship is at the user level and can be dynamically modified. Of course, simply changing the process type is not enough—we also need to convert the attributes. However, now the user controls the state, behavior, and classification relationship.

The core of a DOM architecture is a combination of *Type Object*, *Property* and *Strategy*. Figure 4 shows the corresponding UML class diagram. The *Type Object* pattern divides the system into `Entities` and `EntityTypes`. `Entities` have `Attributes`, each of which has an `AttributeType`. Each `EntityType` specifies the `AttributeTypes` of its `Entities`. The `EntityType` also holds a set of `Strategy` objects as *Properties*.

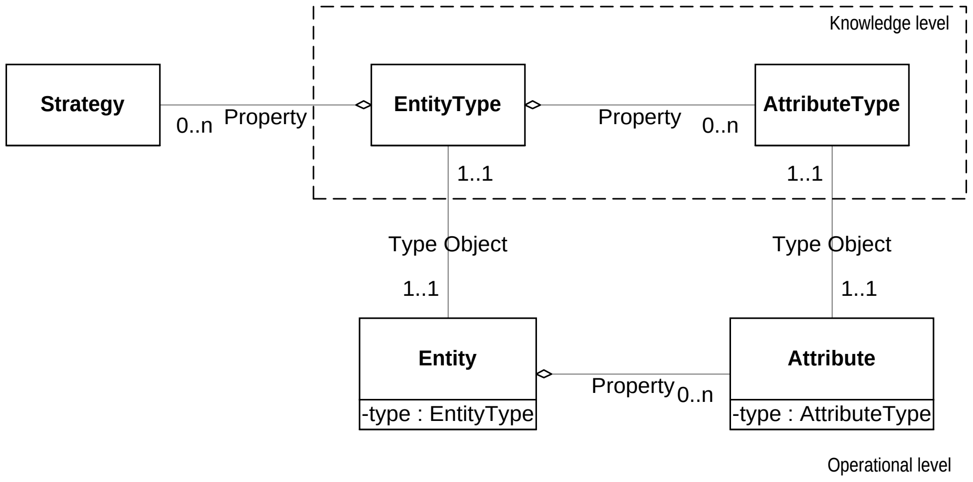

Figure 4: The core of the Dynamic Object Model architecture—class diagram.

3.4 Metadata

Type Object, *Property* and *Strategy* are the building blocks of the architecture. DOMs represent the reminder of the object model as configuration data, or *Metadata*. Therefore, modifications of the configuration data change the object model. In turn, changing the object model alters the system behavior.

Two factors contribute to the extended flexibility of DOM architectures. First, only the generic part is hardwired. Types, entities, attributes and algorithms are universal concepts. Therefore, this core is the same for a wide range of problems and we don't need to change it. Second, the variable part is pushed into *Metadata*. Users have full control and modify it to adapt the object model to their needs.

However, abstracting the generic parts and identifying the parts that are likely to change requires experienced architects. Usually they achieve this insight only after a few iterations. This is one of the reasons that make DOMs hard to build.

Let's return to the real estate workflow example. How does the leasing process obtain the information about its attributes? A possible implementation is to have code that performs this initialization. However, this requires programming. Our goal is to avoid code changes. Further, now *Property* allows users to dynamically add or remove attributes without touching the code. Likewise, *Strategy* represents behavior with objects. We need a different mechanism. The alternative is to describe the types, their attributes and behavior in *Metadata*. When bootstrapped, the system reads this information from a repository and initializes the process types.

3.5 Visual builder

One of the main reasons to build a DOM is to extend the system without programming. Users have direct access to the object model. They control the relationships between its core components and the *Metadata* information.

DOM architectures push complexity into the configuration data and delegate configuration decisions to the users. We can think of metadata as a domain specific language. Users become specialized programmers. The key here is to expose only those concepts and rules that they need to operate with, in a way that they could understand. Non-programmers are likely to use a specialized programming language if it is in the guise of *Visual Builder*s.

Let's return one more time to the leasing process workflow. We have discussed how we employ *Strategy* objects to encapsulate different types of `LegalChecks` and `EconomicChecks`. Users use a *Visual Builder* to configure the types of checks for each leasing process. They perceive this as picking an item from a list and dropping it on a process configuration window. Within the DOM architecture, this action translates into plugging a *Strategy* object into the type side of a *Type Object*. However, the builder hides all these details from the user.

4 An Object-Oriented Workflow Model

Workflow systems amalgamate technologies, principles and methodologies from numerous areas of computer science. Consequently, they are hard to build. Support for modifiable process models and dynamic environments makes their construction even more challenging. For example, OMG's Request for Proposals for the Workflow Management Facility [OMG97] lists ad-hoc workflow (i.e., where the process is not well-defined in advance) as an optional requirement. This shows that the community is aware of the difficulties and is seeking solutions. According to [Moh98], only a few commercial systems allow their users to modify executing workflows, e.g., InConcert and TeamWARE Flow.

The typical workflow system architecture consists of three components [Hol95, AAAM97, Sch96]. Build-time functions provide support for process definition and modeling. Run-time control manage process execution and their associated resources. Finally, run-time interaction functions provide the interfaces to human users and IT application tools. Usually, a workflow engine integrates both run-time control and interaction functions.

The workflow engine handles process enactment. This consists of reading process definitions; creating new process instances; and scheduling the various steps within the process and the appropriate resources. For adaptive workflow, we should be able to change the process definition while the process is running. We should also have a way to change a particular process instance.

When engineers approach a new problem, sometimes they translate it into an equivalent problem for which they already have a solution. We apply a similar strategy for adaptive workflow. We look at workflow through the lens of object-oriented technology. Since we are familiar with DOMs in the context of flexible object-oriented systems, we'd like to use the same architectural style and techniques for adaptive workflow and modifiable process models.

We first need an object model for workflow. To date, a standard model is not yet available. The OMG is in the process of adopting one as part of its Workflow Management Facility. We present a critique of the Nortel and jFlow proposals in [MJ].

Looking at process enactment from an object-oriented standpoint, we see a familiar image. *Creating process instances from a process definition is similar to creating instances of a class.* This similarity is the crux of our framework. Having established a correspondence between the workflow domain and the object-oriented domain, we're on a well trodden path.

5 A Workflow Framework

We represent workflow with "procedures." Procedures form the glue that connects together domain objects. They encapsulate control flow and, indirectly, data flow. The procedure system provides a framework for workflow management. Carefully designed, such a framework is reusable across different domains.

Most applications consist of domain objects and glue code that links them together. It takes a while to get domain objects right, but they don't change a whole lot afterwards. (This doesn't mean that they don't change at all.) In contrast, the glue code changes quite frequently. Many “new” products consist of the same “old” domain objects wired in a different way.

For example, an insurance company may pay claims for auto insurance that total less than a certain amount without examining each claim individually. The company would require the client to use the services of selected repair shops with which it has agreements. This decision saves the company the costs of auditing each claim. The reduced turnaround time also keeps clients happy. However, to prevent fraud, auditing personnel selects claims that appear suspicious and checks whether the repair shop is overcharging. This strategy doesn’t require new domain objects. However, the auditors must be able to change the processing rules for individual process instances. Therefore, it makes sense to invest into a workflow framework. First, the framework will help reuse existing domain objects better. Second, it doesn’t tie you to a particular domain. Should you want to use different domain objects, the workflow problems are still there.

We implement procedures on top of a Domain Model Engine (DME). This is a specialized object-oriented language for process and product models [MJ98]. The DME resides above the hardware and the system software. It provides the core of the DOM architecture described in Section 3.3.

5.1 Procedures as types

Applying the analogy between workflow and object systems, **we regard procedures as types/classes**. Procedure *types* in the knowledge level govern the configuration of procedure *objects* in the operational level [Fow97].

Here we use the *Type Object* pattern. The procedure definition resides on the type side. During the bootstrap stage, the DME initializes the definitions from metadata. At runtime, the DME builds the object model starting from these definitions.

We use the following basic types of procedures in the knowledge level:

Primitive corresponds to a single action on a domain object.

Sequence represents a *Composite* procedure. It has a number of steps, each of which is another procedure.

Conditional implements if-then branches .

Iterative iterates over the components of *Composite* domain objects (described as “composite work items” in [CJ98]). This procedure type doesn’t make any contribution to control flow. We use it as an additional structure that helps us to keep the control logic outside domain objects. For example, a portfolio may contain different kinds of assets. Let’s assume that we want a procedure that computes some function on this portfolio—e.g., its projected value, which consists of the values of its components plus the interest. The procedure needs each component of the portfolio, since different assets have different rules. The Iterative procedure first obtains the components of the portfolio and then executes the procedure that computes the value on each of them.

Parallel executes its procedure components simultaneously.

The implementation of a process consists of a combination of procedure types, organized in a tree structure. To perform a process, we execute the procedure at the root of the corresponding procedure tree. In turn, this will execute its children, and so forth. A regular (i.e., not dynamically modified) process ends when the control returns back to the root node. Figure 5 shows the UML instance diagram corresponding to the leasing process from Section 3.

We usually employ the *Composite* design pattern to implement tree structures. For adaptive workflow we want to be able to dynamically change the type of a procedure. (Section 6.2 has an example.) The catalog

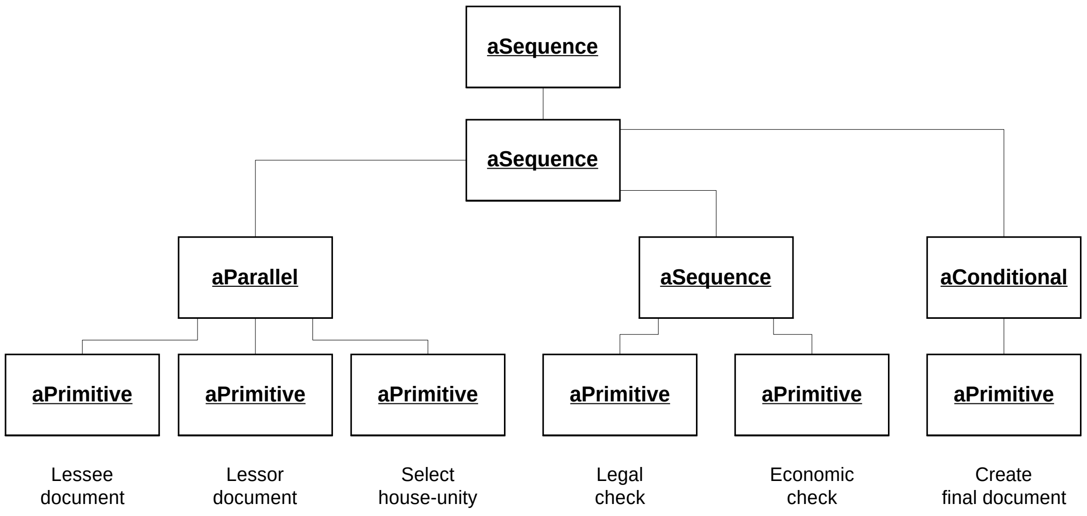

Figure 5: The definition of a process corresponds to a tree of `ProcedureType` objects—instance diagram.

form of the *Composite* pattern [GHJV95] uses separate classes for node and leaf elements. To provide maximum runtime flexibility, we don't want to commit a procedure type to a particular class. Therefore, we employ a variant that doesn't use a classification relationship. Leaves can now change into composites and vice versa.

In programming languages, a component closely related to data types is the type system. Strongly-typed languages like C++ or Java perform type checking. The compiler analyzes and checks the function or method signatures at compile time and guarantees them at runtime. Although far from proving program correctness, this type checking is one reason why some users prefer strongly-typed languages for mission-critical applications. Since the workflow framework also employs types, what is the equivalent of type checking? Here we can benefit from another consequence of the *Type Object* pattern. Besides the classification relationship, *Type Object* also brings the type checking mechanism at the user level. In other words, we can define the rules that determine the valid combinations of procedure type objects. Moreover, unlike for programming languages, these rules are not fixed any longer. They can evolve with the rest of the system. We'll resume this discussion in the next section.

5.2 Data flow

So far we have discussed how procedure types encapsulate basic control structures. What about the information that procedures operate with?

Although usually workflow systems do not explicitly represent data flow, it is nevertheless an important component. Many researchers identify the ability to pass data among the participants as one of the important factors that determine the effectiveness of a workflow management system.

We extend our parallel between workflow management and object-oriented programming. In programming languages, interpreters use closures or environments to capture values [FWH92]. For example, a function definition has a closure that stores the values of free variables. For each function call, the interpreter plugs in the values of the bound variables and performs the computation.

We use a similar idea to supply inputs and collect outputs from procedures. **Context objects serve as vehicles for data flow between procedures.** At runtime, each procedure executes within a context. An executing procedure extracts its input parameters from the context. When execution completes, the procedure stores its results (if any) back in the context. The parent procedure passes the context to the next procedure.

In adaptive workflow systems users can change the procedure types at runtime. We need context objects that can change as well. For example, assume that a procedure type adds an extra parameter. DOMs accommodate for this change without code modifications. The *Payloads* pattern [Man97] is a good candidate for this type of situations.

Having explained how procedures exchange information, we are now ready to complete the discussion of type checking. Procedure type checking rules work at the context level. Type checking ensures that each procedure type finds in the context supplied by its parent all the attributes that it needs. The *Visual Builder* uses the type information to assist users when they build new procedures. For example, when we edit the component of the Conditional from Figure 5, the builder displays a list with all possible choices. Initially, this list shows a *Limited View* [YB97] with only those procedures whose context requirements are compatible with the parent. Although generally this scheme works well, it is too restrictive for adaptive workflow.

5.3 Executing procedures

How do procedure types execute? Once again, we revisit the analogy between the workflow domain and the object-oriented domain.

Procedure execution is similar to object instantiation. At runtime, the type side creates a procedure object on the instance side and then transfers control to it. However, procedures only coordinate and connect domain objects. These objects carry out the domain-specific processing. This symbiosis between procedures and domain objects forms the application.

Execution begins with a client (an application or a procedure) requesting to execute a procedure type on a particular domain object. The client transfers control to the procedure type, within the workflow domain.

The procedure definition on the type side acts like a factory of executing procedure objects. This creates a new procedure instance with an empty context. Next the type side initializes the context of the new instance.

Now the procedure instance is ready to execute. The procedure definition transfers control from the type side to the instance side. We are still in the workflow domain, but have left the knowledge level.

At this point the procedure object triggers domain-specific work. It can't perform this work by itself, since procedures encapsulate only control logic. However, it can delegate to the domain object on which it executes. Procedure instances transfer control across domain boundaries.

Procedure execution resumes when the domain object transfers control back to the procedure instance. The context contains the results of the domain-specific processing. Finally, the procedure instance returns its context to the client, which can be a parent procedure or an application.

The UML sequence diagram from Figure 6 shows the mechanics of procedure execution. This execution model has several characteristics that support adaptive workflow. First, it keeps the control logic (workflow) outside the domain objects (application). Changes on any side of the inter-domain boundary have a minimal impact on the other side. Second, it maintains a clear separation between the type and instance levels within the workflow domain. Users can make local changes (at the instance level) or structural changes (at the type level). And finally, the context of the executing procedure determines the state of the system. Users can alter a particular process instance by modifying its context.

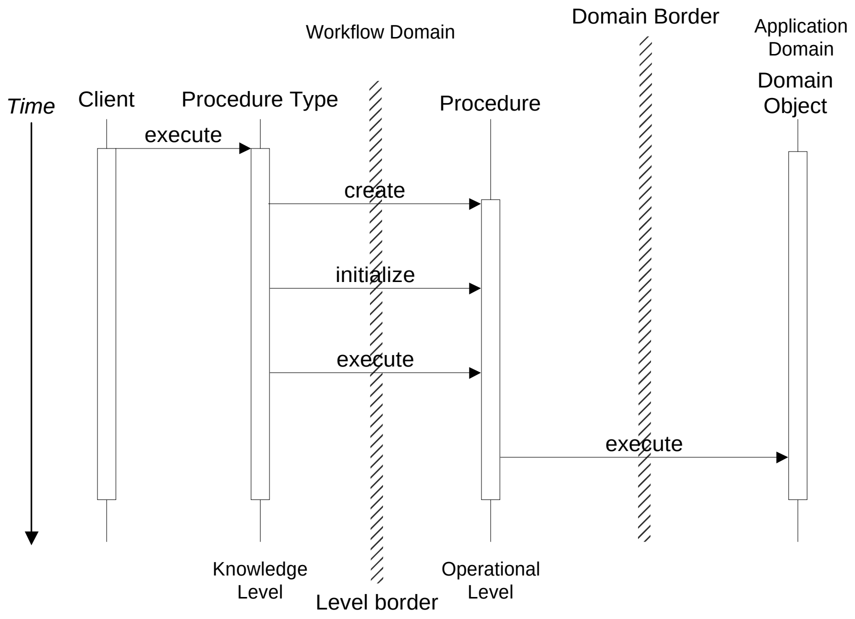

Figure 6: Procedure execution—sequence diagram.

6 Dynamic Object Model and Workflow Adaptation

We are now ready to see how the DOM workflow framework handles adaptive workflow. But what are the key requirements for adaptive workflow management? First, we need a workflow model that supports *evolution* as well as *ad-hoc modifications* of process instances. Model evolution is essential for Business Process Reengineering (BPR) and Continuous Process Improvement (CPI). Ad-hoc modifications allow users to adjust a particular process instance to specific circumstances. Second, we need a *flexible infrastructure*. The underlying software systems must be able to keep up with the process changes.

For example, let's consider the following common scenario from health care, adapted from [HSB98]. A patient arrives at the hospital with a certain health problem. The physician performs an initial examination and arrives at a diagnosis based on the observable symptoms and the patient's medical history. She then proceeds with the treatment corresponding to this first diagnosis. However, the physician also orders lab tests to obtain more information about the patient's health problem. When the test results arrive, the physician may have to modify the treatment to take into account the additional information.

We'll use this example to describe how the DOM workflow framework supports adaptive workflow. We proceed from the highest to the lowest level of abstraction. We also look at an additional characteristic supporting adaptive workflow, namely the convergence of build-time and run-time environments.

6.1 Domain level adaptation

We can think of software as having two dimensions. One corresponds to the application domain and contains domain-specific logic. The other corresponds to control and data flow (i.e., workflow) and is domain-independent.

Generally we use the same tool (usually a programming language) in both dimensions. This characteristic encourages beginners to combine the two dimensions. Consequently, domain-specific code and control logic are usually intertwined within applications. Inexperienced programmers see (and think in) a unidimensional problem space.

In contrast, workflow is a tool only for the control and data flow dimension. On the one hand, this means that we have to think a little differently about software. Sometimes it may even seem awkward to wear two hats, one for the application domain and one for the workflow domain. On the other hand, the divided perspective pays off with additional flexibility. We can change on either dimension with minimal consequences for the other.

Throughout Section 3 we used a process for leasing real estate properties. This Section begins with a process for health care. From a workflow standpoint, the difference between these two examples is the application domain. The leasing workflow focuses on contracts and various types of checks. In contrast, the health care workflow concentrates on symptoms, tests and treatments. Aside from domain specific processing, both workflows handle control and data flow. To facilitate domain level adaptation, application- and workflow-specific processing should be loosely coupled. However, only experienced architects achieve a good separation.

The procedure framework tries to maintain a reduced contact area between the application and workflow domains. Control crosses the domain boundary only from the instance side in the workflow domain, at the level of `Procedure` objects—see Figure 6. The Iterative procedure type helps us to keep the application domain unspoiled from control logic. This reduced coupling means that changing the domain objects has a low impact on procedures, and vice versa.

6.2 Process level adaptation

The workflow framework abstracts control flow in procedures. We use *Type Object* to separate the definition, `ProcedureType`, from the running instance(s), `Procedure`. In Section 3.3 we discussed some of the general consequences of *Type Object* in object-oriented design. For the workflow framework, this pattern separates the issues of workflow evolution and those of dynamic ad-hoc changes:

- On the one hand, process *owners* can modify the `ProcedureType`, in the knowledge level. This structural change affects all `Procedure` objects of the corresponding type. It is similar with the meta-model mechanism for adaptive workflow used by systems like CRISTAL [BBG+98]). Modification of the `ProcedureType` may have local or global *temporal* scope. Local changes affect the `Procedure` objects instantiated after the changes become effective. For example, modifications to the enrollment procedure for an insurance policy affect all policies issued after the new rules become effective. In addition, global changes also affect the `Procedures` that are currently running. For the insurance domain, changes in legislation are likely to have this effect.

  In the health care example, let's assume that the facility decides to offer its services only to patients who have a certain coverage. An employee authorized to change the workflow definition edits the corresponding `ProcedureType` with a *Visual Builder.* He selects a Conditional, adds a check and updates the definition. Now every instance of this workflow performs the coverage check.

  Figure 7 shows the UML instance diagrams corresponding to the original and updated process definitions. In the original process (left side), the root node contains a Sequence ("Diagnosis and Treatment," grayed) and a Primitive. In the updated process (right side), the process owner introduces a Conditional procedure between the root node and the "Diagnosis" Sequence. This procedure checks whether the patient has the required coverage. (In reality we'll probably need more than just a Conditional, but here we'd like to keep the definition simple.) The framework executes the "Diagnosis" Sequence only when the guard within the Conditional node resolves to true.

- On the other hand, process *users* can change the `Procedure` instance, in the operational level. These

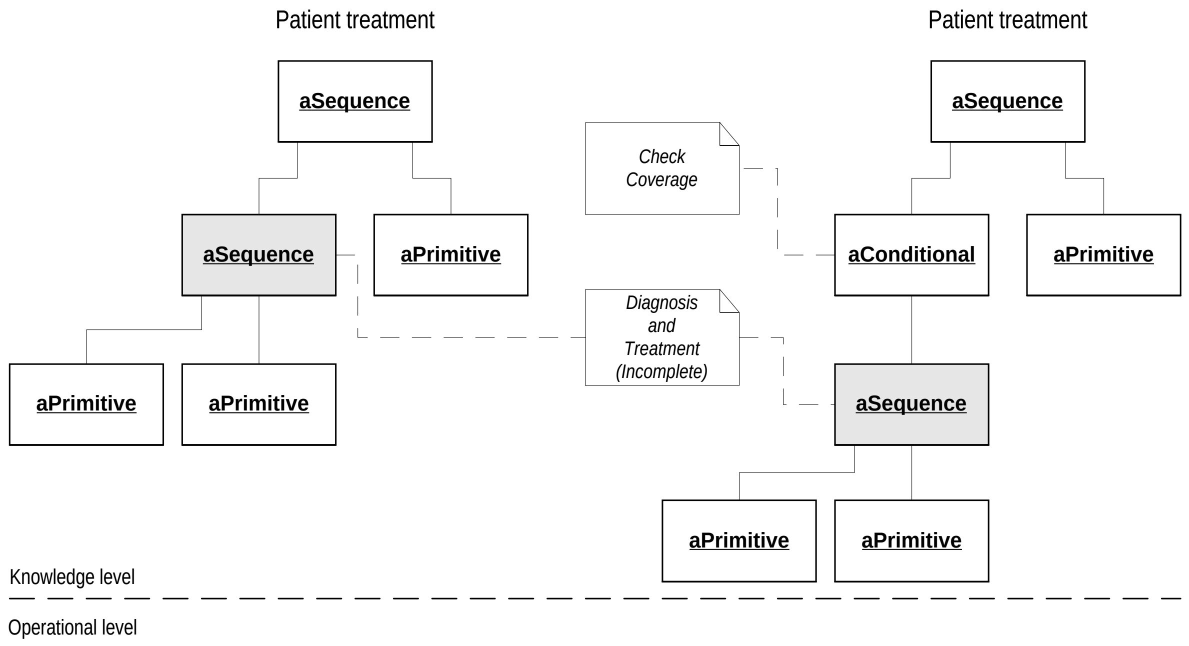

Figure 7: Changing the process definition—instance diagrams. The original (left) and the updated (right) process definitions. To improve readability, we show only two components within the "Diagnosis and Treatment" (grayed) Sequence.

changes have local scope and don't affect other users. Open-point workflow systems like $MOBILE$ [JB96] use a similar scheme.

Let's return back to the health care example. What happens when the test results arrive? For instance, they may reveal that the patient has acquired a new allergy. If this conflicts with the initial treatment, the physician needs to change the prescription and avoid an allergic reaction. She edits the workflow instance with the *Visual Builder* and replaces the steps to reflect the new information.

Figure 8 shows the UML instance diagrams corresponding to the original and updated running processes. In the knowledge level (upper half), the process definition consists of a tree of procedure types. Likewise, in the operational level (lower half), the process instance consists of `Procedure` objects. *Type Object* associates each `Procedure` with a `ProcedureType` in the knowledge level. (To improve readability, we show only one of these relationships.) As the process unfolds, the framework traverses its definition and creates the corresponding `Procedure` objects in the operational level. (`Procedure` objects exist only for the `ProcedureTypes` that have executed or are executing.) We show the currently executing procedure and its corresponding type in gray. The left figure corresponds to the initial treatment. The process executes according to its definition, determined from the observable symptoms and medical history. Now the lab tests arrive and reveal the new allergy. In the right figure, the physician makes a local change to the process instance. She changes the classification relationship from the initial treatment (dark gray) to a new one (light gray), selected ad-hoc for this process instance.

The implementation of a process corresponds to its decomposition in terms of the basic procedure types—see Figures 5, 7 and the upper half of Figure 8. Through this recursive decomposition (divide and conquer), the procedure framework keeps complexity under control. Each process corresponds to a tree of `ProcedureType` objects in the knowledge level. The *Composite* pattern maintains the same interface for

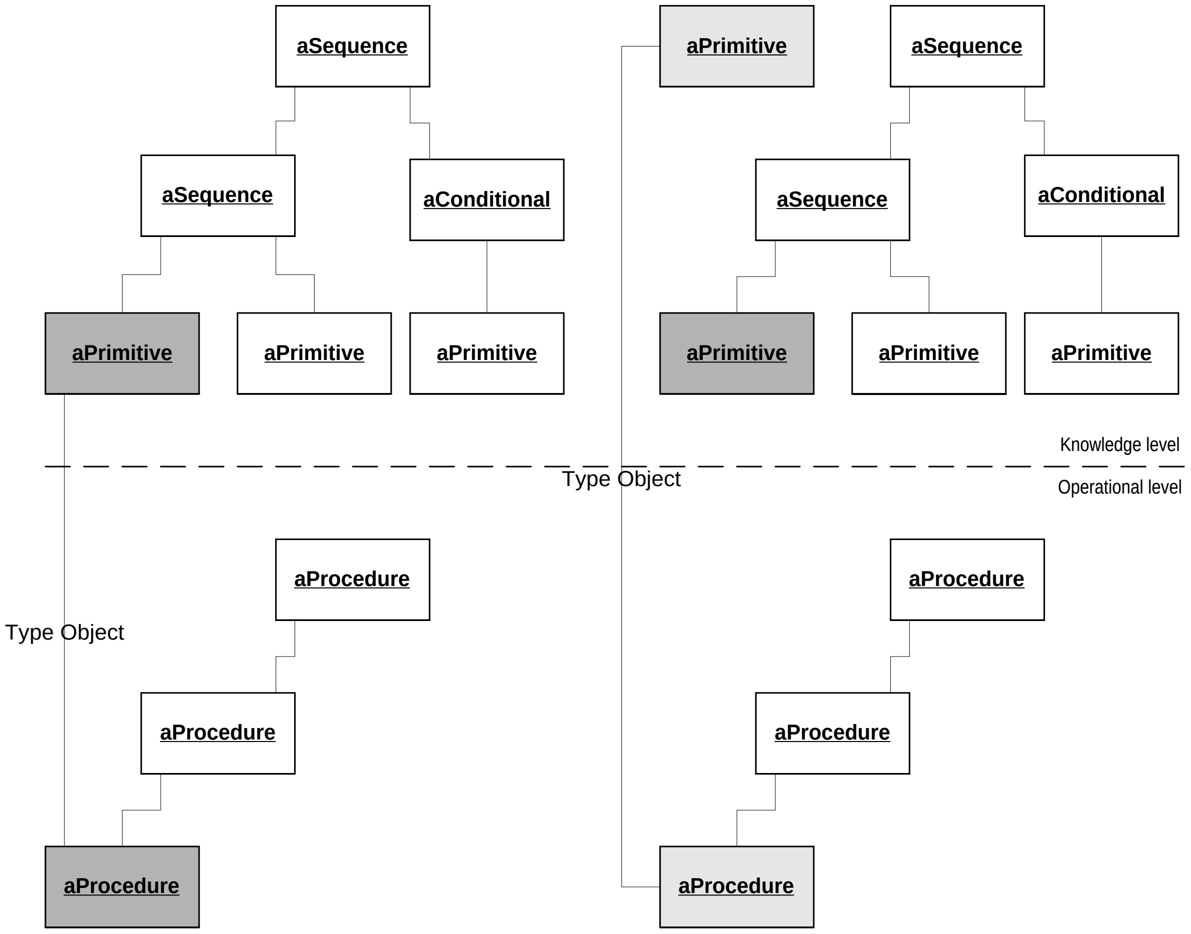

Figure 8: Changing the process instance—instance diagram. The left side illustrates the process executing according to its definition and the right side illustrates a local change. We show the currently executing `Procedure`s and their corresponding `ProcedureType`s in gray.

node and leaf procedure types. In other words, sometimes we may want to implement a (sub-)process with a single primitive procedure (leaf). But under different circumstances, we may want to use a composite procedure (node) for the same (sub-)process. For example, this could be a Sequence with an additional Conditional procedure. The UML diagram from Figure 9 illustrates this situation. The Primitive procedure (grayed) is now part of the Sequence, which contains another Primitive and a Conditional. Since both solutions have the same interface, the choice has minimal impact on the workflow framework. Further, we can make these changes at runtime. This **flexible composition** supports process evolution as well as ad-hoc modifications.

The procedure framework has another important characteristic for adaptive workflow. Let's revisit the mechanics of procedure execution—Figure 6. A client requests to execute a procedure type on a domain object. The `ProcedureType` first creates a `Procedure` instance and then transfers control to it. The key observation here is that the `ProcedureType` instantiates the `Procedure` at runtime, right before its activation. From an object-oriented perspective, we can think of this as a *dynamic binding* between a process and various potential implementations. In other words, we don't need the complete process specification at build time. A procedure can complete its specification at runtime. Alternatively, the workflow system may ask the

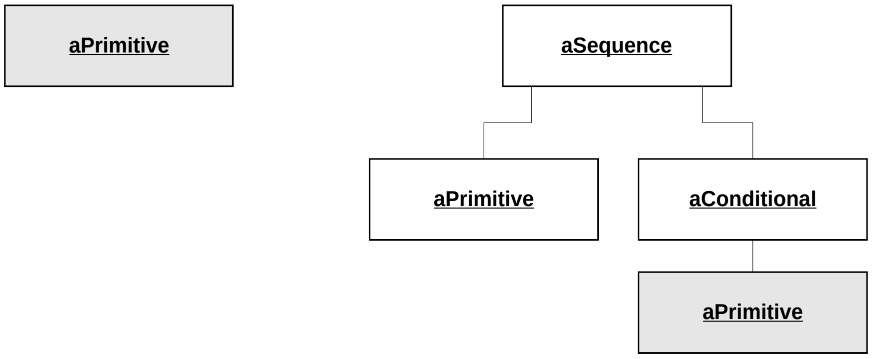

Figure 9: Replacing a Primitive procedure type (left) with a Sequence (right)—instance diagram.

user to choose one of the available choices. This **dynamic refinement** is key for handling processes with incomplete specifications.

A typical workflow process involves various software and human resources. Sometimes these don't respond according to the process designers expectations. Computers can crash, networks can go down and people can get sick and stay home one day. Further, while process designers plan ahead for some of these events, usually they can't envision every possible combination. Workflow systems employ exception handlers to deal with unpredictable events. For adaptive workflow we need **adaptive exception handlers**. At the framework level, we can implement exception handlers as procedure types. Users change the handlers in the same way they change process definitions. At the architecture level, we can use *Strategy* objects. Users manipulate and configure them with *Visual Builder*s, as any other objects. However, we only provide the mechanism for exception handlers. Their intelligent management is a better task for a different kind of systems [KD98].

6.3 Resource level adaptation

The procedure framework employs the DOM mechanisms to track the evolution of workflow resource objects. Some resources never change. Procedures obtain the information about static resources from *Metadata*. However, in an adaptive workflow system resources are dynamic.

Back to the health care example, a resource may be a system that provides various information about the patient, e.g., temperature, blood pressure, etc. The staff may retire the old system and introduce a new one, with extended functionality. For instance, the new system could also read the blood sugar level. Even when the staff doesn't replace the system, they could upgrade its software and change the interface.

Under these circumstances, the framework obtains information about a resource through the mechanisms provided at the DOM level. For example, it can query an `EntityType` to discover whether there are any changes in the `Attribute` or `Strategy` configuration. Alternatively, the framework can use the reflective capabilities of the resource object [VJK96, MKVlC98, EtH98]. However, we don't cover reflection in this version of the paper.

6.4 Infrastructure

Layered architectures [BMR+96] insulate each layer from changes on the underlying layers. For the workflow framework, the DME and the Common Services provide layers above the hardware and system software. The diagram from Figure 10 shows the system architecture. Changes in the infrastructure are transparent to the procedure framework.

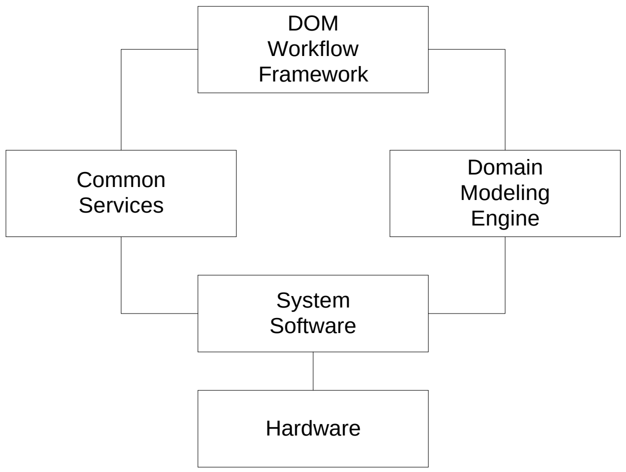

Figure 10: The layered architecture insulates the Workflow Framework from the System Software.

For example, the hospital may decide to replace the relational database used for persistence with an object-oriented database. This change is visible only at the DME level. In contrast, the procedure framework uses the mechanisms provided by the DME and consequently it doesn't depend directly on the underlying database. The procedure framework doesn't provide direct support for changes at the infrastructure level. This kind of adaptation is handled at the DME level.

6.5 Build-time and run-time environments

Many current workflow management systems keep apart the build-time tools and the run-time tools. For example, in the workflow reference model [Hol95], Interface 1 defines a "point of separation" between the build-time and run-time environments.

Based on the insights gained from applying workflow technology to real problems, researchers are investigating the relationship between build-time and run-time environments. Initially, separate environments seemed reasonable. Recent findings advocate for an integrated environment, particularly for adaptive workflow.

Ouksel and Watson investigate the capabilities needed by workflow systems to accommodate adaptive workflow [OJW98]. They identify the integration of build-time and run-time environments as an important requirement. Chiu and Li [CKL98] also recognize the importance of an integrated environment for workflow evolution. In [GT98], Georgakopoulos and Tsalgatidou examine the relationship between workflow management systems (WfMSs) and business process modeling tools (BPMTs). They conclude that their integration is a requirement for comprehensive business process lifecycle management.

As we explained in Section 3.5, the DOM architecture integrates build-time and run-time functions and facilitates user involvement. In the health care example, physicians interact with the object model through *Visual Builders*. The build-time environment enables them to create new procedure types, in the knowledge level. For instance, a new treatment for a disease may be discovered. Likewise, the run-time environment handles procedure execution. Physicians customize a particular running process (procedure instance) according to their needs.

We are familiar with the benefits of integrated build-time and run-time environments. Smalltalk, a powerful object-oriented language, has one of the best programming environments. The development environment is fully integrated with the run-time. A user can interrupt a running program, change the code and resume execution. DOM applications share the same characteristics. The main difference stems from the different audience. Smalltalk programmers want a full-featured programming language. In contrast, workflow users (real estate agents, physicians, etc.) prefer to work within their domain.

7 Summary

This paper describes a framework for adaptive workflow systems. This framework is an example of a Dynamic Object Model, and process models in this framework are similar to object models in more traditional systems. The framework is useful for systems using a Dynamic Object Model, because then workflow integrates closely with the rest of the system, and is easy to implement. But the framework is useful in its own right as a flexible workflow management system that makes it possible to change both process definitions and processes.

References

[AAAM97] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems, 1997. Available on the Web at `http://www.almaden.ibm.com/cs/exotica/wfmsys.ps`.

[AJ98] Francis Anderson and Ralph Johnson. The Objectiva telephone billing system. MetaData Pattern Mining Workshop, Urbana, IL, May 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html`.

[BBG+98] A. Barry, N. Baker, J.-M. Le Goff, R. McClatchey, and J.-P. Vialle. Meta-data based design of workflow systems. OOPSLA Meta-data workshop, Vancouver, BC, October 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/OOPSLA98MetaDataWkshop.html`.

[BMR+96] Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. John Wiley & Sons, July 1996.

[Cha96] James Champy. *Reengineering Management—Managing the Change to the Reengineered Corporation*. HarperBusiness, 1996.

[CJ98] Steinar Cårlsen and Havard D. Jørgensen. Emergent workflow: The AIS workware demonstrator. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[CKL98] Dickson K. W. Chiu, Kamalakar Karlapalem, and Qing Li. Exception handling with workflow evolution in ADOME-WfMS: a taxonomy and resolution techniques. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[DGSZ94] Guido Dinkhoff, Volker Gruhn, Armin Saalmann, and Michael Zielonka. *Entity-Relationship Approach–ER'94, Business Modelling and Re-engineering*, chapter Business Process Modeling in the Workflow Management Environment *Leu*, pages 46–63. Number 881 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[DKÖS98] Asuman Doğaç, Leonid Kalinichenko, M. Tamer Özsu, and Amit Sheth, editors. *Workflow Management Systems and Interoperability*, volume 164 of *NATO Advanced Science Institutes (ASI), Series F: Computer and Systems Sciences*. Springer-Verlag, August 1998.

[DT98] Martine Devos and Michel Tilman. A repository-based framework for evolutionary software development. MetaData Pattern Mining Workshop, Urbana, IL, May 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html`.

[EtH98] David Edmond and Arthur H. M. ter Hofstede. Achieving workflow adaptability by means of reflection. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[Fow97] Martin Fowler. *Analysis Patterns—Reusable Object Models*. Addison-Wesley Object-Oriented Software Engineering Series. Addison-Wesley, 1997.

[FS97] Martin Fowler and Kendall Scott. *UML Distilled—Applying the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, June 1997.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, 1992.

[FY98] Brian Foote and Joseph Yoder. Metadata and active object-models. OOPSLA Meta-data workshop, Vancouver, BC, October 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/OOPSLA98MetaDataWkshop.html`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GT98] Dimitrios Georgakopoulos and Aphrodite Tsalgatidou. *Technology and Tools for Comprehensive Business Process Lifecycle Management*, pages 356–395. Volume 164 of Doğaç et al. [DKÖS98], August 1998.

[HC93] Michael Hammer and James Campy. *Reengineering the Corporation—A Manifesto for Business Revolution*. Harper Business, 1993.

[Hol95] David Hollingsworth. *The Workflow Reference Model*. Workflow Management Coalition, Avenue Marcel Thiry 204, 1200 Brussels, Belgium, 1995. Available on the Web at `http://www.aiim.org/wfmc/`.

[HSB98] Yanbo Han, Amit Sheth, and Christoph Bussler. A taxonomy of adaptive workflow management. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[JB96] Stefan Jablonski and Christoph Bussler. *Workflow Management—Modeling Concepts, Architecture and Implementation*. International Tompson Computer Press, 1996.

[Joh] Ralph E. Johnson. Dynamic object model. Work in progress; available on the Web at `http://st-www.cs.uiuc.edu/users/johnson/DOM.html`.

[JW97] Ralph Johnson and Bobby Woolf. *Type Object*, chapter 4. In Martin et al. [MRB97], October 1997.

[KD98] Mark Klein and Chrysanthos Dellarocas. A knowledge-based approach to handling exceptions in workflow systems. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[Man97] Dragoş-Anton Manolescu. A data flow pattern language. In *Proc. 4th Pattern Languages of Programming*, volume 1, Monticello, IL, September 1997. Available as Washington University Technical Report WUCS–97–34; on the Web from `http://www.uiuc.edu/ph/www/manolesc/`.

[MJ] Dragoş-Anton Manolescu and Ralph E. Johnson. Patterns of workflow management facility. Available on the Web at `http://www.uiuc.edu/ph/www/manolesc/Workflow/PWFMF/`.

[MJ98] Dragoş-Anton Manolescu and Ralph E. Johnson. A proposal for a common infrastructure for process and product models. In *OOPSLA Mid-year Workshop on Applied Object Technology for Implementing Lifecycle Process and Product Models*, Denver, Colorado, July 1998. Available on the Web at `http://www.uiuc.edu/ph/www/manolesc/Workflow/OOPSLA98/`.

[MKVlC98] Theo Dirk Meijler, Han Kessels, Charles Vuijst, and Rine le Comte. Realising run-time adaptable workflow by means of reflection in the Baan workflow engine. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[Moh98] C. Mohan. *Recent trends in workflow management products, standards and research*, pages 396–409. Volume 164 of Doğaç et al. [DKÖS98], August 1998. Available on the Web at `http://www.almaden.ibm.com/cs/exotica/wfnato97.ps`.

[MRB97] Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors. *Pattern Languages of Program Design 3*. Software Patterns Series. Addison-Wesley, October 1997.

[OJ98] Jeff Oakes and Ralph Johnson. The Hartford insurance framework. MetaData Pattern Mining Workshop, Urbana, IL, May 1998. Available on the Web at `http://www.joeyoder.com/Research/metadata/UoI98MetadataWkshop.html`.

[OJW98] Aris M. Ouksel and Jr. James Watson. The need for adaptive workflow and what is currently available on the market—perspectives from an ongoing industry benchmarking initiative. CSCW Towards Adaptive Workflow Systems Workshop, Seattle, WA, November 1998. Available on the Web at `http://ccs.mit.edu/klein/cscw-ws.html`.

[OMG97] Workflow management facility request for proposal. OMG Document cf/97–05–06, May 1997. Available on the Web at `http://www.omg.org/library/schedule/Workflow_RFP.htm`.

[RJ97] Don Roberts and Ralph Johnson. *Evolving Frameworks—A Pattern Language for Developing Object-Oriented Frameworks*, chapter 26. In Martin et al. [MRB97], October 1997. Available on the Web at `http://st-www.cs.uiuc.edu/~droberts/evolve.html`.

[Sch96] Thomas Schäl. *Workflow Management Systems for Process Organizations*. Number 1096 in Lecture Notes in Computer Science. Springer-Verlag, 1996. ISBN 3-540-61401-X.

[VJK96] Vijay Vaishnavi, Stef Joosten, and Bill Kuechler. Representing workflow management systems with smart objects. In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems*, May 1996. Available on the Web at `http://www.cis.gsu.edu/~bkuechle/allsec3.html`.

[WF86] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition*. Addison-Wesley, 1986.

[YB97] Joseph W. Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *Proc. 4th Pattern Languages of Programming*, Monticello, IL, September 1997. Available as Washington University Technical Report WUCS–97–34; on the Web at `http://www.joeyoder.com/papers/patterns/Security/appsec.pdf`.