# Good Architecture = Good (ADL + Practices)

Vincent Le Gloahec[1,3], Regis Fleurquin[2], and Salah Sadou[3]

[1] Alkante SAS, Rennes, France
v.legloahec@alkante.com
[2] IRISA/Triskell, Campus Universitaire de Beaulieu, Rennes, France
regis.fleurquin@irisa.fr
[3] Valoria, Université de Bretagne-Sud, Vannes, France
salah.sadou@univ-ubs.fr

**Abstract.** In order to ensure the quality of their software development process, companies incorporate best practices from recognized repositories or from their own experiences. These best practices are often described in software quality manuals that do not guarantee their implementation. In this paper, we propose a framework for the implementation of best practices concerning the design of the software architecture. We treat the case of architecture design activity because it's the basis of the software development process. Our framework enables on the one hand to describe best practices and on the other hand to check their application by designers. We present an implementation of our framework in the Eclipse platform and for an ADL dedicated to Web applications. Finally, we give an example of use from the context of our industrial partner.

**Key words:** Best Practices, Design, Software Architecture Quality

## 1 Introduction

The software architecture plays a fundamental role in modern development processes. Throughout a project, it can serve as a baseline against which the various stakeholders analyze, understand, build their decisions, and evaluate the software [1]. The languages (Acme [2], xADL [3], ByADL [4], UML as an ADL [5]) used to elaborate architectures highlight concepts (such as connectors, components, etc.) that meet two requirements: (i) be enough expressive to represent all targeted systems, and (ii) allow the architect to focus his attention on key issues such as information hiding, coupling, cohesion, precision, etc. Architecture Description Languages (ADL) direct and sometimes compel the architect to comply with some relevant and universally recognized rules in the target area. Thus, they restrict the form of representable architectures by excluding undesirable ones. The aim is to produce architectures with good quality properties.

However, these languages are designed to allow the representation of architectures that answer various type of needs. Thus, some architectural motifs can be considered useful in some contexts and avoided in others. The quality of architecture is not absolute but is estimated according to each project's requirements (cost, schedule, quality of service, etc.) [6] that sometimes are conflicting. So, the quality of an architecture is the result of a compromise. The language must be tolerant and not unduly restrict the range of possibilities to let free the creativity of architects. Consequently, the use of an ADL alone, as elegant as it may be, can not guarantee obtaining an architecture that meets the quality requirements desired for a given project.

Best language Practices (BPs) found in the literature, such as modeling processes [7] [8], styles [9], patterns [10] and metrics can then provide an essential complementary tool. Based on the specific context of the project, they will help to direct the architect toward the subset of relevant models among those allowed by the language. In this sense, BPs help the architect to limit the area of choice thanks to a language restriction adapted to the project. They help to increase the effectiveness of development in terms of quality and productivity. Additional BPs specific to an application domain, a technology or a managerial and cultural context may also emerge from projects within companies. Properly used in a project, these best language practices constitute the expertise and the value-added of a company. This valuable capital of knowledge guarantees to a company the quality of its architectural models and thus allows to satisfy its customers, to stand out, and to solicit labels and certificates [11]. In other words, to be competitive.

Unfortunately, we show in section 2 of this paper that due to a lack of an adequate formalism to document this knowledge, companies that try to capitalize on this knowledge use informal documents, often incomplete, poorly referenced, and sometimes scattered. This leads to an inadequate and ineffective use and sometimes loss of best language practices. This loss decrease the quality of the designed architectures. We rely for that on a study conducted with an industrial partner that uses a dedicated ADL for Web applications. We propose a language (section 3) and a software platform (section 4) that allow respectively to document and to enact these BPs for any graphical ADL. In this way, we ensure the durability and reuse of knowledge, as well as a constant verification of the application of best language practices. We then show, in section 5, how this language can be used to document some BPs for web applications coming from our industrial partner. In the same section, we show also how these practices can be integrated in their ADL tool (AlCoWeb-Builder). Thus, this helps developers to respect the best language practices defined in their own companies, without changing their working habits. Finally, we describe related work in section 6 before concluding in section 7.

## 2 Problem Statement

In this section, we show the interest for a company to make productive its language practices. We rely on a study undertaken in one of our industrial partners: the Alkante company[4]. We begin by presenting the development environment

[4] Alkante is a company that develops Web applications (www.alkante.com).

(language, tool, best practices) developed by this company for designing the architecture of its applications. Then, we present the difficulties it faces in some of its developments. The analysis of the causes of these difficulties highlights the interest to capitalize and automate best language practices.

### 2.1 Development Environment

In the context of rich Web application development, Alkante has defined an ADL (referred to as AlCoWeb) to help design the architecture of its applications [12]. This ADL is an UML profile. The UML language has been chosen mainly because the version 2.0 of the UML specification contains most of the abstractions needed to design rich Web applications with hierarchical entities. Alkante develops mainly Geographical Information Systems (GIS) with the help of a component-oriented framework composed of PHP and Javascript code artifacts. Thus, when designers define the architecture of their applications, they need to deal with entities such as modules, pages, forms, html controls and raw PHP scripts. In order to manipulate those specific entities in the AlCoWeb ADL, they have been defined as stereotypes dedicated to the specific Alkante's architecture. In this profile, we can found stereotypes such as `«AlkModule»`, `«AlkHtmlForm»`, `«AlkHtmlButton»`, etc.

Based on the AlCoWeb ADL, Alkante has developed a complete model-driven architecture platform called AlCoWeb-Builder. This tool allows the designers to model and assemble Web components to create large applications. Components are designed hierarchically and incrementally using component assemblies and connectors. Once atomic and hierarchical components have been designed, they are made available as components off-the-shelf and can be reused to build larger artifacts, like an authentication form or a geographical Web service for example. AlCoWeb-Builder is a graphical editor, build upon several frameworks of the Eclipse platform. It also comes with a code generation facility. Based on a template system, this tool allows to generate the code of designed Web applications, as illustrated in Fig. 1.

Using this ADL in many projects, the company has identified over the years some language practices. For instance, to ensure a complete code generation, the architecture of Web applications should be very specific. Thus, the quality assurance manager (QAM) of the company has defined a dedicated BP in the form of a complete process, documented in a quality manual.

The core of this BP consists of the following steps:

1. Create one and only one module: a module represents the root container of an architecture. At the implementation level, it corresponds to a physical folder that will embeds the code artifacts of the designed Web application;
2. Create one and only one application container in the module: the module component must contain a single application container. This component is the central piece that represents the business logic of the Web application and also provides services for inter-application communications;

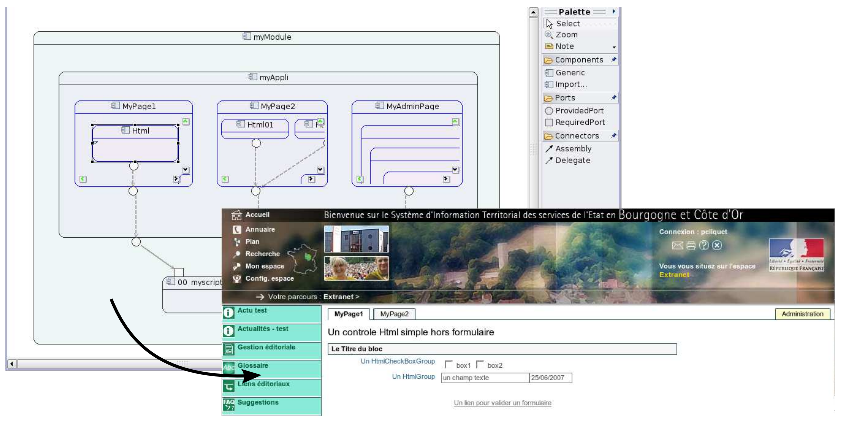


**Fig. 1.** Design example with AlCoWeb-Builder and the resulting Web interface

3. Create pages in the application container: a page component directly maps to a Web page. In the AlCoWeb ADL, pages are considered first class entities for building the presentation tier of Web applications. Consequently, the application container must contain at least one page component for the architecture to be valid. At runtime, pages are responsible for the general layout of their sub-components;
4. Create forms in pages: each page must contain at least one HTML form component. This component is always required to build a valid Web page. This architectural decision has been made because dynamic web pages intensely use forms to submit user's data to a web server;
5. Create HTML controls associated with forms: basic and advanced HTML controls (buttons, lists, calendars) must belong to a form component. For the sake of simplicity in the design of Web applications, all HTML controls without exception must be systematically placed in a form;
6. Create scripts: finally, script components represent code artifacts in charge of the rendering of dynamic Web pages. Scripts can be connected directly to page components, and also use the application container to call services from external applications. Without those components, even if the rest of the architecture is valid, the Web application could not be rendered in a client browser.

The QAM in charge of the definition has added some other BPs in the quality manual. Many of them take the form of modeling rules that need to be checked to ensure the quality of a final architecture. These BPs ensure things such as naming conventions or the way components can be put together.

## 2.2 Recurrent Problems

The MDA approach allows Alkante's team to ease their development effort through components reuse at the model level, and to automate, as possible, code generation and deployment. Although this approach reduces development costs, they observed on occasion that some architectures had not been properly designed.

Some architectures led to errors in the generated code. A major drawback of code generation is that it is difficult to find, from the generated code, the origin of the errors in an architecture. Consequently, developers take a long time to repair. A causal analysis has shown that these errors result from faults made during the component assembly stages when building large applications, while atomic and small hierarchical components are mostly modeled correctly. As AlCoWeb is a hierarchical language, the code generation engine expects the architecture to be designed hierarchically, where HTML forms must be contained in a page, pages must be contained in the root application component, and so on. If this constraint is not respected, the generated application won't be usable. The BP cited above should have guarantee the respect of this constraint. Clearly, this BP has not been correctly applied or not applied at all.

Another recurrent problem concerns the way components must be assembled. All basic and some more advanced HTML controls – more than 30 components, such as buttons, expanded lists, tabs, calendars, etc. – are available as components off-the-shelf. By default, all those components are designed to provide a service named `getHtml()`, which returns the HTML code of the component. At runtime, stacked calls of this service on a hierarchical component allows to produce the complete HTML code of a complex and rich Web page. However, structural components that form the basis of the architecture – e.g. modules, application containers, and scripts – must be designed from scratch by the developers. To design a valid architecture, each `getHtml()` service of `«AlkHtmlPage»` components must be delegated to the parent application container (using a delegate connector from a page's provided port to one provided port of the application component). Then, those provided services must be connected to a required port of a script component using an assembly connector. As for the composition of hierarchical components, the non respect of this specific assembly leads to a poor quality of the architecture that results in the generation of unusable Web applications. Again, this problem results from the non-application or misapplication of a BP, yet documented in the quality manual.

### 2.3 Discussion

The source of the two problems cited above is the non-compliance with some of the BPs outlined in the quality manual. Further causal analysis shown that the root cause has always been one of the followings:

1. Involvement of new designers who did not know when and how the documented practices should be applied. This problem occurs because most of the documented practices do not describe precisely their application context and some of them are ambiguous;
2. Some BPs become fastidious when the architecture complexity grows (for instance, inducing the same manual verification but on numerous model elements), thus developers have ignored or partially applied them;

3. Some BPs are complex and consequently manually error prone (for instance, inducing many verifications on several model elements);
4. When a project is subject to significant time constraints, developers have chosen to ignore some BPs in order to respect the deadline.

To remedy this, we must make productive the BPs. They must be enforced in the used tools (editor, transformations, and code generator). We can try to "hard-code" the BP in the tools suite (if possible by the tool). But, we believe it would not be a good solution. Firstly, tools change over time. We do not want to have to re-code all the BPs each time a tool change. Secondly, BPs evolve too. Each time a BP change, we have to do the corresponding changes in the tool's code. Thus, we must separate the BP definition from the tools.

We advocate that the BPs become first class entities when using an ADL language. Thus, the language will be adapted to fit a particular context (developer, project, company, application domain, etc.). In this way, each company can contextualize a general purpose language to its own needs.

Consequently, to produce quality products, a language should not be reduced solely to its three components (abstract syntax, semantics, and concrete syntax). Indeed, as we emphasized throughout this section, companies often define their own best practices which enrich the language to fit a specific context. The remaining of the paper introduces our approach for the definition and application of best language practices at the early stage of design of software architectures.

## 3 Best Practices Description Language

The language we introduce in this section, called *GooMod*, contains some properties needed for the description of best design practices. In this section we show how we have done to identify these properties and then we describe the abstract syntax and semantic of the *GooMod* language.

### 3.1 Identified Properties

Architectural design is a particular case of modeling activity. However, there is a rich literature on best practices for modeling activity. Thus, we made a survey on best practices in modeling activity in order to identify their characteristics and forms. Through literature we observed three types of best modeling practices: those that are concerned only with the form (style) of produced models, those that describe the process of their design, and those that combine both. As the third type is only a combination of the first two, we limited our study to examples covering the former types. For the first type we found that the best practices for Agile Modeling given in [9] are good examples. In [13], Ramsin and Paige give a detailed review of object-oriented software development methods. From this review we extracted properties concerning the process aspect of BPs. For the sake of brevity, we can't go further on this study in this paper. Interested reader may found more detail on the dedicated Web page[5].

[5] http://www-valoria.univ-ubs.fr/SE/AMGP

Thus, we have identified the following properties:

**Identification of the context:** to identify the context of a BP, the language must be able to check the state of the model to determine whether it is a valid candidate for the BP or not.

**Goal checking:** to check that a BP has been correctly applied on a model, we must be able to check that the status of the latter conforms to the objective targeted by the BP. At the BP description language level, this property highlights the same need as the one before.

**Description of collaborations:** a CASE tool alone is able to achieve some parts of a BP's checking. However, some BP cannot be checked automatically and the tool would need the designer's opinion to make a decision. In case of alternative paths, sometimes the tool is in a situation where it cannot determine the right path automatically. Thus, a BP description language should allow interactions with the designer.

**Process definition:** a process defines a sequence of steps with possible iterations, optional steps, and alternative paths. A BP description language should allow processes to be defined with such constructs.

**Restriction of the modeling language:** several good practices based on modeling methodologies suggest a gradual increase in the number of manipulated concepts (e.g., each step concerns only a subset of the modeling language's concepts). Thus, the BP description language should allow the definition of this subset for each step.

The documentation of a BP associated with a design language requires a description that is independent of any tool; indeed, a BP is specific only to the language. It describes a particular use of its concepts. It should not assume modes of interaction (buttons, menus, etc.) used by an editor in order to provide access to these concepts. Therefore, a BP must be described in a way that can be qualified as a Platform Independent Model (PIM) in Model-Driven Engineering (MDE) terminology (see next section). Ignoring this rule would lead QAM to re-document the BPs at each new version or tool change. The *GooMod* language contains all properties described above and offers a way to document BPs independently of any editor. To introduce the *GooMod* language we present its abstract syntax then its semantic.

## 3.2 Abstract syntax of the GooMod Language

The abstract syntax of the *GooMod* language is given in Fig. 2. The process part of a BP is described as a weakly-connected directed graph. In this graph, each vertex represents a coherent modeling activity that we call a step. Arcs (identified by *Bind* in our meta-model) connect pairs of vertices. Loops (arcs whose head and tail coincide) are allowed, but not multi-arcs (arcs with the same tail and same head).

A step is associated with four elements: its context, its associated design style, the set of language concepts usable during its execution, and a set of actions. The

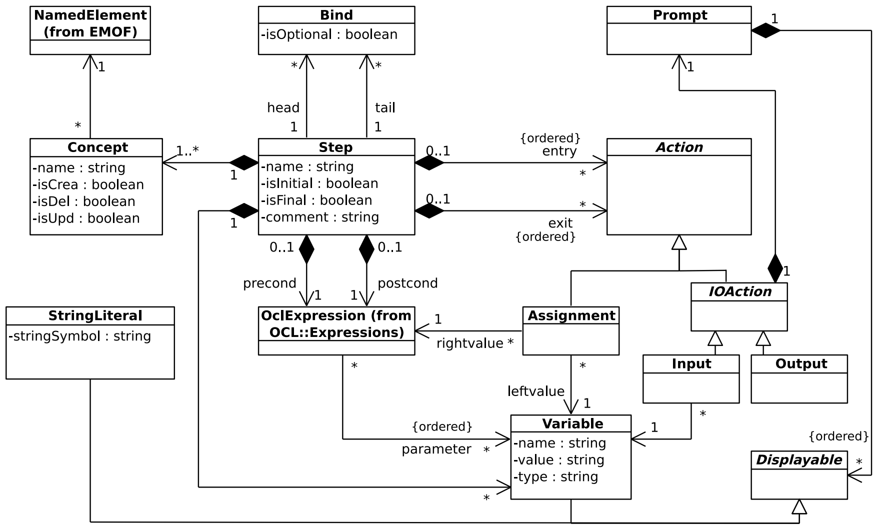


**Fig. 2.** GooMod Meta-model

*context* is a first-order formula evaluated on the abstract syntax graph of the input model before the beginning of the step. We call this formula a *pre-condition*. The design style is a first-order formula that is evaluated on the abstract syntax graph of the current model to allow designer to leave from the step. We call this formula a *post-condition*. The set of the *usable language concepts* is a subset of the non-abstract meta-class of the abstract syntax (described in a MOF Model) of the targeted design language.

Because some BP require the establishment of a collaboration between the system and the designer, we have included the ability to integrate some actions at the beginning (*entry*) and/or at the end (*exit*) of a step. The possible actions are: output a message, an input of a value and the assignment of a value to a local variable. Indeed, at each step, it is sometimes necessary to have additional information on the model that only the designer can provide (goal of *Input* action). Conversely, it is sometimes useful to provide designers information that they can not deduce easily from the visible aspect of the model but the system can calculate (goal of *Output* action). This concerns introspection operations that can be achieved with MOF operators at pre- and post-conditions level. Hence, the usefulness of variables associated with steps to hold results. Thus, actions allow interaction with the designer using messages composed of strings and calculated values.

Steps are also defined by two boolean properties: *isInitial* and *isFinal*. At least one step is marked as initial and one as final in a graph. Finally, an arc can be marked as optional, meaning that its head step is optional.

## 3.3 Semantic of the GooMod Language

Semantically, the graph of a BP is a behavior model composed of a finite number of states, transitions between those states, and some *Entry*/*Exit* actions.

Thus, a BP is described as a finite and deterministic state machine with states corresponding to the steps of the BP's process.

At each step, the elements that constitute it are used as follows:

1. Before entering the step, the pre-conditions are checked to ensure that the current model is in a valid state compared with the given step. Failure implies that the step is not yet allowed;
2. If the checking succeeds, then before starting model edits a list of actions (*Entry Action*), possibly empty, is launched. These actions initialize the environment associated with the step. This may correspond to the initializing of some local variables or simply interactions with the designer;
3. A given step can use only its associated language concepts. In fact, each concept is associated with use type (create, delete, or update).
4. When the designer indicates that the work related to the step is completed, a list of actions (*Exit Action*) will be launched to prepare the step's environment to this end. With these actions the system interacts with the designer to gain information that it can not extract from the model's state;
5. Before leaving the step, the post-conditions are checked to ensure that the current model is in a valid state according to the BP rules.

Leaving a step, several transitions are sometimes possible. These transitions are defined by the *Binds* whose tail is this step. A transition is possible only if the pre-condition of the head step of the concerned *Bind* is verified by the current state of the model. If several next steps are possible, then the choice is left to the designer. A *Bind* can also be defined as optional. In this case, its tail step becomes optional through the transition it defines. Thus, the possible transitions of the tail step are added to those of the optional step, and so on.

## 4 Implementation of GooMod

To implement the *GooMod* language, we developed a complete platform for the management of BPs, starting from their definition at the platform independent model (PIM) level up to their enactment at the platform specific model (PSM) level. Figure 3 illustrates the platform and its PIM-PSM separation. This section describes both levels and their associated tools.

### 4.1 PIM-level: Modeling BPs

The PIM level of the *GooMod* language allows description of BP independently of the used design tool. This level is implemented thanks to the **BP Definition Tool** (see top of Fig. 3). This tool is designed for QAM in charge of the definition of BP that should be observed in a company. Our graphical editor, designed using the Eclipse Graphical Modeling Framework[6] (GMF), allows the representation of BPs in the form of a process. Such a process is represented by a path in a graph.

[6] See Eclipse Modeling Project (http://www.eclipse.org/modeling)

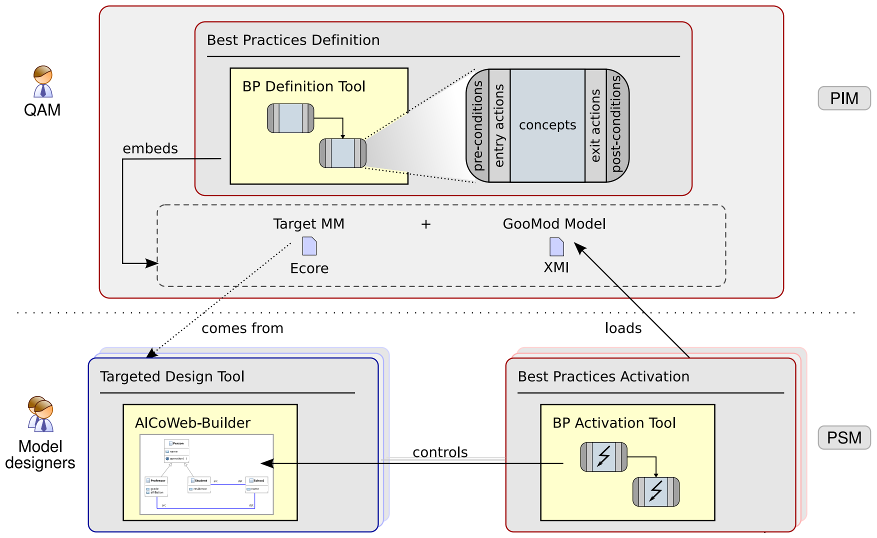


**Fig. 3.** GooMod platform general architecture

Each node of the path is a step. The BP Definition tool uses the meta-model of the target language as its input. At each step of the process, the BP Definition tool allows for the selection of a subset of manipulated concepts from the target language, as well as the definition of a set of OCL pre- and postconditions, and actions before entering and exiting the step.

### 4.2 PSM-level: BPs Enactment

The PSM level aims to attach the definition which is done at the PIM level with a specific design tool. For that, our platform is composed of two parts:

**BP Activation Tool:** that aims to link a BP model defined with the BP Definition tool to a target design tool. It controls the enforcement of the BP process.

**Targeted Design Tool:** which is the end-user design tool where the BP will be performed. This tool is not intended to be modified or altered directly, but will be controlled by an external plugin, which in our case is the BP Activation Tool.

A targeted design tool can be, for instance, the AlCoWeb-Builder tool (see bottom-left of Fig. 3), which allows Alkante's designers to model the architecture of their Web applications. However, our approach is not limited to this design tool. Indeed, the BP Activation tool has been designed to interact with any Eclipse GMF-generated editors. If the first feature of BP Activation is to enact a process and check the elaboration of models, the second feature consists of controlling some parts of the targeted design tool. At each step of modeling, only the editable concepts of the current step are active. Based on the extension capabilities of the GMF framework, the BP Activation tool dynamically activate/deactivate GMF creation tools of the targeted design tool according to the editable concepts allowed for this step. With this approach, we are able to control any GMF editor within the Eclipse platform. To tackle the problem of how to interact with other design tools, we plan to elaborate a mapping meta-model so that QAM could map editable concepts with the creation features (buttons, actions, items) of the design tool.

## 5 Case Study: Alkante's BP

In the following we present how to define the BP presented in section 2 and how to apply it during a design process.

### 5.1 Formal Description of BP

The *GooMod* language allows to represent the different steps of this BP in the form of a process. The steps are described with the help of the BP Definition tool as depicted in Fig 4.

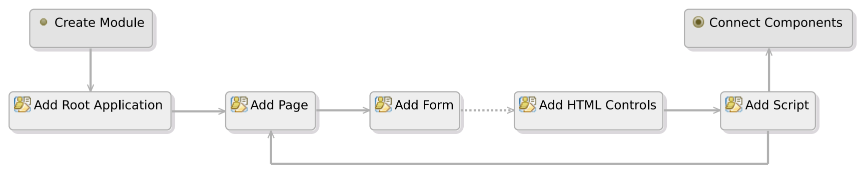


**Fig. 4.** Process part of Alkante's BP for building Web applications architectural models

The process defines an iteration that allows to create multiple pages and their content. Indeed, once a script has been added, the designer will either be able to continue through the process or to iterate by adding new pages. The dashed arrow between "*Add Form*" and "*Add HTML Controls*" represents an optional transition, thus making the latter step optional. This indicates that adding HTML controls to forms is not necessary to produce a valid Web application in this specific context.

The QAM in charge of the definition of this BP is able to detail each step of the process by adding some rules that need to be checked to ensure the quality of the final architecture. For each step, the QAM can define both pre- and post-conditions that will ensure that models will be well-constructed. Those constraints are given using the OCL language. Besides defining constraints, the QAM adds entry and exit actions that allow the designer to collaborate with the system by means of inputs and/or outputs. Those actions are described using a script-like syntax, where it is possible to declare variables and input/output operations. In addition, each step comes with a list of editable concepts that are used to follow the defined process. In the Alkante's BP, each step is associated with a list of meta-classes of the AlCoWeb language: *Component* and *Port* for each step except the last one, and the meta-class *Connector* for the last step (*Connect Components*), thus allowing to connect components with each other.

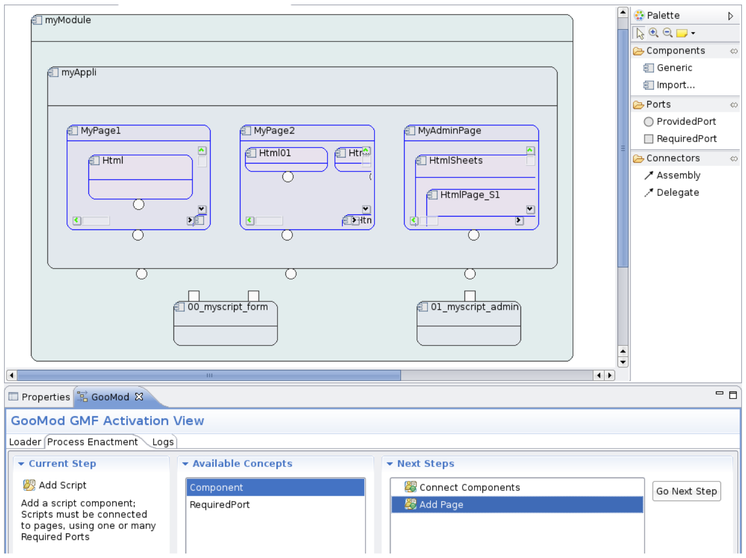
For example, here is a complete description of the step "*Add Form*":

```
     Pre  description: at least one page must be present
          context: Component
          inv: Component.allInstances()->select(c:Component |
                   c.stereotype='AlkHtmlPage')->notEmpty()

   Entry  output("You have to create at least one form per page.")
          output("Make sure to respect the graphical guidelines.")

Concepts  [{"Component","cud"}, {"Port","cud"}]

    Exit  input($response, boolean, "Did you respect the graphical guidelines?")

    Post  description: each form must be contained in a parent page
          context: Component
          inv: Component.allInstances()->select(c:Component |
                   c.stereotype='AlkHtmlForm')->notEmpty()
               and
               Component.allInstances()->select(c:Component |
                   c.stereotype='AlkHtmlForm').owner.stereotype='AlkHtmlPage'
               and
               Component.allInstances()->select(c:Component |
                   c.stereotype='AlkHtmlPage')->forAll(page |
                       page.ownedForms->size() >= 1)
               and OclQuery_graphicalCheck() = true
```

The pre-condition checks that before entering the step, the model contains at least one page. The entry action is used to inform the designer about the constraint related to this step (at least one form per page) and recommendation about graphical guidelines. The syntax used to describe editable concepts is given in the form of two strings: the first is the name of the concept, the second is composed of the first letters of the authorized behaviors. In our case, "cud" means "create, update and delete". In the above example, the exit action is used to ask the designer to check whether the graphical guidelines are respected. The post-condition is used to check that the model contains at least one form per page and that the graphical guidelines were respected. In the BP Definition tool, all the rules listed above are editable using advanced editors and content assistants, so that designers don't have to manipulate the syntax given in this example.

## 5.2 BP in Action

When developers starts designing architecture models with AlCoWeb-Builder, they first load the *GooMod* model defined by the QAM, and then launch the controlled editing process. The bottom of Fig. 5 shows the current state of the BP Activation tool: the current design step is "*Add Script*" (on the left), allowed editable concepts for this steps are *Component* and *Port* (in the middle), and



Add a script component; Scripts must be connected to pages, using one or many Required Ports

**Fig. 5.** Applying best practices in AlCoWeb-Builder

the right part shows the next available steps. As we can see, the BP indicates that we have the choice to go back to the "*Add Page*" step, or to go ahead to the last step of the process to finish the design of the Web application.

In the company, the *GooMod* platform is used differently depending on skills and experience of developers. Novice developers systematically use the platform, whereas experts prefer to make verifications at key steps of the design process. Indeed, novices are not fully aware of all the practices that have to be respected to produce a quality architecture, therefore they prefer to be guided through the whole design process. This reinforces our idea that a tool must be flexible enough to adapt to the most users. The *GooMod* platform has been designed accordingly.

The reader may find other examples of use of the GooMod platform through screencasts at http://www-valoria.univ-ubs.fr/SE/goomod.

## 6 Related Work

Best practices management is a particular case of Knowledge Management. This domain aims to identify, gather and capitalize on all used knowledge (including BP) to improve companies performance [14]. Thus, in the domain of BP for software development, there are three types of works: those interested in BP archiving, those interested in their modeling, and those who seek their implementation directly in the development tools.

Several works suggest introducing processes and tools that facilitate storage, sharing, and dissemination of BP within companies (e.g. [15], [16]). They advocate in particular the use of real repositories allowing various forms of consultation, thus facilitating research and discovery of BPs. However, the BP referred to by these systems are documented and available only through textual and informal. It is therefore impossible to make them productive in order to control the use within CASE tools. To our knowledge, there is no other work on the definition of rigorous languages for documenting BPs. However this field can benefit from works concerned with method engineering ([7], [8]) and software development process ([17]). Indeed, a BP is a particular form of development method. It imposes a way to run an activity, a process, and also imposes a number of constraints on the proper use of language concepts. Language engineering and its definition tools are therefore very useful.

With CASE tools, several works suggest encouraging, even requiring, the respect of certain BP. The domain that had produced good results in recent years is the one that focuses on the automation of BP concerning detection and correction of inconsistencies. These include, in particular, the work presented in [18], [19], [20] and [21]. They propose adding extensions to modeling tools, such as Eclipse or Rational, that are able to intercept the actions of designers and inspect the information system of the tools in order to detect the occurrence of certain types of inconsistency. The inconsistencies treated by these works are various, but they remain on the analysis of syntactical consistency of models expressed in one or more languages, which is already quite complex. Sometimes they address the problem of what they call methodological inconsistencies, i.e., the detection of non-compliance with guidelines relating to how the language should be used. However, these works involve BP with a much smaller granularity than those we are dealing with.

In the domain of software architectural modeling, Acme has been proposed as a generic ADL that provides an extensible infrastructure for describing, generating and analysing software architectures descriptions [2]. This language is supported by the AcmeStudio tool [22], a graphical editing environment for software architecture design implemented as an Eclipse Plug-in. AcmeStudio offers the possibility to define rules (invariants and heuristics) to check whether an architectural model is well formed. However, rules have to be defined directly at the design stage and are embedded in the architectural model. This limits the portability of the solution to another tool and the expressiveness of BPs. In our approach, we prefer the definition of such rules to be at the PIM level, so that they can be reused and remain independent of any tool. The *GooMod* platform can be easily adapted to work with AcmeStudio, since this tool is an Eclipse-based graphical editor. In this way, it could propose features not available in AcmeStudio: support for a process representation of the design activity, better understanding of the ADL through the manipulation of only valuable concepts at each design steps, and ways to collaborate dynamically with designers.

## 7 conclusion

To produce softwares with high quality, a company must first ensure that its architecture is of high quality. To achieve a desired level of quality, the use of an ADL alone, as elegant as it is, is not enough. It should be used with best practices to get good solutions depending on the context of use. Through the use of best practices, designers avoid reproducing well-known errors and follow a proven process. But the quality has a cost related to two aspects: the capitalization of these best practices and roll-backs in case of non compliance with them.

With our approach, quality assurance managers are able to define, in a formal description, their own design practices based on books, standards and/or their own gained experience. Since these descriptions are formal, they become productive in tools. They can be automatically applied by designers to produce high quality architectures. Thus, we provide not only a way to capitalize best practices, but also a means to check their compliance throughout the design process to avoid costly roll-backs.

Our approach provides to architects, directly in editing tools, a collection of BPs. This automation relieves the architects of much of manual verifications. Consequently, they do not hesitate to activate them when needed. They can also choose the BPs to use, depending on the given context, their own skills and type of the project.

As a continuation of this work, we plan to provide a best practice management tool that allows the quality assurance manager to optimize BPs use. In addition of defining BPs, this tool should help to involve designers in projects (process + human), with management of access rights and temporary permissions of violation. Finally, it must allow the generalization of individual BPs to make them usable by all designers. This last point will enable the company to go up from the level of individual know-how to the level of collective know-how.

## References

1. Erdogmus, H.: Architecture meets agility. IEEE Softw. **26**(5) (2009) 2–4
2. Garlan, D., Monroe, R.T., Wile, D.: Acme: architectural description of component-based systems. (2000) 47–67
3. Dashofy, E.M., Hoek, A.v.d., Taylor, R.N.: A comprehensive approach for the development of modular software architecture description languages. ACM Trans. Softw. Eng. Methodol. **14**(2) (2005) 199–245
4. Di Ruscio, D., Malavolta, I., Muccini, H., Pelliccione, P., Pierantonio, A.: Developing next generation adls through mde techniques. In: 32nd International Conference on Software Engineering (ICSE 2010) to appear. (2010)
5. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Trans. Softw. Eng. Methodol. **11**(1) (2002) 2–57
6. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1998)
7. Henderson-Sellers, B.: Method engineering for OO systems development. Commun. ACM **46**(10) (2003) 73–78
8. Gonzalez-Perez, C., Henderson-Sellers, B.: Modelling software development methodologies: A conceptual foundation. Journal of Systems and Software **80**(11) (2007) 1778 – 1796
9. Ambler, S.W.: The Elements of UML(TM) 2.0 Style. Cambridge University Press, New York, NY, USA (2005)

10. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented software architecture: a system of patterns. John Wiley & Sons, Inc., New York, NY, USA (1996)
11. Gratton, L., Ghoshal, S.: Beyond best practices. Sloan Management Review (3) (2005) 49–57
12. Kadri, R., Tibermacine, C., Le Gloahec, V.: Building the Presentation-Tier of Rich Web Applications with Hierarchical Components. In: International Conference on Web Information Systems Engineering (WISE'07), Springer (2007) 123–134
13. Ramsin, R., Paige, R.F.: Process-centered review of object oriented software development methodologies. ACM Comput. Surv. **40**(1) (2008) 1–89
14. Stewart, T.A.: The Wealth of Knowledge: Intellectual Capital and the Twenty-first Century Organization. Doubleday, New York, NY, USA (2001)
15. Fragidis, G., Tarabanis, K.: From repositories of best practices to networks of best practices. Management of Innovation and Technology, 2006 IEEE International Conference on (2006) 370–374
16. Zhu, L., Staples, M., Gorton, I.: An infrastructure for indexing and organizing best practices. In: REBSE '07: Proceedings of the Second International Workshop on Realising Evidence-Based Software Engineering, IEEE Computer Society (2007)
17. OMG: Software Process Engineering Meta-Model, version 2.0 (SPEM2.0). Technical report, Object Management Group (2008)
18. Biehl, M., Löwe, W.: Automated architecture consistency checking for model driven software development. In: QoSA '09: Proceedings of the 5th International Conference on the Quality of Software Architectures, Springer-Verlag (2009) 36–51
19. Egyed, A.: Uml/analyzer: A tool for the instant consistency checking of uml models. Software Engineering. ICSE 2007. 29th International Conference on (2007) 793–796
20. Hessellund, A., Czarnecki, K., Wasowski, A.: Guided development with multiple domain-specific languages. In: MoDELS. (2007) 46–60
21. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, ACM (2008) 511–520
22. Schmerl, B., Garlan, D.: Acmestudio: Supporting style-centered architecture development. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 704–705