JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 6, November-December 2003

How to increase your business reactivity with MDA / UML ?

Patterns and Frameworks for synchronizing IT with the changing business environment

Birol Berkem, GOObiz / CNAM, Paris - France

Abstract

For the last few years, companies have tried to develop their software systems with use case driven development processes. This practice brings many benefits by allowing you to concentrate your analysis and design efforts on the usage dimension of a system.

However, modeling a system with **only** use-case driven UML specifications does not allow good levels of **business reactivity** (= response time necessary for a business system for implementing changes as required by its controlled process of adaptation to its environment). Implementing requested changes as a reaction to new requirements for time-to-market is still very much a challenge for organizations.

In this sense, we have experienced issues related to:

- **Lacks of flexibility in specifications**
- **The Gap between business and application layers**

that render the evolution of systems hazardous and in consequence the business very slow to react to changes !

Indeed, without respect of patterns enabling flexible, executable and traceable specifications, UML practitioners fall in some kind of "spaghetti oriented development" that makes the evolution of their system difficult.

We explain below reasons of these weaknesses and their particular impact on the business reactivity. In sections 2, 3 and 4 we introduce six Goal-Driven Development Patterns for preventing these issues. These patterns assure platform independence, portability and reusability of modeled specifications as required by the OMG's Model Driven Architecture (MDA[1]) [Ref: http://www.omg.org/mda]. Finally, section 5 presents a summary of the Goal-Driven Development Framework whose UML artifacts elaborated using these patterns are traceably linked in order to ensure good levels of reactivity to changes.

[1] MDA and UML are trademarks of the OMG (Object Management Group)

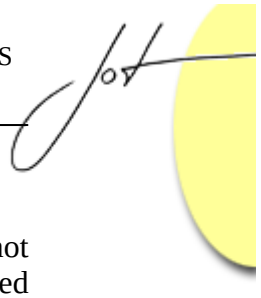
Cite this article as follows:Birol Berkem: “How to increase your business reactivity with UML/MDA”, in *Journal of Object Technology*, vol. 2, no. 6, November-December 2003, pp. 117-138. http://www.jot.fm/issues/issue_2003_11/article4

1 INTRODUCTION: WHAT ARE REASONS AND IMPACTS OF WEAKNESSES IN THE BUSINESS REACTIVITY WITH UML?

Factors that impact negatively the reactivity of systems to changes are mainly due to the lacks that concern evolution of specifications, to their absence of traceability from requirement analysis toward software implementation and also to the gap that persists between the business and the application system layers. A brief insight for each one of these development issues is given below :

- **Lacks of Evolutivity in Specifications**: Specifications are not rendered identifiable in the UML diagrams. Indeed, in general, a given requirement references operations implemented in more classes. As a consequence of this orthogonality between requirements and classes, it is not easy for analysts to specify evolution of requirements in face of changes and for designers to implement required "corrections". In order to specify evolution of requirements within their kind of nominal and alternate realization scenarios, specifications need to be designed as identifiable classes and components like objects and their behaviors!

- **Specifications are not rendered traceable from requirement analysis toward implementation**: Specifications are not rendered traceable toward lower abstraction levels in the development. For example, behaviors defined at the analysis level of business or application layers are not used at the design level with respect to their original description. This inconvenience is essentially due to the assignment of functional responsibilities to domain objects prematurely at the analysis level. Indeed, at the design level, designers have to retouch these specifications with their architectural choices. For example, in the case of a sale transaction, an object like *ticket* that is specified to be created and printed at the end of the transaction at its analysis description, may be suppressed and replaced at the design level by the *sale* object that implements this function by a *print()* method. Similarly, in the context of an application for project supervision, at the analysis level *resources* may be specified as directly controled by the *project* object in their assignment scenario. At the design level, another controller like *resource-manager* could be asked to manage assignments for these *resources* instead of *project*. Finally, the dependence of the design level specifications from constraints of a given technological target platform presents another inconvenience for the business reactivity in face of the frequency of technologic changes. To prevent this factor, analysis and design specifications need to be rendered executable independently from any target platform (using PIM - Platform Independent Model in MDA) and traced by transformation to any Platform Specific Model (PSM) that focus on code generation for a specific platform [*Ref: MDA*].



- **Lacks in the Traceability between Abstraction Layers**: Specifications are not traceable between the business and the application layers. For example, targeted behaviors that are defined for a requirement in the business process layer are not usable in the application layer. As a direct impact of this lack of traceability, analysts are not encouraged to formalize business reactions in separate models: both business rules and their usage constraints are mixed inadvertently in application use cases. This impacts negatively the validation process of use cases and the evolution of business rules: use cases descriptions are often rendered very long, evolution of business rules they utilize become difficult! The figure below shows requirements-gathering process by use cases in the business and in the application system levels. It also highlights needs for traceability between requirements captured in the business level toward the application system level.

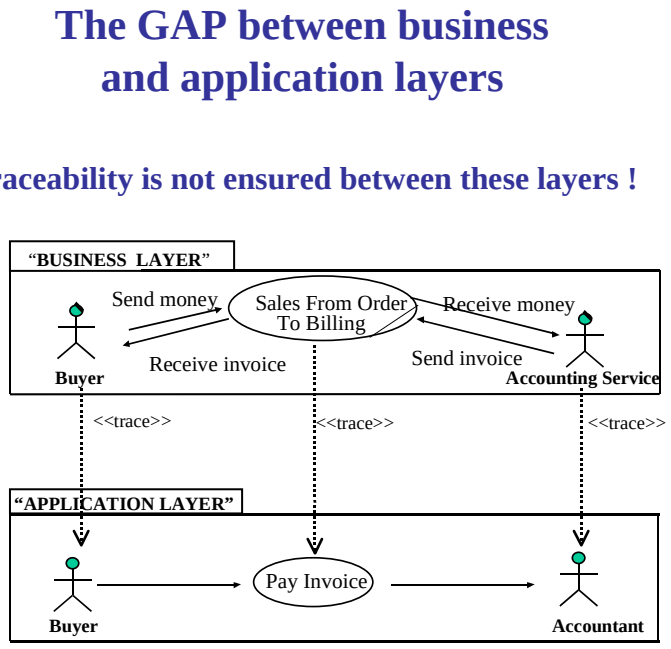


Figure 1: Applications cannot efficiently react to changes specified in the business layer. Because as business specifications are not sufficiently structured in the business layer, they are not efficiently traced toward the application layer.

Goal-Driven Development Patterns presented in the next section permit to avoid modeling issues introduced above.

2 PATTERNS FOR INCREASING BUSINESS REACTIVITY WITH MDA AND UML

Business systems need to react swiftly and accurately to changes that occur in their environment. In this adaptation process, in order to prevent issues that we have seen above, specifications that constitute behaviors of such systems need to be rendered:

- identifiable like objects and components but with flexible behaviors (easy to change) in order to confer maintenable evolution to specifications,
- traceable from the requirement analysis level toward their implementation,
- platform independent for avoiding to duplicate effort in design with choices related to this level and to ensure validity of analysis specifications independently from constraints of a given technological target platform,
- executable even at the analysis/design level independently from any technological target platform for assuring early tests (to ensure correct understanding of requirements) and completeness of specifications in order to transform them directly in the language of any target platform (**portability**),
- traceable between business and application layers to allow applications (use cases) invoke correct business behaviors as they are defined at the business process layer where they evolve according to strategic decisions (**reusability**).

In order to build specifications with these properties, we have identified six patterns that constitute the **backbone for the development of such an agile business system**. The first group of three patterns presented in section 3 **ensure flexibility in specifications**. The last three ones presented in section 4 **are designed for closing the gap between business and application layers**.

3 PATTERNS FOR CONFERING FLEXIBILITY TO SPECIFICATIONS

The first group of three patterns are intended to provide an easier maintenance to specifications throughout evolution of the system. These patterns are: Identifiable Specifications (PIS), Evolutive (flexible) Specifications (PES) and Executable Specifications (PEX).

A summary of these patterns and dependencies between them are presented in the schema below:

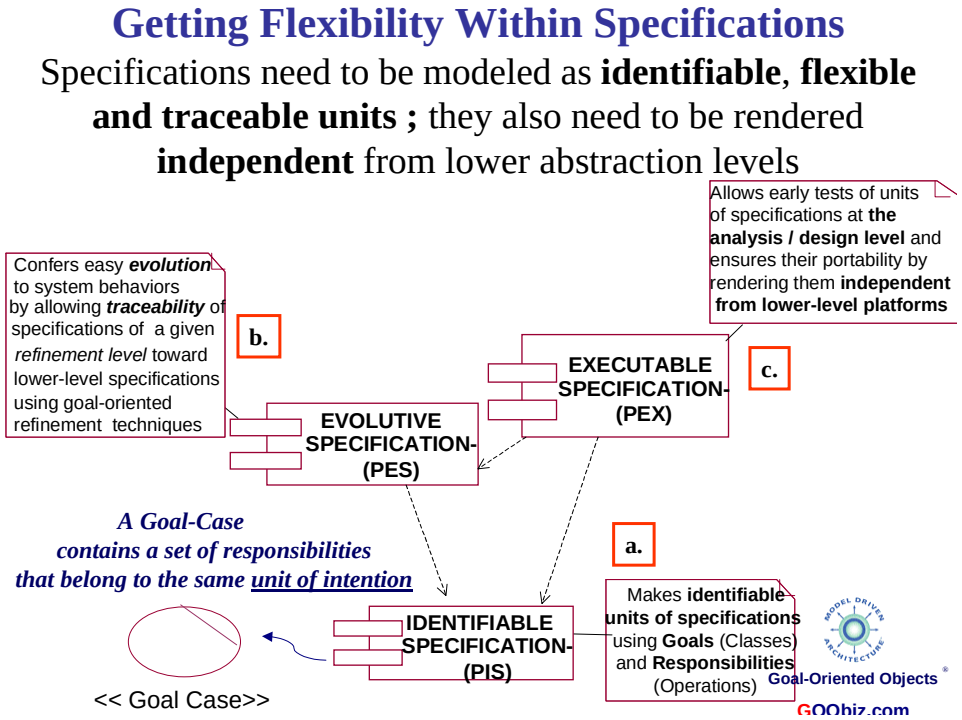


Figure 2: Dependency relationships between patterns that assure easier evolution to system specifications

a) Pattern for Identifiable Specifications (PIS)

Intent: Making Identifiable Specifications

Solution: Make identifiable specifications by capturing requirements within goals and responsibilities that are meaningful within these goals.

Explanation: Requirements that belong to the same functional context -or unit of intention- are grouped in **goal-cases**. Reifying a goal-case as a Goal-Oriented Object (GOO) and encapsulating responsibilities as operations of this GOO class allow related behaviors to become identifiable within their corresponding goal structure.

A GOO may be modeled in UML using the notation of an object-in-state ; so it can be described by the object name followed by the state of this object, in brackets. For example, *Visitor [Registration]* represents a GOO class which deals with the registration state of *Visitor*. The state of a GOO covers activities that must be executed within its *execution -functional- boundaries*. In the case of *Visitor [Registration]*, these execution boundaries cover operations that are meaningful in the registration state of the *Visitor*.

So, a GOO can react to a request, only if this request can be kept inside its functional boundaries as a responsibility (operation). For example, such a GOO like *Visitor [Registration]* can react exclusively to requests related to the registration process of a

visitor, that are implemented inside its functional boundaries by the corresponding operations.

Operations of a GOO class can also be discovered via an activity diagram elaborated for the related goal case. Nominal and alternate sequences of actions that are encountered during the realization of a goal-case become operations of the corresponding GOO class. So, an operation specified in a GOO class may play a role of *machinery*, *exception or post-action* depending on the sequence of actions it represents in the achievement of the related goal-case. Such operations constitute *contextual operations* of a GOO class as they are fired under the control of the *controler operation* of the corresponding GOO which does supervise their execution. For example in the figure 3 below, *register_visitor()* is the controler operation of *Visitor [Registration]*.

Finally, constraints related to a goal represent values that must be guaranteed or targetted by appropriate operations of the corresponding GOO. They can be appended to the related class name as UML tag-values or constraints, or if necessary using a UML note.

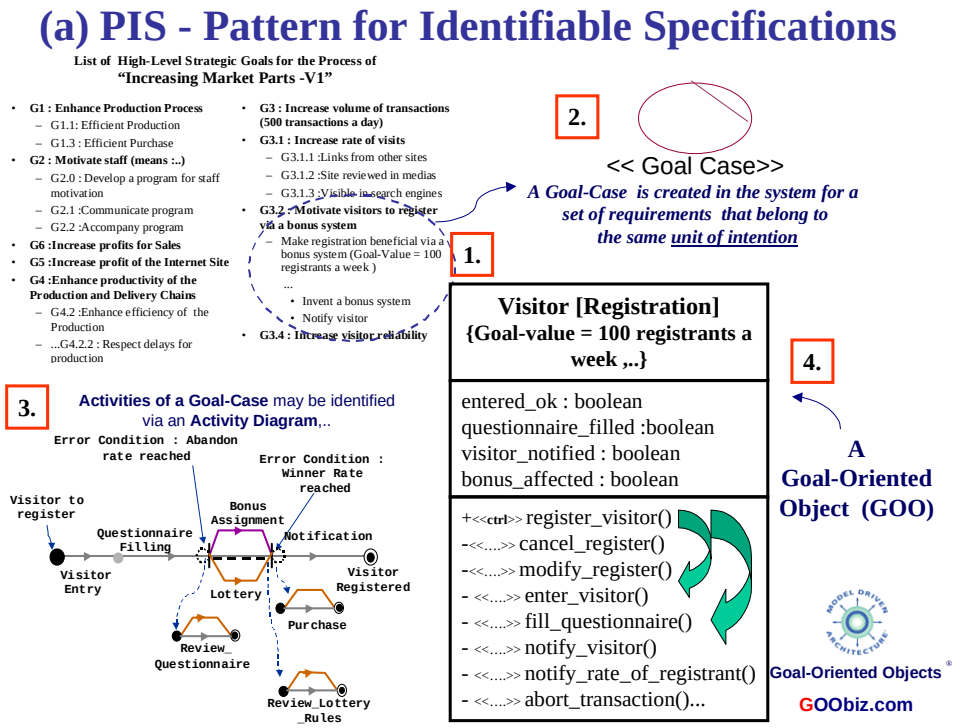


Figure 3: *Visitor [Registration]* is shown as a class of Goal-Oriented Object (GOO class) that specifies on its own necessary collaborations to ensure requested behaviors

b) Pattern for Evolutive (flexible) Specifications (PES)

Intent: Confering easy evolution to specifications by allowing refinement of complex responsibilities of the system (operations of GOO classes) and by ensuring their traceability.

Solution: Refine complex operations of a GOO class using nested GOO classes and their operations. Traceability between a base operation and operations of its nested class is automatically ensured by invocation.

Explanation: In order to make flexible specifications, we need specify nominal and alternate actions in the achievement of complex responsibilities. Indeed, if a responsibility is complex for an abstraction level and necessitates to be identified separately for its evolution purpose, it then requires to be considered as a new GOO at this level. For example, in the case of an ATM machine, for the sentence "eject the card when the transaction is completed", eject card may be designed first as a responsibility in the context of the transaction. So, it can be considered at the analysis level by the operation *eject_card()* as part of the goal *Transaction [Realization]*. But at the design level, this operation may require to be refined by other technical responsibilities ; for instance, it can be refined by adding a new GOO class *Card [Ejection]* to the system that incorporates technical operations related to the card ejection process.

Thus, the pattern confers evolution to complex operations of GOO classes by refining them via other GOO classes and operations. Traceability between a base operation and its corresponding refinements is automatically ensured by invocation. As a result of this transformation process, GOO classes that emerge by refinement constitute contextual classes of their parent class. They correspond to physical or referenced parts of the corresponding base class and constitute with the latter a composite of GOO classes (GOO_Comp).

A GOO_Comp may include physically and can import other GOO classes or components (GOO_Comps) as its contextual parts.

Figure 4 illustrates a GOO_Comp with its nested GOOs that play same role as the corresponding base operation they refine.

Figure 5 shows refinement of certain operations of the GOO class *Visitor [Registration]* using nested GOO classes that become part of the emerging GOO_Comp.

> *Remark : Assigning durable names to goal structures according to the abstraction level in which they are placed, constitutes a fundamental step for ensuring a coherent evolution (pattern PCE presented next) to the system by the use of common components. Such an hierarchical framework of goals permits the overall system to evolve in harmony with the evolution of its environment.*

A Component of Goal-Oriented Objects (GOO_Comp) :

Task oriented set of GOOs that collaborate to achieve a goal

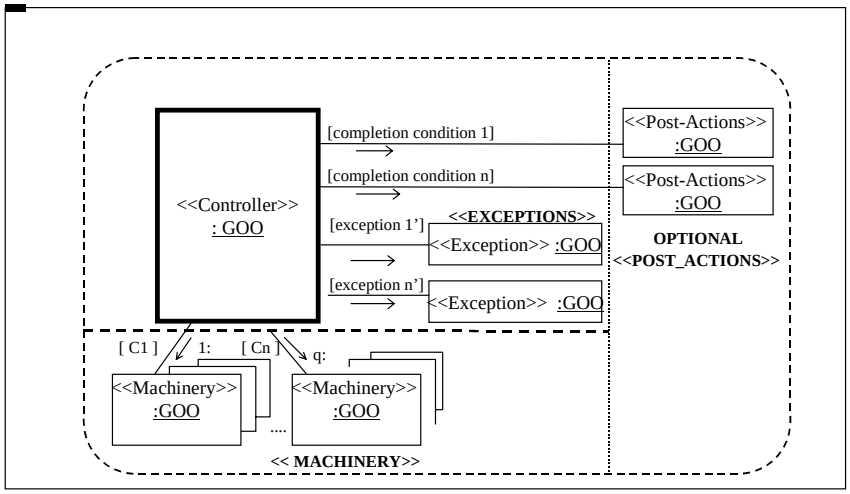


Figure 4: GOOs act as Machinery, Exception or as Post-Actions in the achievement of the responsibility of their controller

(b) PES - Pattern for Flexible and Traceable Specifications

•Provides explicit description of complex behaviors and assures their traceability

Each complex operation may be identified as a separate GOO within a **GOO_Comp**

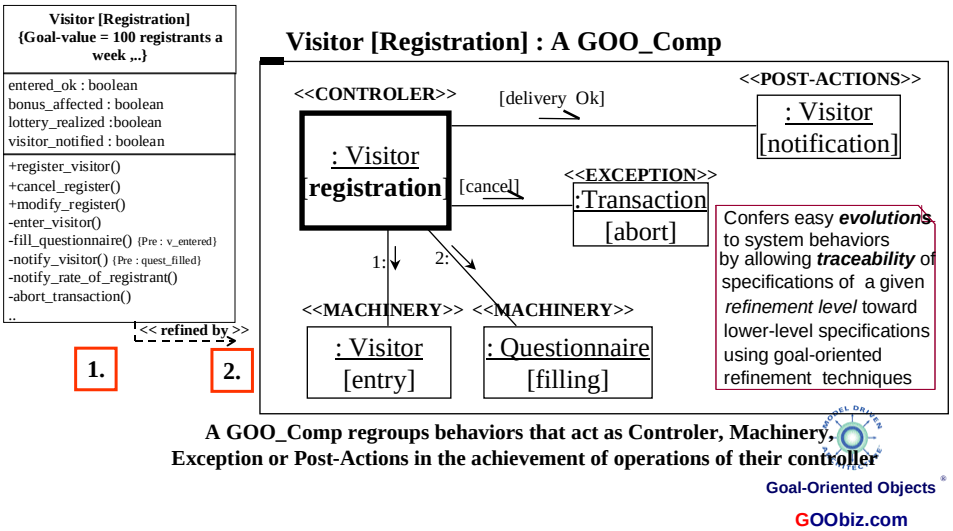


Figure 5: *Visitor [Registration]* is shown as a Component of Goal-Oriented Objects (GOO_Comp) where GOO classes refine complex operations specified within their controller.

c) Pattern for Executable Specifications (PEX)

Intent: Testing specifications at the early analysis/design level independently from specificities of the target platforms and ensuring their portability "as is" on the target platform (Specifications elaborated at the analysis level shouldn't be modified in the design level, those of the design level shouldn't be modified at the implementation).

Solution Step 1: Render specifications independent from lower abstraction levels. To do this, use appropriate goal structures that keep their validity from the analysis level throughout lower abstraction levels.

Solution Step 2: Ensure completeness of analysis/design specifications in order to render them executable "as is" on the target technological platform.

Explanation step 1:

To confer platform independence to analysis/design level specifications from the target technological platform, we assign boundaries to these development levels as follows :

- System analysis/design levels at the technical platform (PIM in MDA) focus on the ***technical what*** **and** ***how***,
- Technological analysis/design levels focus on *the* ***technological what*** **and** ***how*** at the target platform (PSM in MDA).

Figure 6 shows abstraction boundaries assigned to technical and technological platforms and correspondances of these platforms with the PIM and PSM levels of the Model Driven Architecture (MDA).

Within the technical platform, as we have talked about in the previous section, responsibilities assigned to entity-objects at the analysis level **are often altered** by design choices in the design level.

To prevent the invalidation of analysis specifications later by design choices, we need designate controllers independently from entity-objects. Thus, by choosing ***business and application goals structures*** as **controllers** respectively in collaborations of the business and of the application layers, we keep validity of the related analysis specifications "as is" toward design levels, and so on.. (i.e. without modifications of originally specified behaviors at the lower abstraction levels).

The class diagram (figure 7) shows specifications of the technical what (at the analysis level) and of the technical how (at the design level) on the technical platform (PIM). This separation of concerns *permits to test analysis specifications independently from the technical how of the design level* and does ensure *validity of these specifications at the design level.*

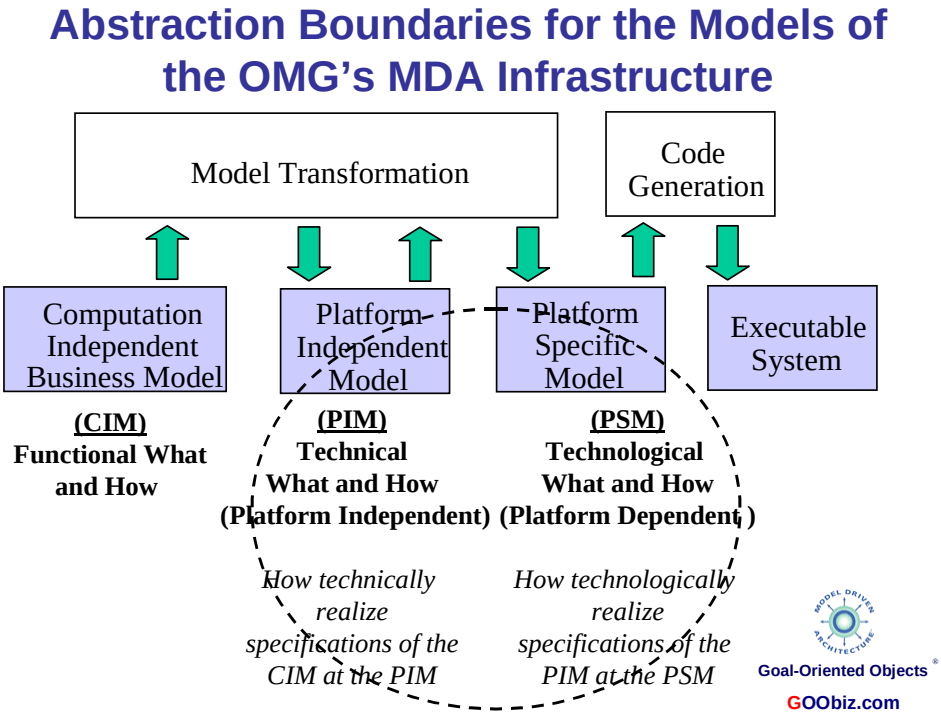


Figure 6: Assignment of abstraction boundaries to technical (PIM) and technological platforms (PSM) for assuring independence of related specifications

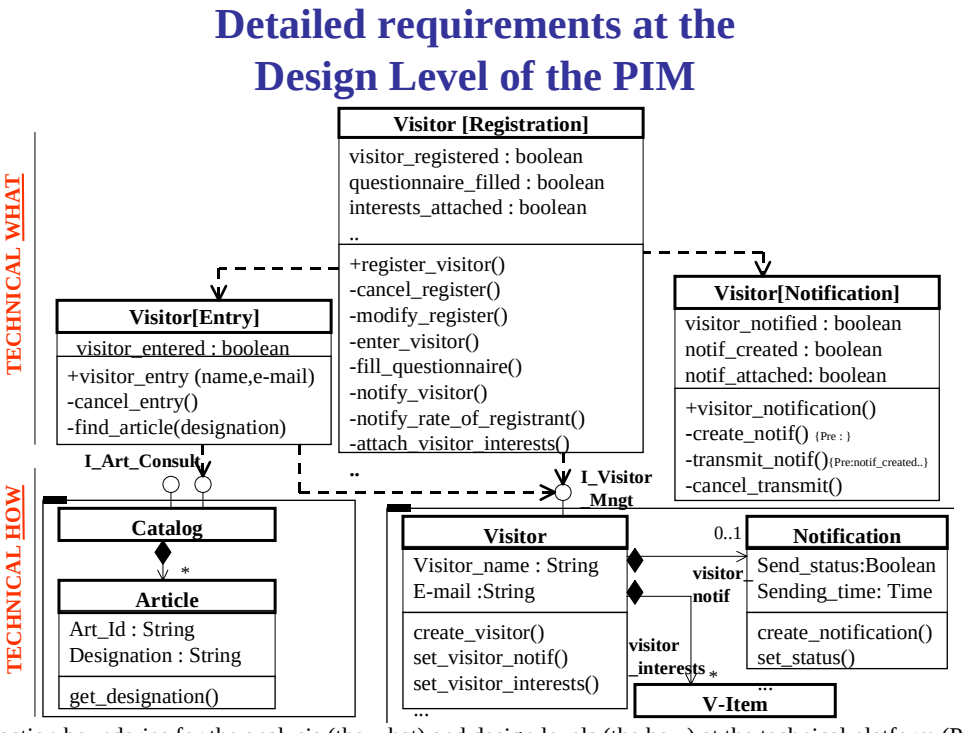


Figure 7: Abstraction boundaries for the analysis (the what) and design levels (the how) at the technical platform (PIM) for assuring independence of related specifications

Explanation Step 2:

In order to render analysis / design specifications executable "as is" on the target technological platform (PSM), we need ensure completeness of these specifications at the technical platform (PIM) by executing them independently from the target platform.

Platform independent specifications (PIM level in the MDA) can be executed by goal, starting at the analysis level. Operations are triggered by their controler after comparing their pre-conditions to the state of the system expressed by attribute values.

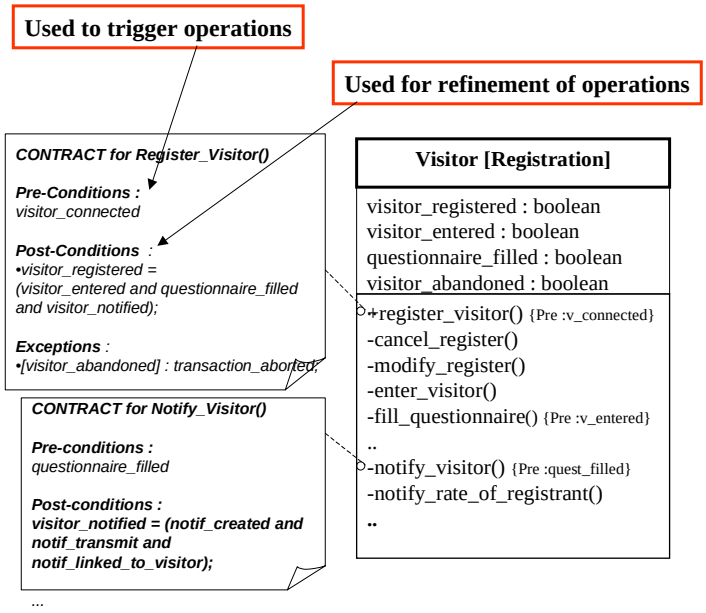


Figure 8: Pre-conditions specify conditions for triggering operations

Completeness related to the execution of an operation is supported by the refinement (decomposition) process of its **post-conditions**. Post-conditions of operations are refined there -if possible, assisted by a graphical tool- until CRUD (Create, Retrieve, Update, Delete) functions that permit to handle entity objects, their attributes and links between these objects are reached (see figure 9).

Post-conditions specified for an operation permit to discover at the immediate lower refinement level operations and attributes of the nested GOO class that support these post-conditions. These operations are used by the controler operation of this nested GOO class in order to realize requested post-conditions.

The class diagram below shows refinement of the operation *notify_visitor()* of *Visitor [Registration]* by operations of *Visitor [Notification]* that respect its post-conditions. Operations like *create_notification()* and *attach_visitor()* that should belong

to this list have not been shown for their granularity reason. Instead, the code related to these operations are implicitly incorporated within the body of *visitor_notification()*.

The body of *visitor_notification()* shows actions that have to be executed to reach specified post-conditions. Attributes that are derived from post-conditions act as triggers for these operations.

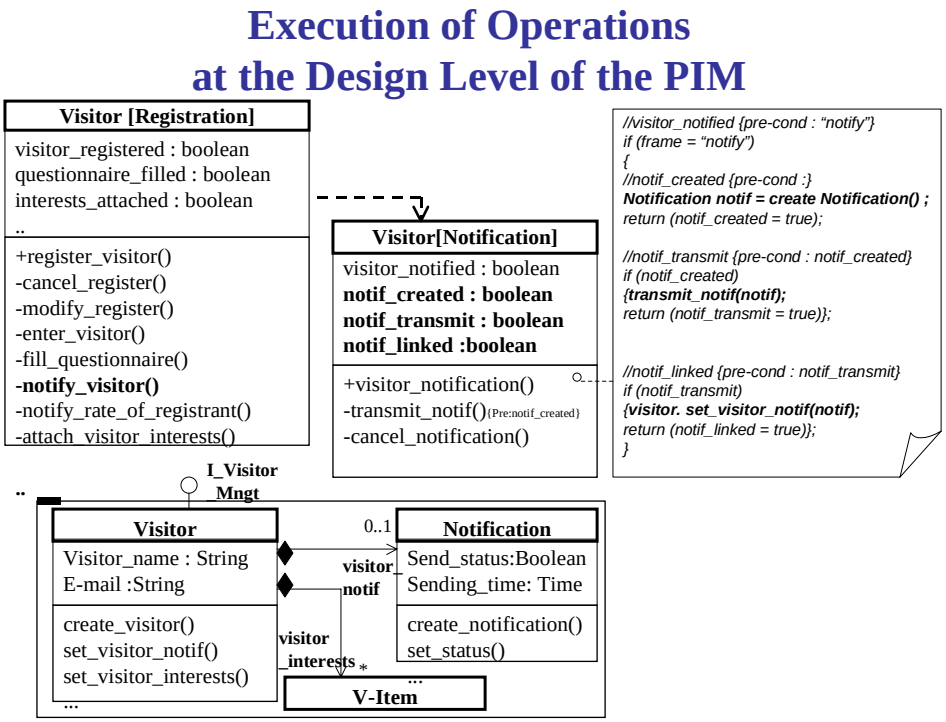


Figure 9: The body of visitor_notification() shows actions that have to be executed to reach specified post-conditions. Attributes that are derived from post-conditions act as triggers for these operations.

As a conclusion for the pattern PEX, testing specifications at the early analysis and design levels as well as rendering them executable "as is" on the target platform bring flexibility to specifications. Indeed these factors ensure respectively:

- early understanding of requirements without waiting for the target platform to be ready for testing them and without necessary technological knowledge,
- portability of specifications whatever changes arising on both functional and technological sides.

4 PATTERNS FOR CLOSING THE GAP BETWEEN BUSINESS AND APPLICATION LAYERS (WITH A COHERENT ADAPTATION TO CHANGES)

The second group of three patterns aims to closing the gap between the business and application layers as well as adapting system coherently to changes. Changes that are captured in the business environment impact appropriate components of the system ; they are propagated through the application layer to synchronize IT actors and applications with the planned business behaviors.

These patterns are: Traceable Abstraction Levels (PTAL), Use Business Behaviors (PUB-BAL) and Coherent Evolution (PCE).

A summary of these patterns and dependencies between them are presented in the schema below:

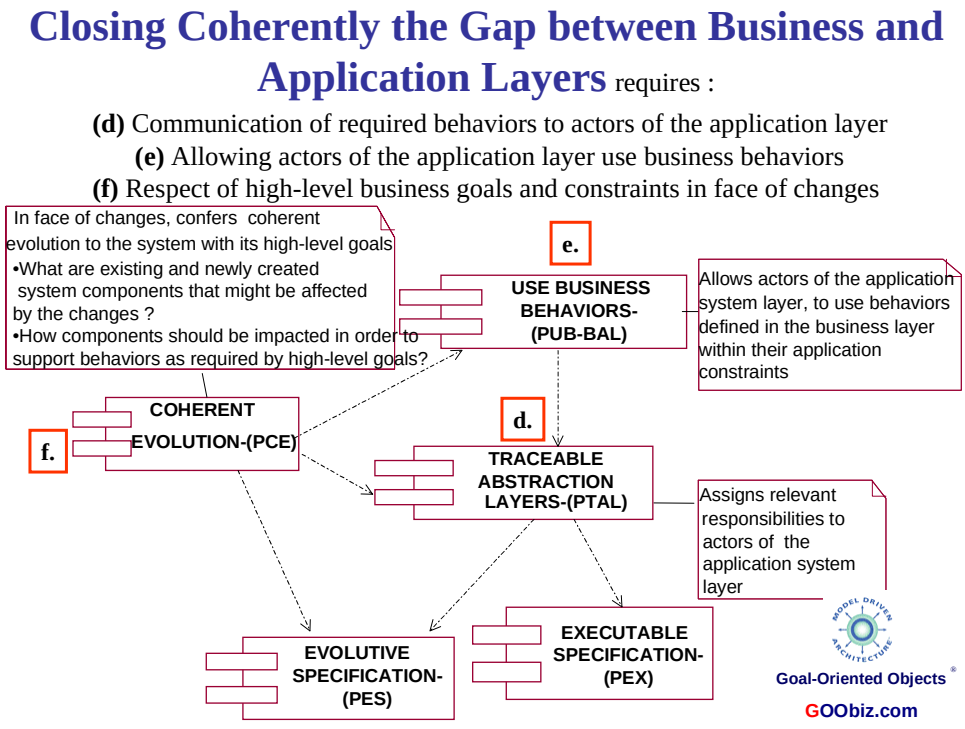


Figure 10: Dependency relationships between patterns for assuring traceability between business and application layers and to contribute the system evolve coherently in face of changes.

d) Pattern for Traceable Abstraction Layers (PTAL)

Intent: Making traceable specifications between business and application layers

Solution: Consider actors of the application layer. Let appear them as role-controllers for the business specification layer and indicate their responsibilities.

Explanation: This pattern aims to suppress the semantic gap between business and application layers in the realization of a requirement. To do this, it suggests to look for components of the system that necessitate at least one actor for their realization at the application layer in a given chain of refinement and describe responsibilities of corresponding role-controllers at the business layer. Such responsibilities can be precisely described using the name of the role-controller component, its functional boundary and its input and output behaviors (if necessary, using interaction and/or state diagrams for more precision) in the achievement of these responsibilities.

However, a component diagram is, in general, sufficient for illustrating a high-level description of the inputs for these role-controllers and for the description of targeted component interfaces that they must conform to.

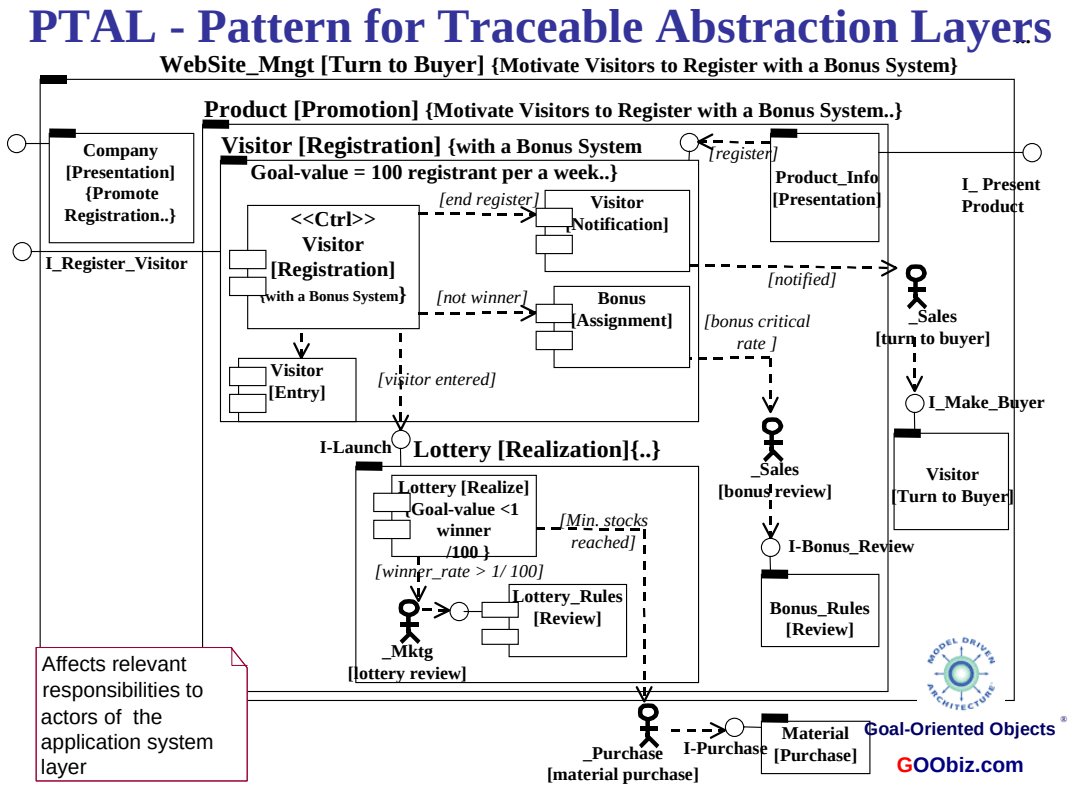


Figure 11: The component diagram illustrates a high-level description of responsibilities for the components and the role-controllers **(stereotyped by actor icons)** at the business level. Role-controllers are added as components to the previous specification of the business system. They represent meta-roles for actors of the application level.

e) Pattern for Using Business Behaviors (PUB-BAL)

Intent: Allow actors of an application system to use business behaviors (rules) as defined in the business layer with their application constraints

Solution:

- Consider instances of role-controllers that were stereotyped by <<actor>> icons in the previous pattern PTAL as actors for the application layer.
- Define if necessary, application components and classes for implementing operations that have to support usage constraints of actors.
- For allowing actors to use business and application behaviors, define use cases that capture actor-system interactions.

Explanations: Actors of the application layer realize responsibilities that were affected to them (via corresponding role-controllers in the business layer) directly by using behaviors defined for target components (if any) or indirectly, by including their usage constraints.

A direct usage allows actors to use business goals exactly as they are defined in the business layer. Actors need also invoke indirectly these behaviors by redefining some of them respecting their pre-conditions and post-conditions. Thus, business goals may also be specialized by other complementary sequences of actions that respond to the usage constraints of the application layer.

In all of these cases, we need to isolate high-level business behaviors (like actions related to the registration process of an internet visitor) from actor's application layer behaviors (like offering a visitor a look-up on promoted items during his/her registration process, ..) to allow the business layer evolve independently.

Finally, for supporting actor-system interactions, application use cases manage actions related to actor events (like selection menu management, fields checking, ..) and those related to communications with behaviors stored in the business GOO_Comps.

The diagram below shows different ways for using business and application behaviors (illustrated respectively by *Business Goal Case* and *Application Goal Case* stereotypes).

(e) PUB-BAL : « Let actors use assigned business behaviors with their application constraints »

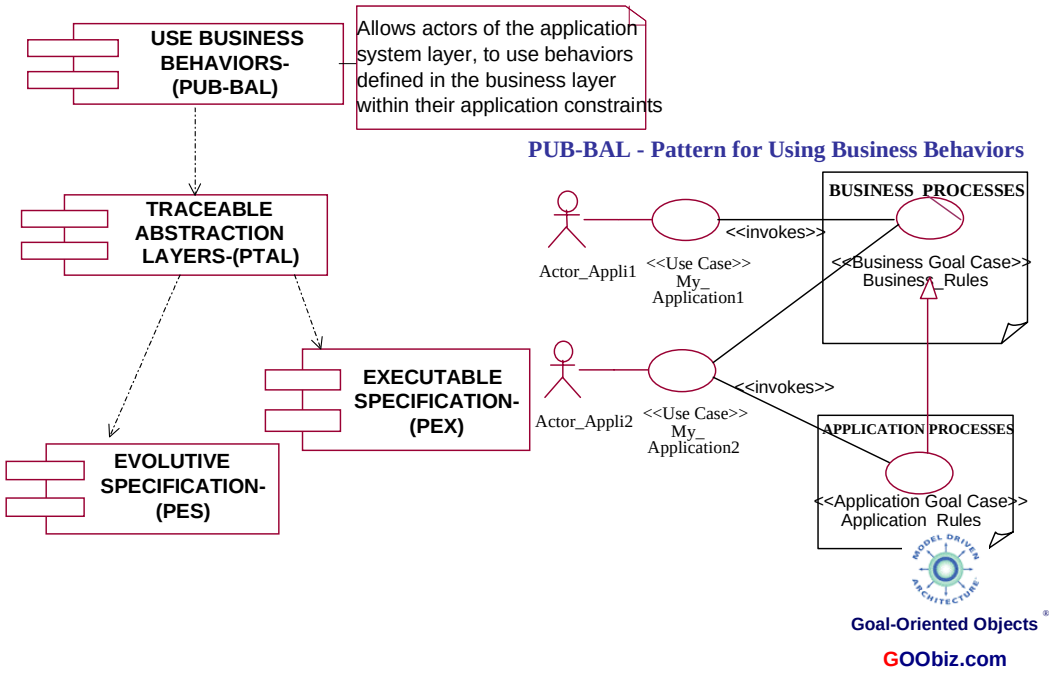


Figure 12: Behaviors stored in business goal-cases and application goal-cases are used via use cases. In the bottom part of the diagram, an application goal-case inherits behaviors from the business layer and eventually redefines some of them allowing actors use the system with their application constraints.

For example, use cases named *Visit Company Presentation* and *Register Visitor* use respectively business behaviors stored inside the GOO_Comps *Company [Presentation]* and *Visitor [Registration]*.

Responsibilities of the use case controllers may be specified based on the actor-system interactions. The description of the use case *UC-Register-Visitor* illustrates part of responsibities in the usage of the system behaviors.

A static aspect of the high-level view on the **usage of business components** can be illustrated by a component diagram.

The component diagram (figure 13) shows the static aspect of the usage of business behaviors by the application use cases.

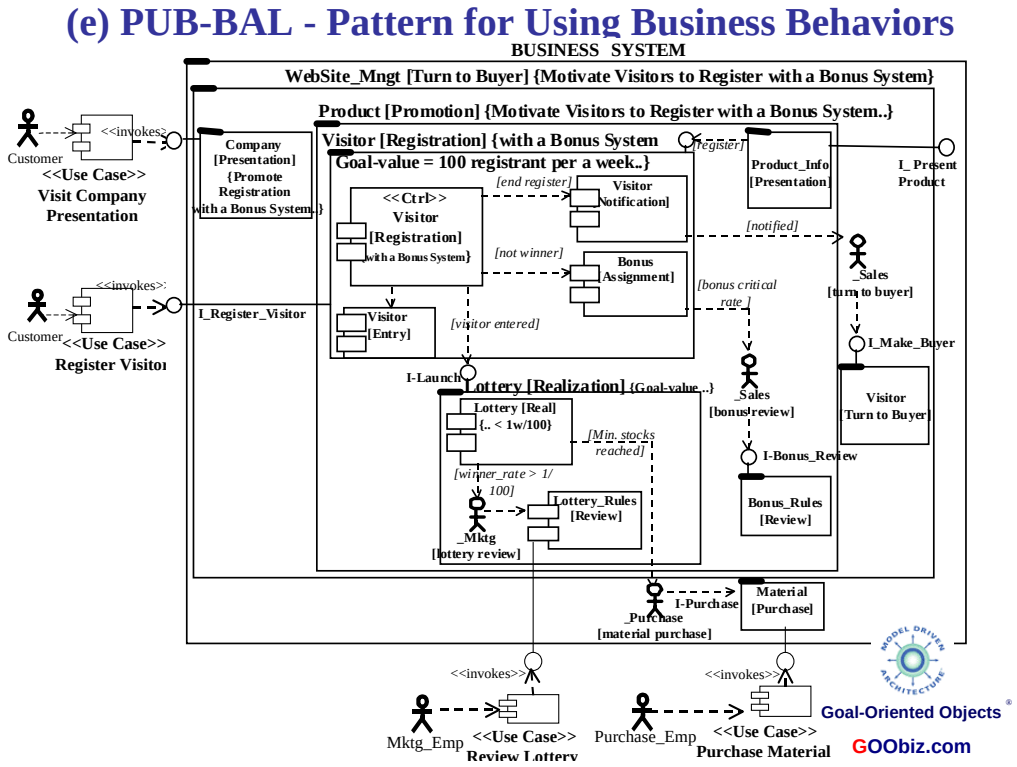


Figure 13: Components of Use Cases (on the border of the diagram) use behaviors from the business process components via their interfaces. Details of the content of the Business System is provided in the figure 11

Summary Description of the Use Case: The use case begins when an internet visitor asks the system for his/her registration. It is ended when the system confirms that a notification will be sent to the visitor. A notification contains information on the registration of the visitor and other relevant information about the bonus affectation and the lottery results.

ACTOR	SYSTEM
1-Visitor activates the UC for his/her registration.	2-System displays the menu of choices to the user
3-User makes his/her selection for the "Registration".	4-System returns the user the "Visitor Registration" form
5-User enters fields (name, surname, e-mail,) and submits the form	6-System checks mandatory fields and displays the questionnaire to the user.
7-User completes the questionnaire and submits, or leaves by canceling	8-System checks the result [**if abandoned : EXC1**] If OK, it stores fields in the database, affects bonus and realizes lottery. Then it finishes the transaction with a message of courtesy and informs the user by sending a notification.

EXC1: If user leaves during the filling of the questionnaire, then system terminates the transaction.

Based on application constraints described as part of actor-system interactions, new behaviors can be added to or substracted from related business GOO_Comps.

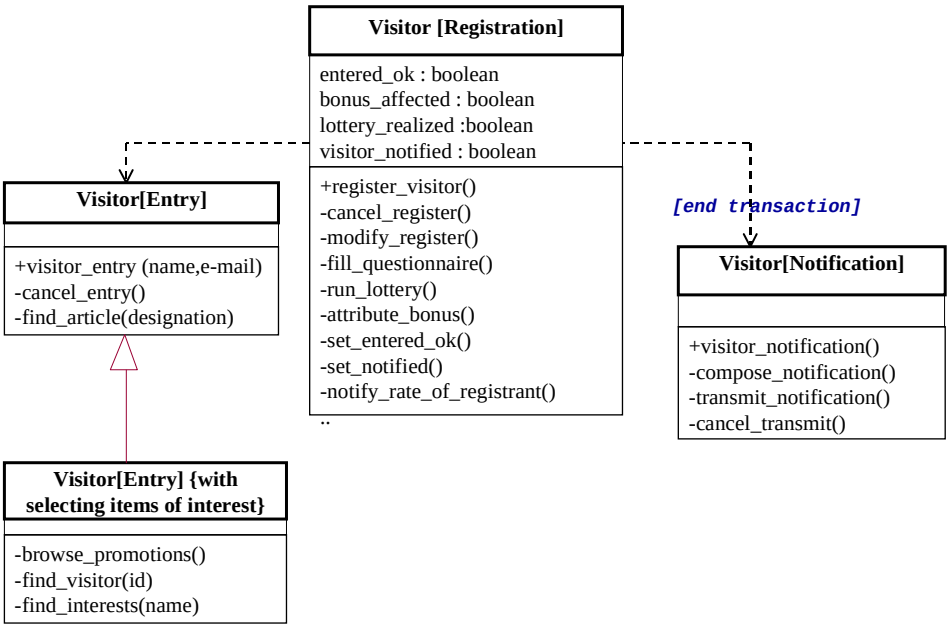


Figure 14: The class diagram shows implementation of part of responsibilities of the GOO_Comp *Visitor [Registration]* with new application constraints that concern *Visitor [Entry] {..},* designed as sub-class of *Visitor [Entry]*.

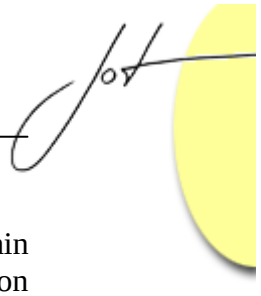
As a conclusion for the pattern PUB-BAL, actors of the application layer invoke business behaviors with their application constraints, according to business responsibilities that are communicated to them via the pattern PTAL. Separation of business goal-cases from the application ones allows business behaviors evolve independently from constraints of the application layer. Thanks to this distribution of responsibilities, use case descriptions become easy to validate and system components easy to maintain.

f) Pattern for Coherent Evolution (PCE)

Intent: Allow coherent evolution to the system with its existing goals when changes arise on its behaviors.

Explanation about this pattern is accessible on the Goal Driven Development Patterns at http://www.goobiz.com/GOObizWP/GOObizWP.htm#Patterns.

As a conclusion for patterns summarized above:



Goal-Oriented Objects constitute basic elements for getting flexibility within specifications. By applying patterns described in section 3, the resulting system is built on evolutive (flexible), executable and traceable specifications. Patterns presented in this section are based on these services for closing the gap between the business and application layers and for adapting system to changes with respect to its existing goals.

The next section presents a summary of the Goal-Driven Development Framework that illustrates **how to bridge these goal-based UML artifacts** to ensure good **levels of reactivity** for the resulting system.

5 GOAL-DRIVEN FRAMEWORK FOR ADAPTING INFORMATION SYSTEMS TO THEIR CHANGING BUSINESS ENVIRONMENT

Patterns described above do help analysts and designers in rendering specifications identifiable, evolutive, executable and traceable, based on requirements of non-technical business experts.

A methodological framework is then necessary to assist people in this process by suggesting necessary artifacts (textual specifications, UML diagrams, prototypes, ..) and patterns to use at each step of the process.

In this context, the **Goal-Driven Development Framework** offers a good level of traceability between related artifacts in the system lifecycle. It does necessitate two main parts:

1. **A business specification part that allows non-technical people to specify their business needs and business analysts formalize them using components of goal-oriented objects.** The figure below shows main steps and artifacts of this Goal-Driven Development Process :

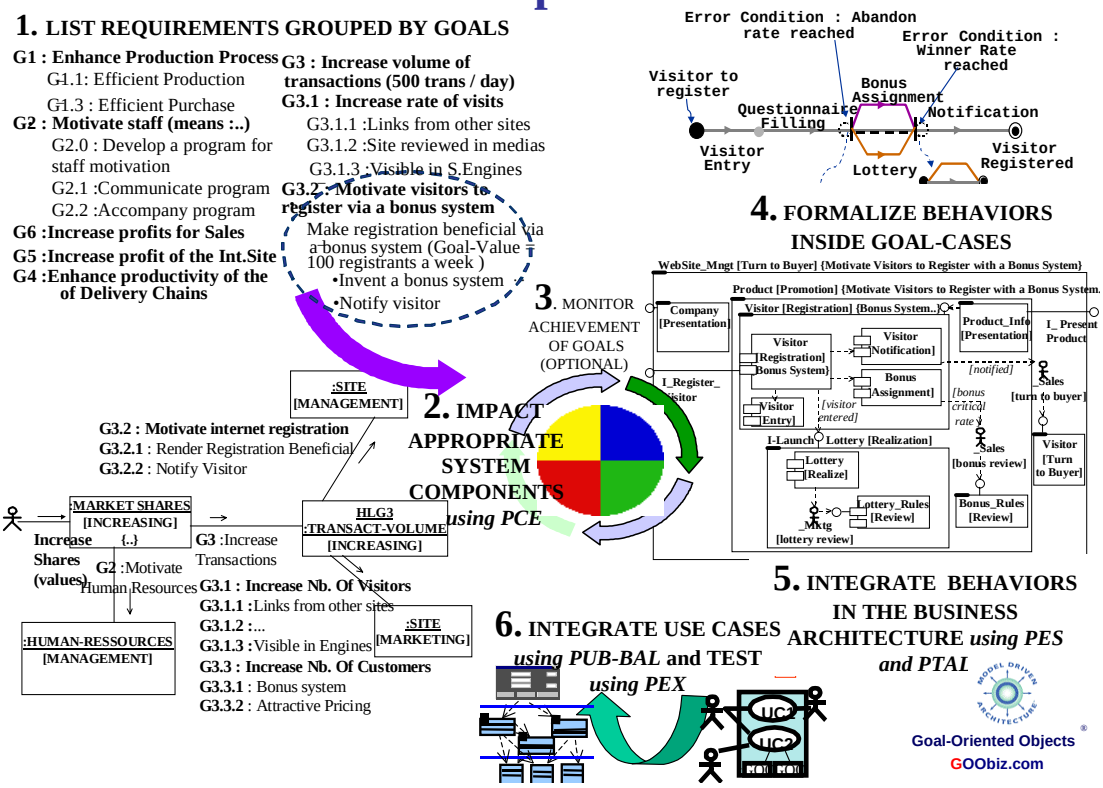


Figure 15: Steps and artifacts of the Goal-Driven Development Process.
Detailed content of the step 5 is provided in figure 11

A brief description of these steps and related artifacts of the process are presented with a case study in the Goal Driven Development Process at http://www.goobiz.com/Process/Overview_Process.htm .

2. **An application software specification part that permits use case specifiers to describe invocation of business behaviors by use cases at the application system layer.** The figure below shows main artifacts of this Goal-Driven Software Development Process. It does allow to make a zoom on the step 6 of the previous figure.

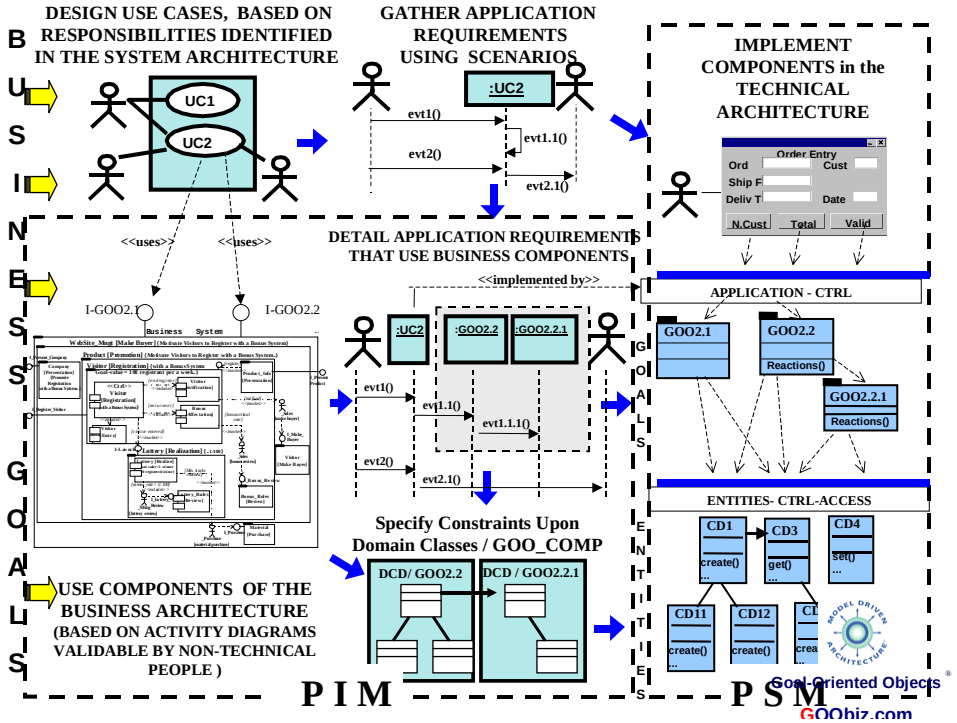


Figure 16: The framework presents artifacts and traceability relationships for the Goal-Driven Software Development Process guided by business goals and organized by application constraints.

A brief description of these steps and related artifacts of the process are presented in the Goal Driven Software Development Process at http://www.goobiz.com/Process/Overview_Process.htm#BM2

As a result, the entire development framework allows continuity of specifications from requirements capture till implementation of related business and application behaviors on the technical platform (PIM). Thus, it does assist portability of these PIM level specifications into appropriate components of the technological target platform (PSM) *such as Servlets/JSPs , Session and Entity Beans in J2EE* ™ using the patterns PEX and PES.

6 CONCLUSION

Patterns and the development framework briefly presented above aim to increase the reactivity of systems developed with UML in the sprit of the OMG's Model Driven Architecture (MDA).

Goal-Driven Development Patterns allow analysts and designers to render their system **specifications easy to change**, to **retain the validity of analysis specifications** at lower development levels (design and implementation). Using these patterns, **business**

components evolve coherently with business strategies. Specifications that are rendered traceable between the business and the application layers allow actors of the application systems able to use business behaviors (rules) as they are defined at the business layer, with a total transparency when changes occur upon these behaviors.

The Goal-Driven Development Framework acts as a catalysor in such a development process by ensuring traceability between related UML artifacts, contributing so directly to the reactivity of the resulting system.

REFERENCES

[OMG] http://www.omg.org/mda/executive_overview.htm

[MDA] http://www.omg.org/cgi-bin/doc?mda-guide

[Odel03] James Odell et al: "The Role of Roles", in *Journal of Object Technology*, vol. 2, no.1, Jan-February 2003, pp. 39-51. http://www.jot.fm/issues/issue_2003_01/column5

[Berk03] Birol Berkem: "Patterns for Model Driven Development with UML", at the OMG Meetings–BEIDTF (Paris) http://www.omg.org/cgi-bin/doc?bei/2003-07-01, June 2003

[Berk99] Birol Berkem: "Traceability Management with UML" - Journal Of Object Oriented Programming (JOOP), September 1999

About the author

Birol Berkem is a consultant and trainer in the areas of business and application system analysis and design with the object technology. He is the author of the Goal-Driven Development Patterns and Frameworks for the traceability of specifications with MDA/UML. His e-mail address is birol.berkem@goobiz.com