# Using Patterns to Integrate UML Views

**Alexander Egyed**
**Center for Software Engineering**
**University of Southern California**
**Los Angeles, CA 90089-0781, USA**
**+1 (213) 740 6504**
**aegyed@sunset.usc.edu**

*Abstract*

*Patterns play a major role during system composition (synthesis) in fostering the reuse of repeatable design and architecture configurations. This paper investigates how knowledge about patterns may also be used for system analysis to verify the conceptual integrity of the system model.*

*To support an automated analysis process, this work introduces a view integration framework. Since each view (e.g. diagram) adds an additional perspective of the software system to the model, information from one view may be used to validate the integrity of other views. This form of integration requires a deeper understanding as to what the views mean and what information they can share (or constrain). Knowledge about patterns, both in structure and behavior, are therefore a valuable source for view integration automation.*

## 1. Introduction

To support the development of software products we frequently make use of general-purpose software development models (and tools) such as the Unified Modeling Language (UML). However, software development in general and software design in particular (which is the main focus of our work) requires more than what most general-purpose models can provide. Architecting is about:

1) modeling the real problem adequately
2) solving the model problem and
3) interpreting the model solution in the real world

In doing so, a major emphasis is placed on mismatch identification and reconciliation within and among architectural views (such as diagrams). We often find that this latter aspect, the analysis and interpretation of (architectural) descriptions, is under-emphasized in most general-purpose languages. We architect not only because we want to *build* (compose) but also because we want to *understand*. Thus, architecting has a lot to do with analyzing and verifying the conceptual integrity, consistency, and completeness of the product model.

The emergence of the Unified Modeling Language (UML), which has become a de-facto standard for OO software development, is no exception to that. This work describes causes of architectural mismatches in UML views and shows how patterns and integration techniques can be applied to identify and resolve them in a more automated fashion. In order to do so, this work describes a view integration framework with its major activities – *Mapping*, *Transformation*, and *Differentiation*.

Instead of focusing on the wide array of integration techniques (which support the activities above), this paper will investigate the role of patterns. Thus, we will investigate how knowledge of patterns help in ensuring the conceptual integrity of software system models. With that we make use of patterns in a manner which is rarely ever done: instead of using patterns as building stones for system composition, we use them for system analysis.

### Views and Models

In software development, we make use of models and views to cope with the complexity of software systems. Here, a model refers to a collections of views or views can be seen as perspectives (or viewpoints) of that model. The IEEE Draft Standard 1471 [AT&T 1993] refers to a view as something which "addresses one or more concerns of the system stakeholder." With stakeholder we mean an individual or a group that shares concerns or interests in that system (e.g. developers, users, customers, etc.). Applied to our context, a *view* is a piece of the *model* that is still small enough for us to comprehend but that also contains all relevant information about a particular concern. In UML,

views are mostly graphical in nature and are often realized through diagrams. Views (e.g. class or sequence diagrams) serve the following purposes:

- Abstract and simplify the model
- Enable different stakeholders to work and to coordinate
- Compensate for different interpretations (different audiences/stakeholders)
- Extract relevant information about a particular concern.

What types of views should be used and when they should be used is, therefore, strongly dependent on the person who is using them and a corresponding tasks that are needed to be accomplished. Views, however, are not the silver bullet of software development because they embody a fundamental problem; they exhibit a fair amount of redundancy of modeling elements within and between them.

To give a simple example, consider a software development case where we have a design (e.g. in form of an UML class diagram) and the product implementation (e.g. in C++ code). The class diagram and the code represent different views expressing the same or similar information in different ways. Although, code can be generated automatically from the design, this approach is limited and some information must still be added multiple times. Even worse, these now redundant pieces of information must be kept consistent – the latter is a mostly manual activity. Thus, whenever the design changes the code becomes inconsistent (or vice versa) and we need to apply some view reconciliation activities to find the resulting inconsistencies and to again ensure the conceptual integrity of the model.

## View Mismatches and Redundancy

Since views are our only effective means of dealing with complexity, we cannot hope to replace them with something less redundant. We need views to break down the amount of information software developer have to deal with at any given time. "It is not the number of details, as such, that contributes to complexity, but the number of details of which we have to be aware at the same time." [Siegfried 1996]

Redundancy is, therefore, a necessary evil. This implies that we need some way of automating the activity of identifying and resolving mismatches between views. Thus, what we need is some form of framework for integrating and analyzing views. Interestingly, a possible way of approaching the view mismatch problem is based on its very problem – that of redundancy. We exploit the fact that redundancy between a set of views implies that one view contains information about another view which can be used to constrain that view. Thus, we use redundant information to verify consistency and completeness between view.

For instance, if we architect a system using some form of architectural pattern (e.g. layering style) then the design must reflect the constraints opposed by that architecture. This means that if the architecture defines three layers then that architecture implicitly defines the constraint that the first layer is not allowed to talk directly to the third layer without using the second one in the process. If that system is designed in UML later on (e.g. using class and sequence diagrams) then calling dependencies within the design element need to be consistent with above architectural constraint. We will show an example of that later.

## UML View Representation versus Integration

Enabling view integration to identify and reconcile views requires two levels of integration – notational integration and semantical integration. With notational integration we mean that the model needs to be capable of fully *representing* views. Semantic integration further refines that by defining *what* information can be exchanged and *how* it can be exchanged. Only after the *what* and *how* have been established, inconsistencies can be identified.

The Unified Modeling Language (UML), like other general-purpose software system development models, satisfies above semantic integration only poorly. UML provides a model for representing various diagrammatic views that deal with classes, objects, activities, states, packages, and others (see Figure 1). However, UML falls short in integrating them. Relationships and dependencies between UML views are rarely captured. If the latter is not done, the model is nothing more than a collection of either loosely coupled or completely unrelated views. Although, UML and its meta-model define the notational and semantical aspects of individual views in detail, inter-view relationships are only captured insufficiently (only some weak forms of dependencies between views exist, such as between classes and objects).

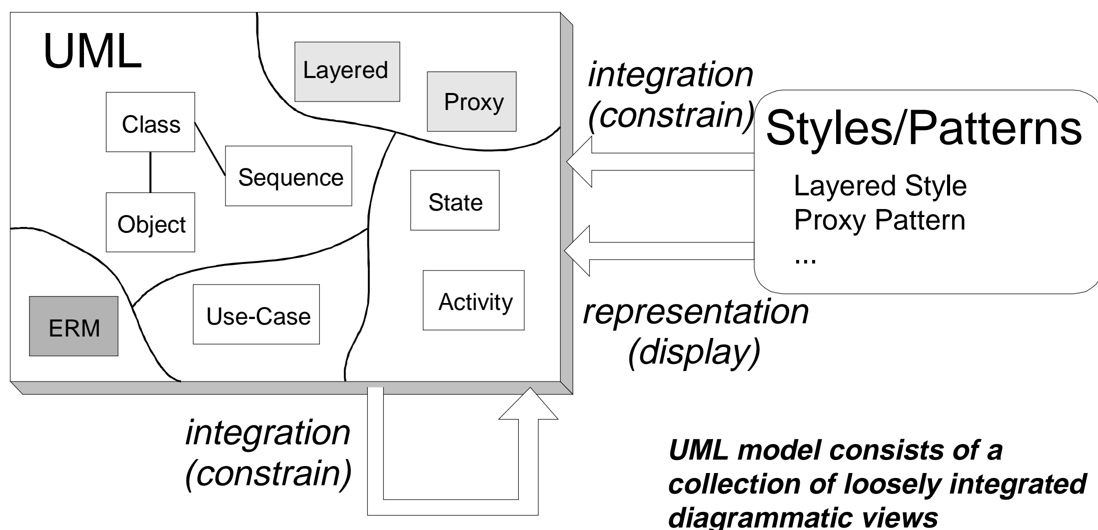


**Figure 1: Representation versus Integration in UML**

Figure 1 also shows another dimension of that problem – that of extending UML to represent new and foreign concepts (e.g. styles and patterns). For instance, how can more advanced patterns such as the layered architecture pattern[1] or the proxy design pattern be used in UML? Again we need to distinguish between merely representing patterns in UML and fully integrating them.

## 2. The View Integration Framework

The major obstacle to view integration is the lack of a well-defined (engineered) model base. Views often use very different ways of expressing information and that makes it rather hard to identify where and how they overlap. Thus, the task of composing and comparing views is often a manual and error-prone one. The goal of the integration framework is to compensate for the lack of automated assistance in identifying and resolving architectural mismatches.

In doing so, our framework needs to deal with the *what* and *how* of information exchange which we briefly discussed above. There we wrote about the need of deciding what information can be exchanged and how it can be exchanged. In our view integration framework, we refer to these two activities as *Mapping* and *Transformation*. We also said that only after those activities were defined, could we try do identify inconsistencies. That latter activity we call *Differentiation* (see Figure 2).

Figure 2 depicts our view integration framework in a high-level fashion. There, a system model is used to represent the knowledge base of the software system. Software developers use views to add new data to the system model and to review existing data (view synthesis). In case of UML, the system model may be seen as being the UML model and view synthesis is done via a UML design tools (e.g. Rational Rose).

Interacting with both, the system model and view synthesis, is view analysis. As soon as new information is added, it can be validated against the system model to ensure its conceptual integrity. Figure 2 shows how view analysis can be further subdivided into its three major activities we discussed above:

[1] Note that we use the term patterns and styles interchangeably as done by [Buschman et al 1996]

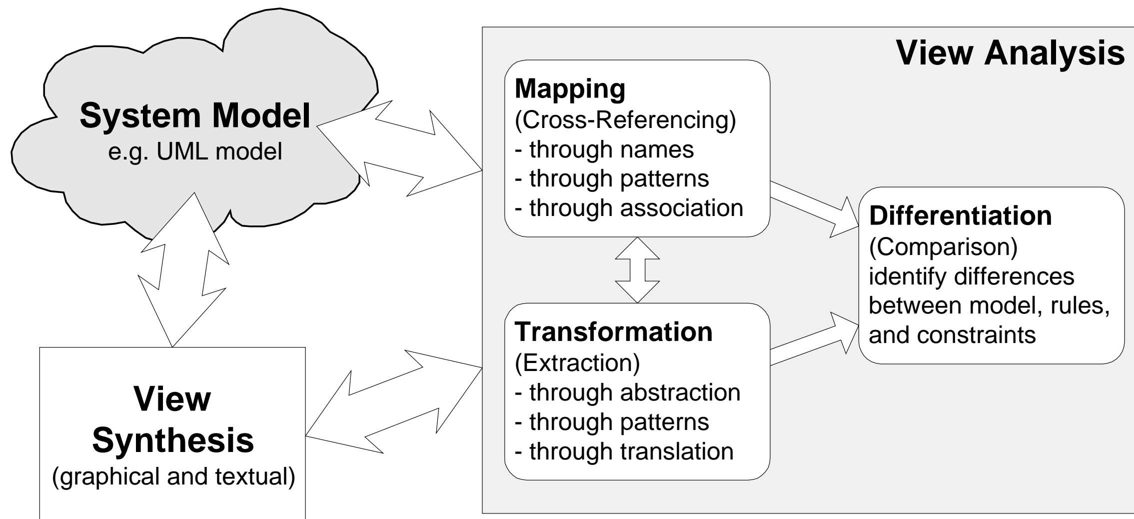


**Figure 2: View Integration Activities**

- **Mapping:** Identifies related pieces of information through the use of naming dictionaries, traces and trace simulation (e.g. the use of same physical classes and methods), and certain forms of associations/patterns (e.g. common interfaces).
- **Transformation:** Manipulates model elements in views so that they (or pieces of them) can be shared with other views (or represented in the system model itself). For instance, we may use abstraction techniques to generalize a detailed diagram, we may use view translation to exchange information between different types of views, or we may rearrange model elements (or pieces) in different manners to create new perspectives (e.g. merging or splitting).
- **Differentiation:** Traverses the system model to identify (potential) mismatches within the system model. (Potential) mismatches are described in form of rules and constraints. Furthermore, mismatch resolution rules can be associated with mismatch identification rules to propose options on how to resolve them. Differentiation is strongly dependent on *Transformation* and *Mapping*.

It must be noted, however, that these activities are not orthogonal to each other. Obviously, we can only make useful transformations if we know the proper mapping of model elements. This dependency is also true in reverse. Information derived through view transformation can clarify many ambiguities in the mapping. Thus, one view may be used to clarify ambiguities in other views.

Furthermore, as seen in Figure 2, view integration is not limited to patterns only, however, patterns constitute a very important cornerstone to view integration. We will discuss this pattern-oriented view integration aspect in the following sections. Other non-pattern related aspects of view integration are described in [Egyed 1999a] and [Egyed 1999b]. The framework presented above is generally applicable even outside UML.

## 3. Product Ordering System Example

To guide through this paper we will make use of a simple *Product Ordering System* which is divided into two primary subsystems – the *Order Capture Subsystem* and the *Order Processing Subsystem*. The first subsystem captures order and payment information from customers via sales representatives. The latter subsystem captures the processing of the product order queue by warehouse employees. The *Product Ordering System* incorporates two COTS (Commercial off the Shelf) products: an *Inventory System* that handles the product inventory and an *Order Repository* as the database (the latter is used by both the *Product Order System* and the *Inventory System*).

Table 1 shows the architecture of our system which is designed using a *Layered Pattern* (or *Layered Style*). This architecture pattern will be complemented by design patterns and other design features later on.

## 4. How do Patterns Help?

As the paper unfolds we will reveal more detail about the *Product Ordering System*. However, due to limitation in space we cannot present the entire system here nor can we present all

**Table 1: Layered Architecture Pattern to describe the Product Ordering System**

| **Product Ordering System** |
| --- |
| - User Interface (Order Capture UI, Order Processing UI, Inventory UI) |
| - Order Framework (Customer, Payment, Order, OrderLine, Reorder) |
| - Inventory System |
| - Network (LAN) |
| - Order Repository |

integration techniques. As mentioned before, we will instead focus on the role patterns can play during view integration. Corresponding to the three integration activities outlined in Figure 2, patterns support the following activities (the next section will show examples):

### Mapping

Patterns support the mapping (cross-referencing) between views of different levels of abstractions. For instance, a high-level diagram indicates the use of a known pattern that is later on realized in a lower-level diagram. Thus, the knowledge that this type of pattern exist in a lower-level diagram as well as the knowledge how this pattern roughly looks like (as defined in books like [Gamma et al. 1994] and [Buschman et al 1996] can help in automatically identifying it in the lower level diagram.

Patterns also support the mapping of views of different types. For instance, pattern descriptions often specify both the structure of patterns and their dynamic behavior. Thus, we may use that knowledge to cross-reference structural and dynamical information in our model.

## Transformation

The way patterns are used for transformation is analogous to their usage for mapping. In transformation we can use them for both abstraction and translation. With abstraction we mean the process of simplifying views. For instance, if we wish to know whether a high-level view and a low-level view are consistent then we need to either refine the high-level view or we need to abstract the low-level view so that a direct comparison is possible. The former cannot be automated, the latter can. In order to abstract views, related pieces need to be identified which can then be replaced by something simpler. Patterns are an excellent source for that because they provide the knowledge of what pieces belong together. We can use that pattern knowledge to abstract low-level patterns into higher-level patterns (or even single components).

Patterns may also be used for translation between static and dynamic structures as we discussed in *Mapping*. Since patterns are often described in both fashions, we can infer behavior through structure (or vice versa). Thus, with patterns we can also translate views.

## Differentiation

As outlined in Figure 2, *Differentiation* is strongly dependent on *Mapping* and *Transformation*. This means, that in order to find inconsistencies between views we need to identify the relationships of their modeling elements and we need to find a way of transforming information from one to the other. Without the former step we would not know *what* information to compare and without the latter step we would not know *how* to compare it. Once that is done, we can use two primary techniques to compare views.

- (Graph) Comparison: Views are compared by means of comparing transformed views with original views. This technique implies that one view can be transformed into another view in a sufficiently complete manner.
- Constraint and Rule Checking: Quite frequently, we find that views cannot comprehensively be transformed into another view but only bits and pieces of it. In this case, we can use constraints and rules to describe and analyze these pieces.

Design patterns are not directly useful for *Differentiation*, however, the information we gathered about views through *Mapping* and *Transformation* are. We already briefly discussed how patterns help in mapping and transforming views. In these cases, comparing views is straightforward since *Mapping* and *Transformation* enables a direct (graph) comparison. However, patterns are also useful in constraint and rule checking. For example, the layered pattern we introduced in Table 1 defined constraints of the nature: *User Interface* is only allowed to talk to *Order Framework*, the

*Order Framework* in turn may only talk to the *Inventory System* and so on. This architectural pattern constraint affects both the structure of the design as well as its behavior.

## 5. Use of Patterns in the Product Order System

This section will complement above discussion by showing examples of how patterns can be applied to integration activities in the context of our *Product Order System*.

### Differentiation

Figure 3 shows a high-level design view of our system using UML packages. The figure shows the interaction of the major *Order System* components (or subsystems). Knowledge about the layered architecture pattern can now assist us in automatically identifying mismatches between the architecture in Table 1 and the design in Figure 3. Table 2 summarizes the constraints opposed by both views.

The *Architectural View Constraints* were derived from Table 1. They define the calling dependencies between the layers of our system (e.g. User Interface depends-on Order Framework). Figure 3 is the foundation for the *Design View Constraints*. This figure shows a UML package diagram containing a set of packages and dependency calls between them (the semantics of packages and dependencies are defined in [Booch-Jacobson-Rumbaugh 1997]).

The creation of these constraints is the responsibility of *Transformation* and can be done

**Table 2: Constraints opposed by Architectural and Design Views**

```
Architectural View Constraints
    Architecture[User Interface depends-on Order Framework];
    Architecture[Order Framework depends-on Inventory System];
    Architecture[Inventory System depends-on Network];
    Architecture[Network depends-on Order Repository];
Design View Constraints
    Design[Order Capture UI depends-on Order Framework];
    Design[Order Processing UI depends-on Order Framework];
    Design[Inventory UI depends-on Inventory System];
    Design[Order Framework depends-on Inventory System];
    Design[Order Framework depends-on Network];
    Design[Inventory System depends-on Network];
    Design[Network depends-on Order Repository];
Mapping
    Design[Order Capture UI]        maps-to Architecture[User Interface];
    Design[Order Processing UI] maps-to Architecture[User Interface];
    Design[Inventory UI]            maps-to Architecture[User Interface];
    Design[Order Framework]         maps-to Architecture[Order Framework];
    Design[Inventory System]        maps-to Architecture[Inventory System];
    Design[Network]                 maps-to Architecture[Network];
    Design[Order Repository]        maps-to Architecture[Order Repository];
Completeness Rules
For all Architectural View Constraints Exist Design View Constraint;
Example:
Architecture[User Interface depends-on Order Framework] =>
       Exist: Design[(Order Capture UI depends-on Order Framework)] or
              Design[(Order Processing UI depends-on Order Framework)] or
              Design[(Inventory UI depends-on Order Framework)];
Consistency Rules
For all Design View Constraints Exist Architectural View Constraint;
Example:
Design[Order Processing UI depends-on Order Framework)] =>
       Exist: Architecture[User Interface depends-on Order Framework];
```

**Table 3: Layered Architecture Pattern to describe the Product Ordering System**

| **Product Ordering System** | |
|---|---|
| User Interface (Order Capture UI, Order Processing UI, Inventory UI) | |
| Order Framework | Inventory System |
| Network (LAN) | |
| Order Repository | |

automatically in this case. For instance, using our knowledge of the layered patterns in Table 1 we can automatically derive the calling dependencies between layers. The advantage is that the semantics of patterns need only be defined once and can be reused later on. The semantics and notations of views can be seen as patterns as well. Thus, the package diagram in Figure 3 has a predefined set of constraints associated with it. Once defined, design constraints can be derived for different instances of package diagrams.

The *Mapping* section in Table 2 defines the relationship between components of the architecture view (Table 1) and the design view (Figure 3). In this example, the use of patterns for mapping is less obvious. We will discuss their use for *Mapping* later on.

The final two sections in Table 2 are part of the *Differentiation* activity. There, two types of rules are defined, one for consistency and the other for completeness. If the architecture defines some components or connectors that are not reflected in the design than this may indicate a potential

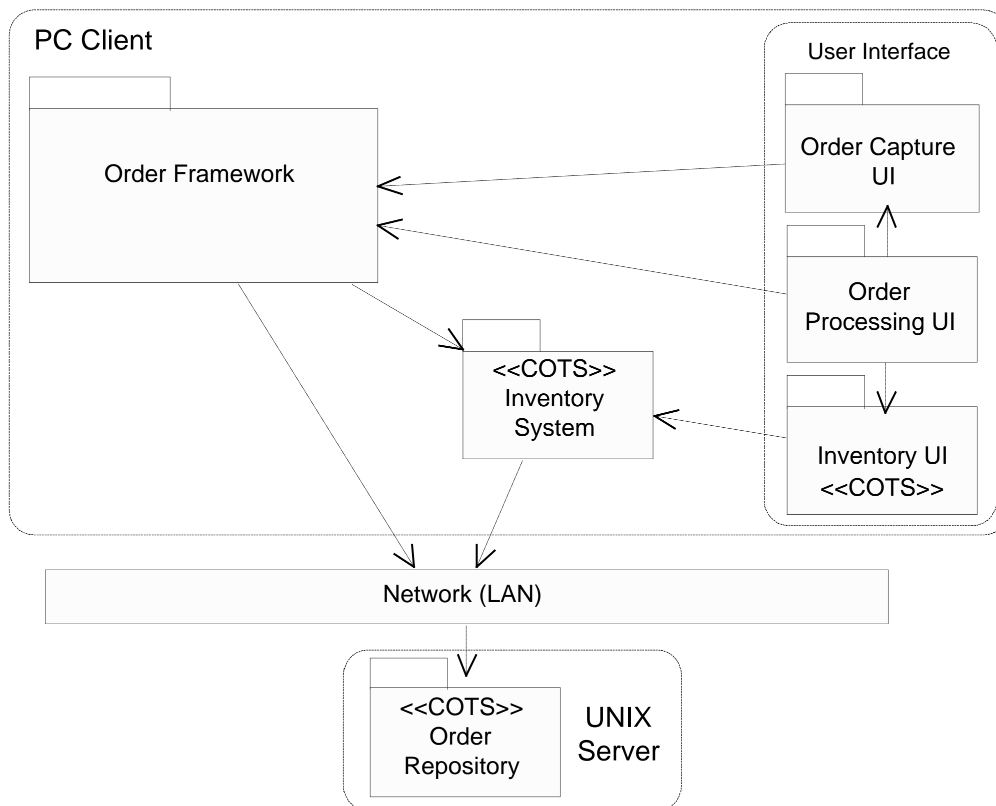


**Figure 3: High-Level Design in UML (Package Diagram and Dependencies)**

incompleteness. On the other hand, if the design contradicts the architecture then this may indicate a potential inconsistency. Again, rules need to be defined only once for each set of views we compare; those rules can then be reused. Identifying mismatches between the architecture and design realization is now a matter of evaluating the rules and constraints. Doing so reveals no completeness mismatches, however, a number of consistency mismatches:

1) Dependence of *Inventory UI* component on *Inventory System* is a violation of the layering architecture where the user interface is not allowed to talk to the I*nventory System* without going through the *Order Framework*.
2) Similarly, the *Order System* needs to use the *Inventory System* to access the *Network*.

Identifying those mismatches does not give any feedback as to their origin. For instance, is the architecture or is the design wrong? Table 3 shows a possible way of resolving all above mismatches without introducing new ones by raising the *Inventory System* to the same level as the *Order Framework*. We do not belief that the actual mismatch resolution should be done fully automatic. This issues is related to a previous endeavor of self-correcting source codes for compilers which ultimately failed because of the social and technical complexities involved. However, we belief that presenting the architect not only with (potential) mismatches but also with ways on how to resolve them is highly beneficial in both dealing with mismatches and in understanding them. Furthermore, it might be very beneficial to also devise techniques that deal with the issue of which options are better. We discuss this issue in more detail in [Egyed 1999b].

## Mapping

Figure 4 shows another view of our system, this time a refinement of Figure 3. Again, this view can be validated against both the revised architecture and the higher-level design, however, no mismatches will be found. This view makes use of additional design patterns such as the *Template*[2] pattern (indicated through the <<Template>> stereotype) and the *Proxy*[3] Pattern (indicated through the <<Proxy>> stereotype). We will use those to further explore the value of patterns in view integration.

[2] Hides the source template and classifier for commonly used templates

[3] "Provides a surrogate […] for another object to control access to it" [Gamma et al 1994]

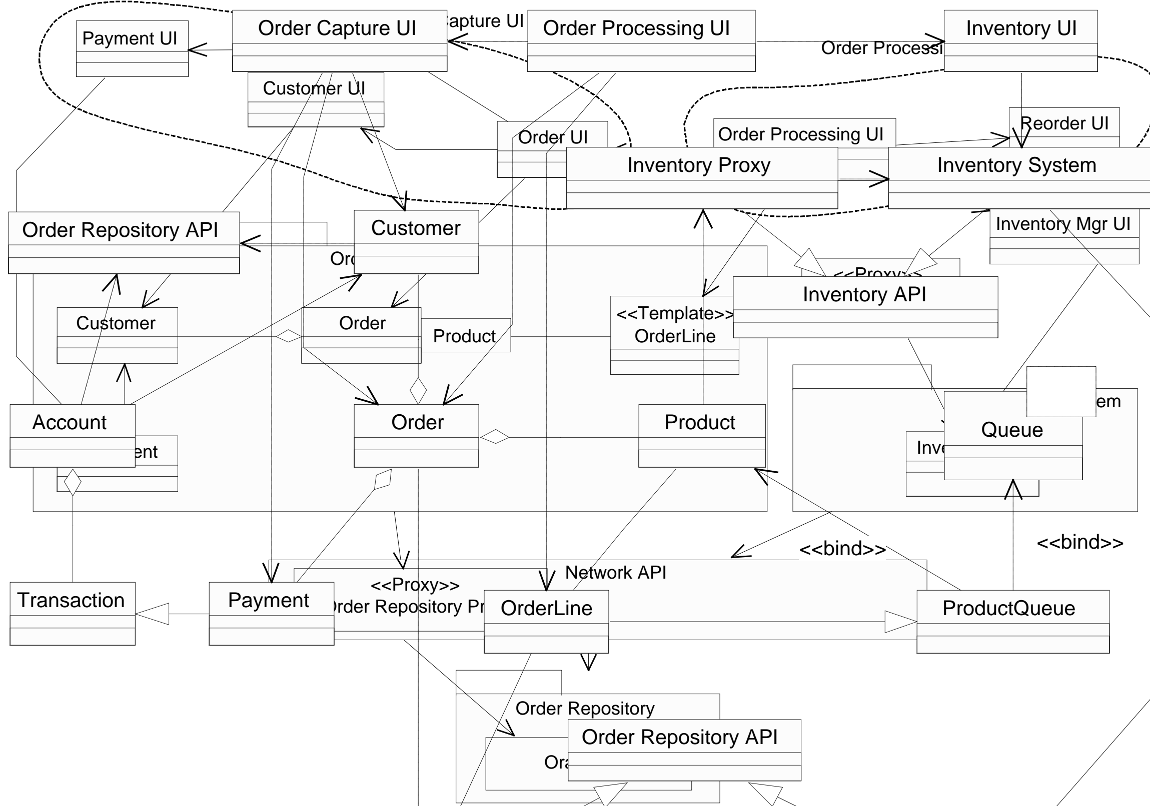


Figure 4: Refinement of High-Level Design View using UML classes and packages

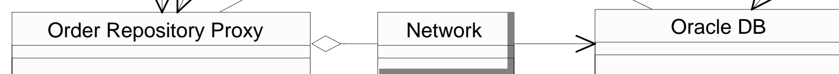


**Figure 5: Low-Level Design View using UML Class Diagrams**

Figure 5 depicts the corresponding lower-level realization, which not only resolves the patterns used in Figure 4 but also refines some of the other modeling elements. The top three classes correspond to the *User Interface* layer, the *Inventory System* can be accessed via a *Inventory Proxy*, and the *Repository* can similarly be accessed via the *Repository Proxy*. In order to find out whether this view is consistent with the previous views we can apply a number of integration techniques.

First, we need to find out which modeling elements correspond to each other (Mapping). There are a number of techniques which can be used, such as similarity in names, etc. However, the extensive use of patterns in this example enables us to exploit knowledge about them for *Mapping*. Through the high-level design in Figure 4 we know several facts:

- Template Pattern is used to describe *OrderLine*
- Proxy Pattern is used to bridge *OrderLine* (Product) and Inventory System
- Proxy Pattern is used to bridge *Order Framework* subsystem with the *Oracle DB*

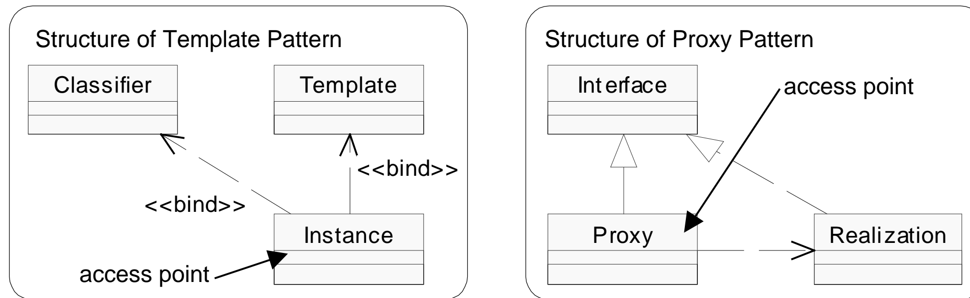


**Figure 6: Structural Pattern Knowledge (adapted from [Buschman et al. 1996]**

- Interface Features (e.g. Customer class is interfacing with Order, Payment, and Customer UI)

Although, technically, the last item in the list is not an explicitly defined pattern, it nevertheless constitute knowledge about the configuration of classes – thus, interface features may be seen as patterns although they are domain patterns for the most part only. The pattern knowledge about the domain as well as the knowledge about predefined patterns (see Figure 6) enables us now to reason about the relationship of modeling information. The task of mapping Figure 4 and Figure 5 is reduced to identifying the location of above patterns and (interface) features using predefined structural pattern knowledge as described in Figure 6.

Using Figure 6 for guidance, we can easily identify the correspondence of the *Template* pattern (*Product*, *Queue*, and *Product Queue* correspond to *OrderLine*). Although, it is also easy to find the *Proxy* patterns, it is less clear how they can be distinguished. Note, that the goal is to enable the computer to automatically identify them. To do the latter, we can make use of the Interface features (patterns) we discussed above. There, the idea is that at least some mapping exists or can be established through similarity in names. Using this additional information it is possible to automatically distinguish the *Inventory* pattern from the *Oracle* pattern.

Unfortunately, *Mapping* through patterns is still a bit more difficult than above examples may indicate. We made the simplified assumption that the structure and behavior of patterns are static. However, often patterns are not that precisely defined and we need a more general description of them. Figure 5 shows such a case with the *Repository Proxy* pattern which does not look exactly like its definition. However, since *Network* is part of the proxy class, it can be merged into it and the proxy class inherits all its dependencies (Rose/Architect in the next section will present a model for doing that).

Another problem with this mapping technique is that patterns, when recognized, may sometimes only roughly indicate their location. For instance, the *OrderLine Queue* pattern can be found in the lower-level diagram as corresponding to *Queue*, *Product* and *ProductQueue*. Although, this is correct, it misses the component *OrderLine* which is also present in the lower-level diagram.

## Transformation

### Patterns and Abstraction

*Mapping* between the high-level view in Figure 4 and the lower-level view in Figure 5 alone is not sufficient in identifying mismatches. For example, we see that Payment is part of Customer in Figure 4, however, that relationship is more complex in Figure 5. In order to verify that both figures describe relationships the same way, we can use the concept of Rose/Architect.

Rose/Architect [Egyed-Kruchten 1999] identifies patterns of groups of three classes and replaces them with simpler patterns using transitive relationships. In class diagrams, a transitive relationship describes the relationship between classes that are not directly connected. A relationship, however, may exist through other classes (e.g. helper classes) which form a bridge between them (e.g. in case of above example *Payment* and *Customer* are not directly connected but a relationship is still given through the helper classes *Transaction* and *Account*). Thus, if some formula is discovered which could derive transitive relationships with sufficient accuracy, then some automatic support in simplifying and abstracting class diagrams could be provided in tool form. This would allow architects to abstract important classes from existing, more detailed models by eliminating the 'helper classes' and thus would further enable them to portrait and analyze the interrelationships between classes even if the classes were scattered in different locations throughout the model (e.g. in different diagrams, or in different packages and name spaces).

RA provides this mechanism and [Egyed-Kruchten 1999] describe this technique in detail. Currently, the RA model consists of roughly 80 rules of abstraction. Rule 4, for instance, describes the case of a class which is generalized by the second class (opposite of inheritance) and that parent class is an aggregate (part) of the third class (see also Figure 7). This three-class pattern can now be simplified by deleting the middle class and creating a transitive relationship (an aggregation in this case) which goes from the first class to the third one. The underlying RA model describes these rules and how they must be applied to yield an effective result.

Figure 7 shows the RA refinement steps for the case of the *Payment* to *Customer* relationship of our lower-level design view (Figure 5). After applying two rules (rules 4 and 17 respectively) we get a simplified pattern of two classes and a dependency relationship between them. If this is also done for the other classes as well as for the patterns[4] we discussed in the mapping section we can automatically generate a higher-level class diagram (see Figure 8). This generated abstracted view can now be compared directly with the original higher-level class diagram we depicted in Figure 4. Thus, through the use of patterns we are now able to convert one view so that it represents information in a very similar manner as the other view. Comparing views can now be done by simply using some form of graph comparison algorithm (see also *Differentiation* above). Figure 8 also shows possible inconsistencies. For instance, the aggregation from *Payment* to *Order* is missing, the *Inventory System* call to *Oracle DB* does not use the *Network* component, and some links are different (e.g. association link instead of dependency link). After *Transformation* (abstraction) these mismatches can easily be recognized.

It must be noted that the abstraction was not done completely automatically. Although, patterns helped in identifying the *Mapping* and *Transformation* between some modeling information, they were not able to fully automate that process. Therefore, some manual assistance was needed to derive Figure 8. Further, the RA tool is currently only able to cope with simple 3 class patterns as described in Figure 7 but not with more advanced design patterns described in Figure 6. Thus, abstracting the design patterns (e.g. proxy) was done manually for the sake of this exercise. However, this abstraction can be automated once the concept of more complex design patterns are incorporated into RA. For that, design patterns need to be described in terms of their input (source) and destination as well as their access points.

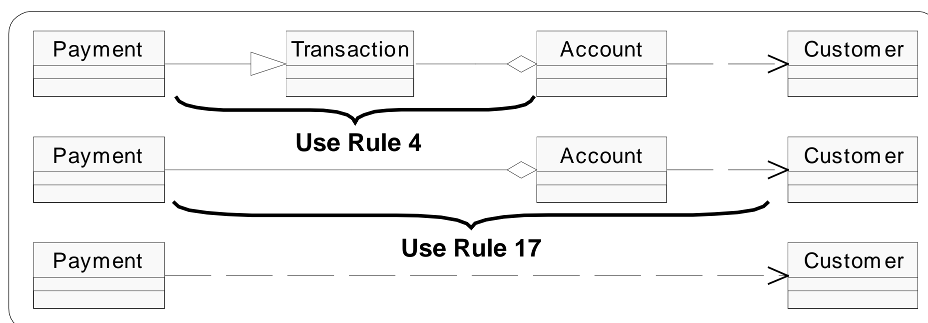


**Figure 7: Abstraction of Patterns through Rose/Architect**

[4] Note: We can simplify lower-level design patterns with higher-level surrogates

Another drawback of RA currently is that it can only be used to abstract information of class diagrams. Although, this technique could also be extended to abstract other types of views, it still does not help us when we want to compare modeling elements of different types of views. The next section addresses that issue.

## Patterns and Translation

Whereas abstraction dealt with the issue of simplifying views, translation deals with the issue of sharing modeling information between different types of views. For instance, above we made extensive use of classes and class diagrams which represent information in a structural form. Since structural views are usually insufficient in representing the behavior of systems, behavioral views such as sequence diagrams are needed to fill the gap.

For instance, consider the constraints of the architectural layer pattern in Table 1 again (together with the constraints of the high-level design view). The architectural constraints there were by no means covering the complete behavior of layered architectures. A layered architecture not only restricts which layers are allowed to talk to each other, it also restricts the direction of the interaction. Although, a structural diagram such as a class diagram can depict some of the directional dependencies it still may omits others.

**Figure 9: Sequence Diagram about the Creation of a New Order (corresponds to lower-level diagram)**

**Figure 8: Abstracted Low-Level Class Diagram using Rose/Architect and Design Patterns**

Figure 9 shows a more complex behavior example. Here a UML sequence diagram is used to describe one possible scenario of creating a new customer order. After some user input is validated, it is verified whether the customer exist. If not, a new customer is created and after that a new order is created. This scenario depicts some behavioral aspects of the lower-level design view (Figure 5). As such we can use Figure 5 to automatically verify the calling dependencies between the classes (or objects) which in this case would not reveal mismatches. This sequence diagram also conforms to our architecture since all layer constraints are observed (both could be verified automatically). Since the structural view is quite ambiguous in terms of the behavior of components, we can again make use of our knowledge of patterns to refine the behavioral information.

Figure 9 makes use of the classes *Order Repository Proxy*, *Network*, and *Oracle DB* and in *Mapping* we found that those classes correspond to the *Proxy* design pattern. Figure 10 shows both the structural and behavioral definition of the *Proxy* pattern as it is found in [Buschman et al 1996]. In effect, the behavioral definition is a translation of the structural one. Thus, we may now use that knowledge to verify the correctness of Figure 9.

Since the level of abstraction between the Proxy definition in Figure 10 and the Proxy instantiation in Figure 9 are not the same, we need to abstract Figure 9 first. Basically, we can use the Rose/Architect concept to abstract *Network* away buy merging it into *Order Repository Proxy* (this is valid since Network is part of the Proxy). After that, a direct comparison between the definition and instantiation is possible. In this case no mismatches are revealed.

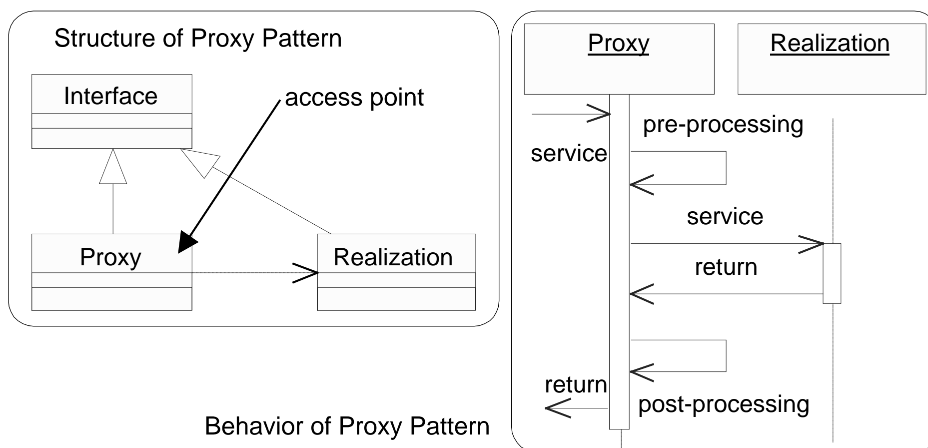


**Figure 10: Structure and Behavior of Proxy Pattern [Buschman et al. 1996]**

Due to limitation in space we cannot further describe this process in more detail. Please refer to [Egyed 1999a] and [Egyed 1999b] for more information.

## 6. Related Work

The absence of view integration is not a new discovery. Quite to the contrary. Many model descriptions talk about the need of keeping the views consistent. Sometimes, process models provide additional guidelines on what tasks one can do to improve the conceptual integrity of architectures. For instance, a case study in using the WinWin Spiral Model [Boehm et al 1998] suggests using Architecture Review Boards [AT&T 1993] after the LCO (life-cycle objectives) and LCA (life-cycle architecture) stages to verify and validate the integrity of the analysis and design. A similar viewpoint can be seen in countless other research work:

- [Sage-Lynch 1998] describe various aspects of integration (enterprise wide). They frequently stress "the important role that architecture plays in system integration." They present the need for three major views: enterprise view, systems engineering and management view, and technology implementation view – and they stress to ensure consistency among these views.
- [Rechtin 1991] emphasizes strongly the validity and consistency of requirements as well as the interface definitions. He further suggests the need for problem detection and diagnosis.
- [IEEE 1998] speaks of *Architecture Evaluation*. "The purpose of evaluation is to determine the quality of an architectural description, and through it assess the quality of the related architecture." They further state the need of evaluation criteria against which the architecture can be verified.
- [Nuseibeh 1995] wrote that "inconsistency is an inevitable part of a complex, incremental software development process" and that "the incremental development of software systems involves the detection and handling of inconsistencies."
- [Perry-Wolf 1992] realized the importance of software architectures early on and they state as one of the four major benefits of architectures that they are "the basis for dependency and consistency analysis."
- [Shaw-Garlan 1996] describe architecture very provocatively as being "a substantial folklore of system design, with little consistency or precision." They further state that "software architecture found its roots in diagrams and informal prose. Unfortunately, diagrams and descriptions are highly ambiguous."

These references, and many more, talk about the need for (or lack of) integration. Nevertheless, they usually do not describe the involved activities in detail (with some exceptions). Sometimes this is done on purpose in order not to favor a particular integration approach. On the other hand, those techniques that are sometime suggested are often only aimed at making people talk to each other. For instance, the Architecture Review Board [AT&T 1993] or the Inspection Process [NASA 1993] are primarily tailored at getting the most capable people together so that they may share

information. These techniques may follow a defined process (e.g. checklists) and may yield very effective results but the actual activities of identifying and resolving defects are still done manually without much automated assistance.

## 7. Conclusions

This work described the causes of architectural mismatches in UML views and showed how integration techniques can be applied to identify and to resolve them in a more automated fashion. We presented this work in the context of the Unified Modeling Language (UML) and its views (diagrams) and used an example to guide through the major stages of view integration. The view integration framework presented in this paper is not restricted to UML. It has also been applied to other models and views (e.g. architectural description languages).

This work further presented the use of patterns in analyzing the conceptual integrity of software system models. Since patterns are well described and documented, frequently both in structure and in behavior, we can make use of that knowledge for view analysis. As such, we showed that knowledge about patterns could be used to map views (cross-referencing of related modeling information) as well as to transform information from one view to another. For the later we showed both an abstraction (Rose/Architect) and a translation technique.

Although, the view integration framework was created with the intent of supporting automated view analysis, manual intervention was necessary from time to time. Since this paper focused on patterns only, other integration techniques were not discussed (see also [Egyed 1999b]). Issues we still have to address are:

- Finding (or developing) integration techniques covering a wider range of views
- Dealing with scalability
- Adding more precision to UML to clarify ambiguities.
- Addressing the issue of automatically supported mismatch resolution (as compared to just automated mismatch identification)

Despite those problems, we have made extensive progress in this field and we feel that the benefits of using integration techniques, such as the ones discussed in this paper, are immense. We have shown that it is possible to (at least partially) automate the task of mismatch identification and since computers are clearly much more efficient in comparing views, this implies substantial savings in manual labor. Another benefit of our approach is that mismatches may be identified as early on as they are created. Every time new data is added to the model, tools  can validate them.

## 8. References

AT&T (1993) "Best Current Practices: Software Architecture Validation," AT&T, Murray Hill, NJ.

Egyed, A. (1999a) "Automating Architectural View Integration in UML," submitted to ESEC/FSE'99, http://sunset.usc.edu/TechRpts/Papers/usccse99-511/usccse99-511.pdf.

Egyed, A. (1999b) "Integrating Architectural Views in UML," Qualifying Report, Technical Report, Center for Software Engineering, University of Southern California, USC-CSE-99-514, http://sunset.usc.edu/TechRpts/Papers/usccse99-514/usccse99-514.pdf.

Egyed, A. and Kruchten, P. (1999) "Rose/Architect: a tool to visualize software architecture", Proceedings of the 32nd Annual Hawaii Conference on Systems Sciences.

Booch, G., Jacobson, I., and Rumbaugh, J. (1997) "The Unified Modeling Language for Object-Oriented Development," Documentation set, version 1.0, Rational Software Corporation.

Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) "A System of Patterns: Pattern-Oriented Software Architecture," Wiley.

Boehm, B., Egyed, A., Kwan, J., and Madachy, R. (1998), "Using the WinWin Spiral Model: A Case Study," *IEEE Computer*, July, pp. 33-44.

Gamma, E., Helm, R., Johnson,R., Vlissides, J. (1994) "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley.

NASA (1993) "Software Formal Inspection Process Standard," NASA-STD-2202-93.

Nuseibeh, B. (1995) "Computer-Aided Inconsistency Management in Software Development," Technical Report DoC 95/4, Department of Computing, Imperial College, London SW7 2BZ.

Rechtin, E. (1991) "System Architecting, Creating & Building Complex Systems," Prentice Hall, Englewood Cliffs, NJ.

Perry, D. E. and Wolf, A. L. (1992) "Foundations for the Study of Software Architectures," *ACM SIGSOFT Software Engineering Notes*, October.

Sage, Andrew P., Lynch, Charles L. (1998) "Systems Integration and Architecting: An Overview of Principles, Practices, and Perspectives," Systems Engineering, The Journal of the International Council on Systems Engineering, Wiley Publishers, Volume 1, Number 3, pp.176-226.

Shaw, M. and Garlan, D. (1996) "Software Architecture: Perspectives on an Emerging Discipline," Prentice Hall.

Siegfried, S. (1996) "Understanding Object-Oriented Software Engineering," IEEE Press.