Pattern for Structuring UML-Compatible Software Project Repositories

Pavel Hruby
Navision Software a/s
Frydenlunds Allé 6
2950 Vedbaek, Denmark
E-mail: ph@navision.com
Web site: www.navision.com/services/methodology/default.asp

Acknowledgement. I would like to thank to Paul Dyson of the Big Blue Steel Tiger, England, for shepherding the article, and for his useful suggestions and comments. I do, of course, take full responsibility for any omission or errors.

Problem

Have you ever tried to describe your software system with UML and struggled to get an overview of the completeness and consistency of the specification? How can we write specifications, where the required information about software and management products is easily located and where closely related information is linked together?

Has your software development method sometimes forced you to create development artifacts you did not need? Have you ever wanted to specify an unusual feature in the system, and found that the method did not suggest any suitable development artifact, diagram or document?

Have you ever wanted to know a simple rule for extending and contracting the method in a consistent manner?

Context

You specify a software system with the Unified Modeling Language (UML) [3] and use some of UML-compatible software development method such as the Rational Unified Process [4], Catalysis [5] and Fusion [1]. During the software development process, you identify certain specifications concerning the software product. This information could be very general, such as the vision of the product, or very concrete, such as the source code. Other examples are use cases, object collaborations and class descriptions. You also specify information about management products, such as projects, project plans, organizational structures and job descriptions.

The UML does not specify how to structure the information describing the software system. The standard method can be used as a general guideline, but it often does not match the needs of a specific project.

Forces

1. In order to improve a development process, an organization will generally either adopt and customize a standard development method, or develop its own method. Part of the method consists of descriptions of development artifacts to be delivered by each project. However, the projects vary in size and complexity, therefore it is not possible to determine a set of development artifacts that fits all the projects.
2. Because of force 1, the organization wants to customize the development method at the beginning of each project. However, unexpected changes during projects prevent any up-front determination of which kind of customization (which development artifacts) will be needed.
3. Because of force 2, the team decides to use a standard development method and customize it on the fly during the project. This means disregarding some development artifacts and possibly including new development artifacts not specified by the method. However, the team does not know any simple rule for contracting or extending the method in a consistent manner.

4. Because of force 3, the project repository contains a set of development artifacts that reflects the project needs at each specific time, but which might be different from what a standard method requires. The team wants to get an overview of the completeness of the specification and the consistency between software development and management artifacts in the project repository. However, the checklists of the standard method cannot be used, because the method was not the driving force behind the selection of the development artifacts in the repository.
5. In short, the team wants to use a standard development method, but it cannot, because it is too restrictive.

Solution

Do not concentrate on what kind of UML diagrams to draw, but concentrate on what kind of system information you want to specify. In other words, make a distinction between the design artifact (the information itself) and the representation of the information (the UML diagram, CRC-card, table, text).

Split the software description into small chunks of information according to certain views and levels of granularity. Describe each chunk (the intersection between the view and the level of granularity) by four types of development artifacts: the classifier relationships, classifier interactions, classifier responsibilities and classifier lifecycles. In UML, classifiers are class, object, interface, use case, subsystem, component, node and datatype. Use stereotypes to extend the semantics of UML classifiers, when using them in specific views. For example, the project and team can be modeled as stereotyped classes, and documents as stereotyped components.

The artifact called *classifier relationships* specifies static relationships between classifiers, such as relationships between objects, components and database tables, or the relationships between use cases and actors. The artifact called *classifier interactions* specifies interactions between classifier instances. Examples of this artifact are object interactions and interactions between the system and actors. The *classifier* specifies classifier responsibilities and static properties of classifier interfaces. These include object or component operations with preconditions and postconditions, input and output events, and attributes that can be read and set. Examples of classifiers are class, subsystem, system, actor, module and test. The *classifier lifecycle* specifies classifier state machine and dynamic properties of classifier interfaces, for example, the allowable order of operations and events.

The solution is illustrated in Fig. 1.

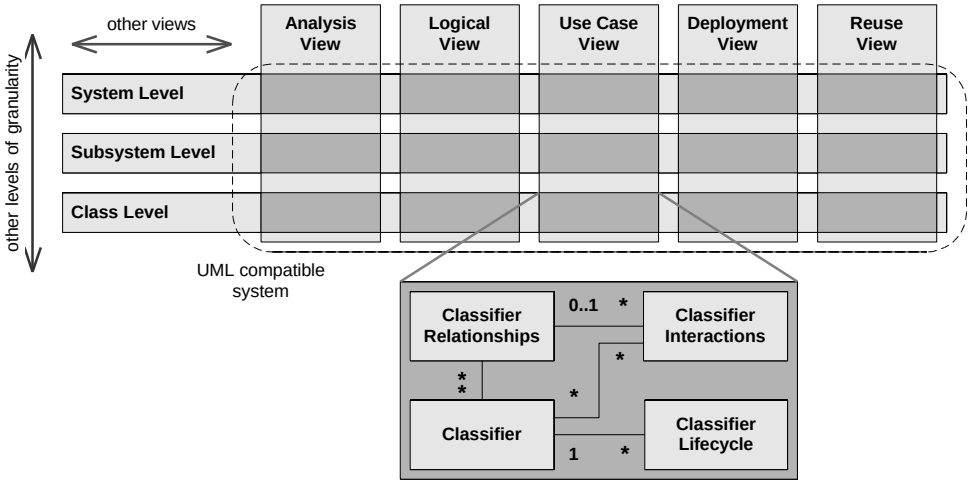

Fig.1. In each view and at each level of granularity the software system is described by four development artifacts.

Table 1 in the appendix illustrates the semantics and representation of development artifacts in the logical and use case views.

A Story

This section is written as a list of problems and the corresponding solutions in which the pattern is used. Examples are based on development of C/SIDE, a development environment for our financial system.

About four years ago, we at Navision started to use the Fusion method [1]. We decided to use UML [3] instead of the original Fusion notation and add use cases to the Fusion process. We made a simple repository (a Lotus Notes database) for the design artifacts. We launched a pilot project and filled the repository with a number of development artifacts. We faced several problems.

Problem 1: Many development methods and CASE tools give the repository entries the same names as UML diagrams, such as Class Diagram, Sequence Diagram, Collaboration Diagram, State Diagram. However, the information specified by the Sequence Diagram is the same as the information specified by the Collaboration Diagram. Moreover, sometimes it is useful to use text instead of diagrams. For example, sometimes it is easier to write a list of Use Cases with Actors (text) instead of drawing the Use Case Diagram. Sometimes, according to the Fusion method, we can use the BNF (Backus-Naur form) to specify the allowable order of system operations, although the same information can be expressed by the UML State Diagrams. The problem was that users of the repository wanted to search for specific information, not for specific diagrams.

Solution: Make a distinction between the information and the representation of the information. In our case, we used the word "model" for the information, and the word "diagram" if the information could be represented graphically. For example, the Interaction Model[1] can be represented by the Sequence Diagram, the Collaboration Diagram, or by text (a user story). The Lifecycle Model can be represented by a Statechart Diagram, the Activity Diagram, the BNF, and the State Transition Table.

Focus on the information, which can be represented in various ways, has additional benefit: we could attach additional attributes describing the information, such as author, version, status, last time modified and by whom.

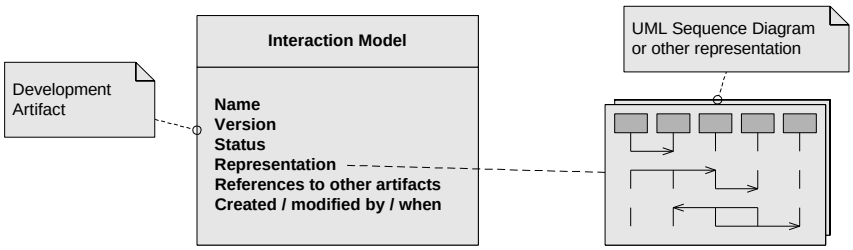

Fig. 2. UML diagrams represent some development artifacts

Problem 2: UML diagrams cannot represent every kind of design information. Some information cannot be represented in diagrams, it needs to be represented by text or tables. Examples are System Responsibility (a paragraph or two specifying the system purpose), Subsystem Responsibility (services provided by the subsystem, services required from other connected subsystems and abstract attributes that can be read and set), System Operation (a system interface operation with preconditions and postconditions) and Use Case (a text describing the use case).

Solution: Useful project repository must be able to contain design artifacts that are not UML diagrams. These design artifacts often correspond to UML classifiers (Class, Subsystem, Node, Component, Use Case and Datatype).
Note: in a sample of 400 artifacts of our largest project, about 40% of development artifacts were UML diagrams and 60% were represented by text.

[1] The term Interaction Model will be refined in Problem 4

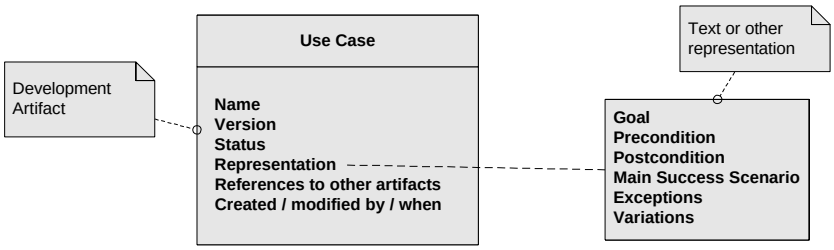

Fig.3. Text or tables represent other development artifacts

Problem 3: Static Structure Models at different levels of granularity describe different information (they have different content). The Static Structure Model, represented by the UML Static Structure Diagram, can describe relationships between classes, but also the system architecture (if the entities in the diagram are subsystems) and the system context (if the entities in a diagram are the system being developed and other connected systems). The fact that these development artifacts have the same name, but different content, is confusing.

Solution: Distinguish between development artifacts at various levels of granularity. Examples of levels of granularity can be the class level, the subsystem level and the system level. The number of levels of granularity can vary from case to case; it depends on the complexity of the system being designed. Instead of the single artifact type Static Structure Model, use the artifact types Class Relationships, the Subsystem Relationships and the System Relationships.

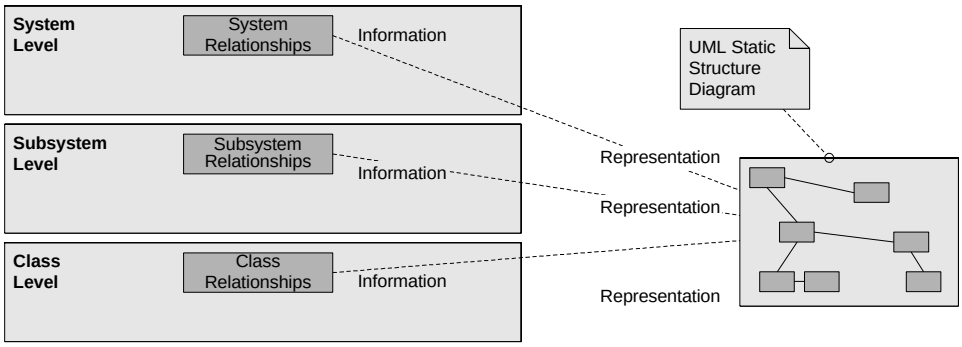

Fig.4. UML static structure diagrams at different levels of granularity specify different system information

Problem 4: Interaction Models and Lifecycles at different levels of granularity describe different information (they have different content). The Interaction Model at the class level describes interactions between objects. The Interaction Model at system level describes interactions between the system and actors (an instance of a use case). The Lifecycle model at the class level describes the class state machine, but the Lifecycle model at the system level describes the allowable order of system interface operations and events. The fact that these development artifacts have the same name, but different content, is confusing.

Solution: At each level of granularity, the system can be described by four types of development artifacts: the classifier relationships, the classifier interactions, the classifier responsibility and the classifier lifecycle. The number of views and levels of granularity can vary from case to case; it depends on the complexity of the system being designed.

In UML, the classifier relationships can be represented by a set of static structure diagrams (if classifiers are subsystems, classes or interfaces), a set of use case diagrams (if classifiers are use cases and actors), a set of deployment diagrams (if classifiers are nodes) and a set of component diagrams in their type form (if classifiers are components). The classifier interactions can be represented by sequence diagrams or collaboration diagrams. The classifier responsibility can be represented by structured text, for example, in the form of a CRC card. A statechart diagram, an activity diagram, a state transition table and Backus-Naur form can represent the classifier lifecycle.

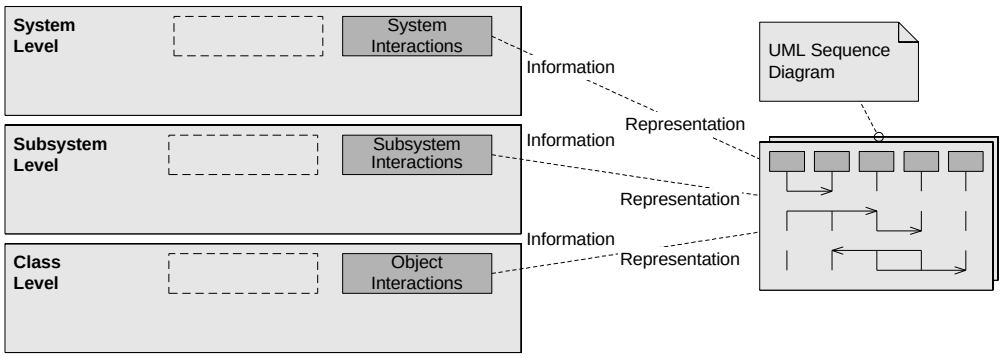

Fig.5. UML sequence diagrams at different levels of granularity specify different system information

Problem 5: The repository should be able to contain both precise and imprecise development artifacts. One of the good practices in the system specification is precision: all development artifacts at all levels of granularity should *precisely* describe the system; they differ only in the level of detail [5]. For example, the System Responsibility is a precise description of what the system does, but the System Responsibility does not contain very many details. The source code is also a precise description of what the system does, and contains many details. But both the System Responsibility and the Source Code are *precise* descriptions of the system. Sometimes, however, it is useful to keep imprecise information in the project repository as well.

Example A: The initial class model of a Petrol Station System contains the following objects: the tank, pump, meter device, switch and the petrol station itself. After some analysis and development, the model of the petrol station contains the following objects: the tank, gun, meter device, the display, the transaction manager and the database. The original model is no longer the precise description of the system and should be removed. However, they are good reasons to still keep it in the repository (it is an "as-is" analysis model).

Example B: The initial specification of the Consistency Checker component contains a class diagram with objects RuleEngine and RuleRepository and a scenario describing interactions between these objects. After some analysis and development, the Consistency Checker specification contains the same class diagram, but a different scenario. The original scenario is no longer a precise description of the Consistency Checker and should be removed. However, they are good reasons to keep it in the repository (for example, potential reuse).

Solution: Split the system description to the logical view, containing the precise specification of the system, and the analysis view, containing the analysis development artifacts. The analysis view contains analysis classifier relationship models, classifier interactions, classifier responsibilities and classifier lifecycles at various levels of granularity. For example, the analysis system responsibility, the analysis class and even the analysis code (usually called prototype).

Note: Many standard development methods, such as Rational Unified Process and Fusion, contain only analysis static structure model, but not analysis interaction models, lifecycles and responsibilities. If we use the pattern, the analysis view is able to hold the same set of development artifacts as the logical view.

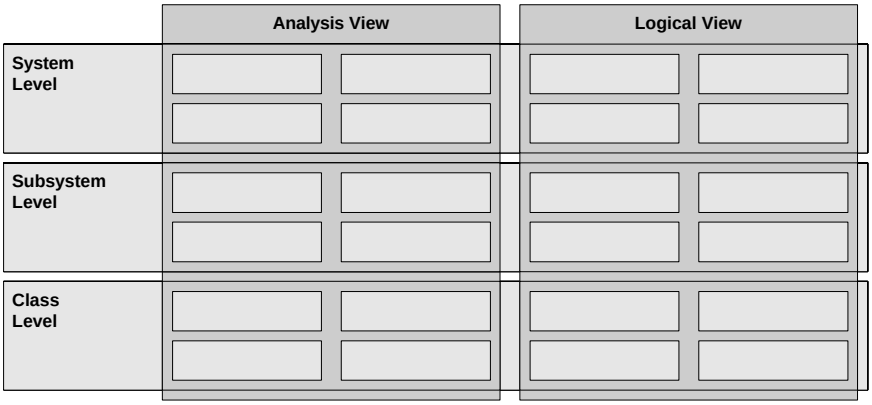

Fig. 7. UML diagrams in different views specify different system information

Problem 6: How to specify typical order of use cases? The UML use case diagram contains generalized usage scenarios, called use cases, and their relationships with the user roles and other connected systems, called actors. The use case diagram describes static relationships between use cases and actors. However, an actor can use a system in a way that initiates use cases in a particular order. Such a scenario – a typical sequence of use cases – can provide useful information about the system, however, the UML does not suggest any convenient way to show it.

Solution: The use case diagram contains classifiers (the use cases and actors) and static relationships (associations and generalizations) between them. According to the pattern, every static model has associated a number of interaction models. The static use case model has associated the use case interaction model, specifying the use case instances and actor instances, links and messages. The use case interaction model can be shown as sequence and collaboration diagrams in which classifier roles are use case roles. Example of such diagram is in Fig. 8.

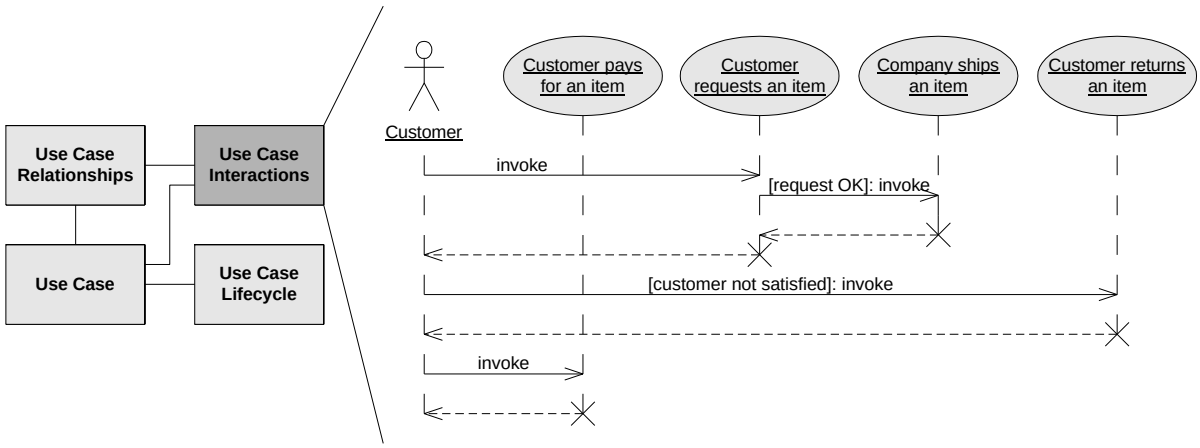

Fig. 8. UML use case interaction diagram specifies an example sequence of use case instances

Problem 7: How to relate development and management artifacts? In addition to the development artifacts, our repository contained two kinds of management artifacts: project and task; text specifying the project's and task's purposes. Developers had the option of relating development artifacts to tasks and projects and, in this way, structuring the repository and tracing the management information. Managers later required the project plans be added to the repository and related to the project descriptions. Can the pattern give a hints on how to structure the projects, tasks, project plans and other management artifacts?

Solution: Projects and tasks are classifiers at two different levels of granularity in the 'project' view. The artifacts project and task specify the project and task responsibilities. The static relationships between projects and tasks are usually specified by PERT charts. The development scenario - project plan - is specified by Gantt charts. The Gantt chart is a specific example of the interaction model: it contains project and task instances and their creation and deletion. Moreover, the project and task might have lifecycle, describing possible project and task states, such as: "not started", "in progress", "delayed", "finished".

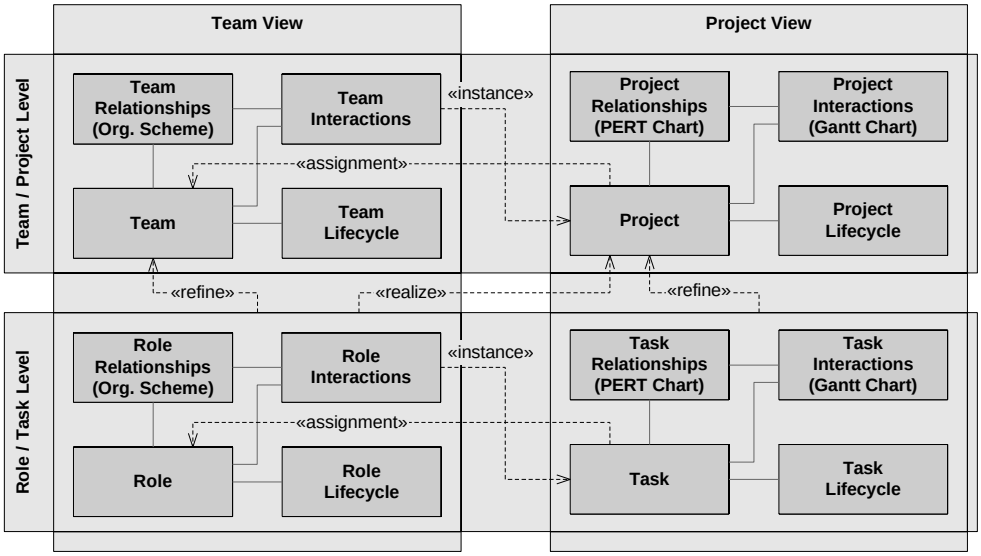

Fig. 9. Structuring management artifacts

Known Uses

This section lists three projects where the pattern has been used, along with benefits and problems that application of the pattern caused.

1. Navision Financials 1.3, C/SIDE (a development environment for our financial system). We used a Lotus Notes database as a repository of development artifacts.
   Benefits: we identified development artifacts that are not included in any software development method, but that were useful in our context. They were the use case interaction model (a sequence diagram describing interactions between use cases) and the interface lifecycle (the permissible order of interface operations).
   Problems: Many entries (about 30%) were notes and comments, such as various documents, links to useful web pages and various tricks. The pattern was a backbone to the repository, but many useful development documents were notes and comments not structured by the pattern.

2. Base Camp (a prototype of our new financial system).
   We used an atypical development scenario: we made a Vision & Scope (the System Responsibility), then code, and at the end, the documentation. The purpose of the documentation was to illustrate the design to people outside the development team.
   Benefits: At the beginning of the project, the project plan was based on the pattern. The pattern helped to identify the work packages and relationships between them. At the end of the project, the pattern helped to identify the relevant development artifacts to be included in the documentation.
   Problems: The pattern does not say explicitly how to structure the paper-based design document. The pattern shows relationships between development artifacts as a graph and does not suggest how to build a “hierarchical tree”, such as a table of content of the “functional specification” document, or, how to build a navigation tree in a

CASE tool. Finding a suitable transformation of the graph to the hierarchical tree is sometimes not straightforward. For example, the Rational Unified Process suggests building a hierarchical tree from use cases at the top and decomposing them into collaborations and classes. The Fusion method suggests the system operation at the top of the tree. The Catalysis method suggests using an abstract type (that is, a system or subsystem responsibility) at the top of the tree. In the Base Camp, we used the Catalysis approach with an abstract type at the top. In the C/SIDE project, we used several hierarchical trees, between which users could switch. We used the above-mentioned trees, plus the tree with a project at the top (showing the artifacts that belong to a certain project and task) and the author at the top (selecting the artifacts modified by a specific user).

3. The NOAH Software System (an integration framework that allows office management systems, audiological measurements and hearing instrument fitting to share a common database).

   The aim was to establish among developers a shared knowledge about the existing system. We documented "interesting" parts of the system with Rational Rose and organized a series of workshops (seminars) where each developer talked about the part of the system he knew. Some of the developers had knowledge about the Rational Unified Process (RUP), but we used only a very small subset of the RUP development artifacts. We used the pattern to identify the development artifacts that were useful in our context.
   Benefits: the pattern helped to shrink the RUP to the absolute minimum of artifacts that developers considered useful.

   Problems: The problem was an interaction diagram that spans several subsystems. The pattern makes a relationship between the artifacts Classifier Relationships and Classifier Interactions. The advantage is a simple consistency rule between these artifacts: each entity that appears in the classifier interactions should also appear in the classifier relationships and vice-versa. In other words, if a class in the relationship model is not used in any interaction model, or, if an interaction model uses an object with unspecified relationships, it is a warning.

   However, in some cases, this close relationship between these two artifact types might be a problem: sometimes we want to specify *object* interactions that span several *subsystems*. Sometimes a large sequence diagram that shows how a certain event is propagated from the user interface, through several layers of business logic to the database, is useful; and it shows concrete *objects* that were involved in this operation. If we applied the pattern to this case, we would have to draw more than one interaction diagram. Firstly, we would need a subsystem interaction diagram that shows how the event is propagated from one subsystem to another. We would also require object interaction diagrams that show the effect of the event inside each subsystem. Because we draw a single large object interaction diagram, we do not have a single class diagram to which it can be related. In other words, we could not use the simple consistency rule and would have to check consistency against several class diagrams.

References

[1] Coleman, D. et al: Object-Oriented Development: The Fusion Method, Prentice Hall, 1994
[2] Hruby, P.: Structuring Software Development Artifacts with UML, JOOP, 12/9, February 2000
[3] UML 1.3 specification OMG document ad/99-06-08, OMG, 6 August 1999
[4] Kruchten, P.: The Rational Unified Process, Addison-Wesley, 1998.
[5] D'Souza, D., Wills, A.: Objects, Components and Frameworks with UML: the Catalysis Approach, Addison Wesley, 1999.

Appendix

The table below specifies the semantics of development artifacts in the logical and use case views and at the business, system, subsystem, class and code levels of granularity. Each field in a table outlines the semantics of a development artifact and its typical representation. As noted in Solution and Story (Problem 1), the development artifacts can be represented in a number of ways. The table outlines only the representation which is the most usual.

		Logical View		Use Case View	
		Static	Dynamic	Static	Dynamic
Business Level	Design	Relationships between Organizations (Class diagram)	Interactions between Organizations (Text)	Relationships between business processes (Use Case Diagram)	Scenario consisting of business process instances (Activity Diagram)
	Specification	Responsibilities of Organization and Business Partners (Text)	Workflow (Activity Diagram)	Business Process Specification (Text)	Business Process Lifecycle (State Diagram)
System level	Design	Relationships between Systems; System context (Class diagram)	Usage scenarios, User stories (Sequence diagram, Text)	Use Case and Actor relationships (Use Case Diagram)	Scenario consisting of Use Case Instances (Use Case Interaction Diagram)
	Specification	Responsibilities of System and Actors (Text)[2]	Order of system operations (BNF)[3]; Actor State Machine (State Diagram)	System Use Case (Text)	Use Case Lifecycle (Activity Diagram)
Subsystem level	Design	Subsystem Relationships (Package Diagram)[4]	Subsystem Interactions (Sequence Diagram)	Subsystem Use Case and Subsystem Actor Relationships (Use Case Diagram)	Scenario of Use Case Instances (Use Case Interaction Diagram)
	Specification	Subsystem Responsibility and Interfaces (Text)[5]	Order of Interface Operations (State Diagram)	Subsystem Use Case (Text)	Use Case Lifecycle (Activity Diagram)
Class level	Design	Class Relationships (Class Diagram)	Object Interactions (Collaboration Diagram)	Class Use Case and Class Actor Relationships (Use Case Diagram)	Scenario of Use Case Instances (Use Case Interaction Diagram)
	Specification	Class Responsibility (CRC Card, text)	Class State Machine (State Diagram)	Class Use Case (Text)	Use Case Lifecycle (Activity Diagram)
Code level	Design	Language Syntax (Text)	Source Code (Text)	[6]	[6]
	Specification	Language Term		[6]	[6]

[2] Also known as Vision & Scope
[3] Allowable order of system operations in the Fusion Method
[4] Package diagram is a static structure diagram in which most of the entities are packages
[5] Abstract Type in the Catalysis Method
[6] Use Cases as imprecise entities do not make sense at the code level where precision is essential.