

CATALYSIS – Practical Rigor and Refinement

Extending OMT, Fusion, and Objectory

Desmond D’Souza, Alan Wills

dsouza@iconcomp.com, alan@icon.demon.co.uk

Abstract

Over several years of applying, consulting, and training using the OMT and Fusion methods, we have evolved and successfully applied an integrated toolkit of techniques for developing systems from objects and components, together with heuristics and a supporting process. The method, called CATALYSIS, began in 1991 as a formalization of OMT, influenced by DisCo, VDM, and Fresco. Building upon “second generation” methods such as Fusion, CATALYSIS combines strengths in analysis and specification with a systematic treatment of architecture, refinement, and design. CATALYSIS advances beyond most current methods by defining methodical refinements from abstract specification to implementation, handling coarse-grained to fine-grained components, formalizing use-cases, recursively decomposing components into patterns of collaborating objects, refining the transactions between the collaborating objects, supporting traceability, and mapping design objects to roles in the specification. This paper describes the main features of CATALYSIS, and highlight those which differ from or extend the OMT and Fusion methods.

1.0 Introduction

What spurred this work...

Over several years of consulting and training, the authors have applied several existing object methodologies to system development, beginning with OMT and Booch. In doing so, we encountered problems of traceability to requirements, semantics for consistency checking, levels of granularity and refinement, and distinction between domain models, system models, and architecture [Dso94], [Dso93]. Most other development methods in use today have similar drawbacks. We evolved and successfully applied an integrated toolkit of techniques for building object systems, together with heuristics and a supporting development process. The method, called CATALYSIS, began in 1991 as a formalization of OMT, influenced by DisCo [Kur90], VDM [Jon86], and Fresco [Wil93]. Influenced further by second-generation methods like Fusion [Col93] and Syntropy [Coo94], it extends these by combining strengths in analysis and specification with a systematic treatment of refinement and architectural design. More details on CATALYSIS are available in our forthcoming book [Cat95].

OMT/Fusion: Strengths and areas for improvement.

OMT [Rum91] was published in 1991 and has become one of the most popular object-oriented notations and methods in use today. It includes a fairly detailed discussion of a continuous development process from analysis to implementation, adding detail to the models as it progresses. Its primary contributions are:

OMT contributions:

- A rich and expressive notation for the object model
- Emphasis on associations as a first-class construct in modeling
- The adoption of Harel-style state-charts for state-diagrams.

OMT weaknesses:

Its primary areas for improvement are:

- Integration of models: due to weak semantics associated with the models, there are no consistency and completeness checks across the models.
- Inadequate description of end-to-end functionality: the use of data-flow diagrams to express functionality proved to be a bad choice.
- Level of granularity: the method does not have a well-defined notion of refinement of the models across different levels of granularity.
- Systematic treatment of design: design often is a much richer activity than simply adding detail to the analysis models.

OMT has also undergone changes recently.

Some recent changes in OMT include: usage of Fusion-styled interaction diagrams for design, flow-graphs over the object model to indicate pre- and post-conditions, use-cases for requirements capture, more discussion of the development process, and ongoing integration with concepts and notation from the Booch method [Boo94].

Fusion contributions

Fusion [Col93] was published in 1993, and quickly made an impact on the landscape of object-oriented methods. The Fusion authors introduced several features which distinguished the method from others which had been published at the time, and recommend that users:

- Explicitly separate the domain model from system models
- In analysis, describe the behavior expected from the system using the system operation schema as the primary vehicle, and a lifecycle model as a secondary vehicle
- In design, describe internal behavior expected between objects, clearly capturing mutual visibility and lifetime decisions
- Use aggregation to visually contain compound relationships

Fusion provides reasonably clear guidelines for checking models informally for consistency, and to the development process. It suggests a recursive process of designing interactions and object modeling, and uses programming by contract as the underlying model for the implementation. This approach results in better understood and documented interfaces at all levels.

Fusion weaknesses

However, there were some areas in which we felt Fusion could be improved.

- The domain model itself describes “static” information in the domain. There is actually a continuum between persistent and transient, and the separation depends completely upon the interaction granularity being considered.
- The lifecycle model describes permitted sequences of events. However, the sequence expressions are quite limited in the presence of certain common kinds of multiplicities and interleavings, and consequently do not help with consistency and completeness checks in those situations.
- The operation schema contains some elements of formal description, but the method does not develop them far enough to allow thorough consistency checks.
- The method does not produce state models of any objects. This was probably because of the very design-like interpretation that other methods place on state-models, using them to discuss internal interactions between objects. However, there is much to be gained from interpreting the state-diagrams as simplified projections of the otherwise compound state of the system.
- While the models and notations for capturing interactions and visibility are reasonable, the guidelines for good design are inadequate.
- Despite the use of programming-by-contract, the notion of contracts is not utilized effectively in the design activities.

- The notion of aggregation could have been pushed further, starting with an interpretation of the system as an aggregate component. In order to understand the behavior of this component, we need to build a model for it, so we can express the behavior in terms of that model (operation schema). This would lead quite naturally to a recursive process for partitioning and design.
- In complex systems there may be significant transformations between analysis models and design. This is particularly true when striving for implementation components which are highly parameterized and configurable, or due to tight performance constraints. However, some of Fusion's consistency checks on design require very simple mapping back to the analysis models.
- Fusion does not directly support multiple views on a complex system. Most complex components have multiple interfaces, and each interface potentially justifies a different model.
- In design, Fusion does not adequately address issues of system architecture. These include issues such as hardware and software platform, databases models, local vs. remote processing, and the corresponding mapping of the analysis and/or design models to this architecture.

1.1 CATALYSIS: Main Features

Distinguishing features of CATALYSIS

Our overall approach has been to adapt the best known practical techniques for developing object systems, and to add a degree of rigor to them to gain the benefits of clarity and reduce their ad hoc nature. The distinguishing features of CATALYSIS are:

- *Clear separation of concerns*: we make a clear conceptual separation of descriptions into three layers of *what*, *who*, and *how*.
- Support for *incremental development* and *mixed description levels*: despite the underlying rigor, we do not mandate a rigid and formal process. Rather, we define the essential relationships between the models produced, then build a pragmatic process to support the realities of development. We also encourage combining formal and informal descriptions at any time.
- Integration of important forms of *abstraction* and *refinement*: specification of collaborative behavior separately from localized behavior, multiple views or "roles", synthesis of views based on *collaboration/design patterns*, deferral of message protocols, distinction of design types ("components") from model types, support for refinements and implementations beyond the current programming language notion of 'class', and support for architecture and design patterns in the recursive transformation of models from domain to implementation.
- Use of several proven *descriptive models and visual notations* with strong semantic *consistency and completeness criteria*, based upon a small set of "core" constructs. We provide heuristics for choosing appropriate constructs to depict specific views of a system.
- Potential for systematic *mechanical support and checking of semantics*: due to the underlying rigor, it is possible for a tool to support type checking, consistency checking, and test-case generation.
- Support for both *forward-engineering and re-engineering* of systems, thanks to the recursive relationship between models and the clear separation of specification types from design types at any level.

- Distinction between the scope of development efforts for *projects* (an application), *products* (an application over its lifetime), and *product-families*. We try to characterize the variability factors across each, and use these to define units of re-use and configurability, and to drive the system architecture.

Rigor, used properly, is very valuable.

In this paper we present an overview of the method, examples of its key constructs, and some of its underlying formalisms. Due to lack of space we will not be able to explore all these issues in depth. They are discussed in more depth in our forthcoming book [Cat95] and in several technical reports available from us. Although we present rigorous aspects in this paper, CATALYSIS does *not* prescribe blind formalisms, but prefers their pragmatic application. However, we have found that a rigorous basis is a valuable complement in many situations, and is essential for a method to be scalable.

How to read this paper.

This paper is organized as follows: summary of terms and process, core constructs, refinements, and design. Section 3.0 will describe application of the concepts to modeling a domain. Section 4.0 onwards will extend the concepts to modeling and design of a software system. Paragraphs related to comparisons with OMT and Fusion are highlighted in the margin notes. We suggest reading the summary of terms in Section 2.0, and then returning to it while reading the subsequent sections. The margin notes provide an overview of each section. We urge you to read all formal expressions as a complement to the informal prose which accompanies them as an integral part of the specification.

Detailed Outline of paper

1.0 Introduction

What spurred this work...

OMT/Fusion: Strengths and areas for improvement.

1.1 Catalysis: Main Features

Distinguishing features of Catalysis

Rigor, used properly, is very valuable.

2.0 Overview: Terms and Process

2.1 Basic Terms and Definitions

2.2 Development Process Philosophy

No single development process can be suitable to all situations.

We define relationships between deliverables and a strawman process.

2.3 Overview of Development Principles and Process

A thumbnail sketch of our development principles.

We distinguish 3 levels: what, who, how

“What” describes joint behavior

“Who” localizes responsibilities

“How” refines collaborations

This formalizes use-cases.

The development process covers domain models to implementation.

3.0 Specifying Collaborations — “What” at the domain level

OMT/Fusion: the Object Model

3.1 Objects and Transactions

Objects interact through transactions.

Abstract transactions may be composed of other transactions.

Transactions may be specialized.

Objects can be “abstract”.

Objects and transactions apply at all levels.

3.2 Specifications

3.2.1 Object Specification

A type specifies the behavior of a set of objects.

Subtypes are subsets of objects.

3.2.2 Transaction Specifications need Models

Preconditions and postconditions specify transactions.

A rigorous specification needs a model.
OMT/Fusion: interpreting models as queries

3.2.3 Specification Models

Models are sets of queries
Models define precise mutual vocabularies.
Queries can themselves yield interesting types.

3.2.4 Formal Transaction Specification

Transaction-specs use predicate logic on queries.
Postconditions relate before and after states.
Specification vs. design types
Multiple pre/post pairs may be combined.

OMT/Fusion: Transactions

3.2.5 Two Kinds of Transactions

There are two main kinds of transactions.
Joint transactions specify combined effects.
Localized transactions represent services of an object.
Both transactions may be refined temporally.

3.2.6 Associations

Model queries can be depicted as associations.
Associations can be “navigated” by name.
The semantics are the same as for model queries.
Queries can return sets and NIL.
Queries can be parameterized.
Implementations are encapsulated; specification queries do not have to be.

3.3 Refining Collaborations

Refining a transaction induces a refinement of the model.
In this case, we refine the model with SaleRecord.
New objects may exist in postconditions.
A collaboration may refine a transaction.
A collaboration constrains its transaction sequences.
This refinement is traceable and can be checked for correctness.
Refinement is an essential part of Catalysis.
OMT/Fusion: Refinement
Fusion and framing

3.4 Roles and Collaboration Patterns

An object can play many roles; each collaboration may need a different model.
e.g. Any retailer plays a certain roles with respect to its suppliers.
Temporal constraints are complex, and do not impose concrete sequences.
OMT/Fusion: Lifecycle model

3.5 Composing Collaboration Patterns

We can compose these two views.
The two models are combined and constrained.
The behaviors in both views must interact.
This refinement retains all previous guarantees.
We combined views and instantiated a parameterized model.
Multiple views on an object are essential to support *Design Patterns*.
OMT/Fusion: and multiple views.

3.6 Specifications, Collaborations, Roles and Design

A quick review of the constructs we have used so far.

4.0 Specifying Services — “Who”

4.1 Services

We can effectively localize responsibilities after understanding joint behaviors.
Localizing behaviors defines “services”.
In order to define a component’s services, we need a model of it.
We distinguish system specification types from their domain counterparts.
This model is drawn from the domain model.

4.2 Snapshots

Snapshots depict configurations of objects.
Scenarios, with accompanying snapshots, depict sequences of transaction occurrences.

All the models can be derived from scenarios and snapshots.
Snapshots also help understand model consistency.
Consistency rules are simple and clear.
CASE tools can readily support snapshots.
OMT/Fusion: scenarios, system interface, and models.

4.3 System Model

System models should follow the domain.
Design models should ideally follow the domain, but will not always do so.
Some development heuristics for system modeling.

4.3.1 Inputs and outputs of the System

Services and their inputs have specified effects on the receiver's model.
“Out” parameters might effect the sender's model.
Outputs can be specified by the desired post-condition on external objects.
Outputs can also be specified by requiring output events to be generated.
Invariants can relate system model states to states of external objects.
GUI presentations can be defined by such invariants.
Implementing such invariants is an issue for architectural design.

4.3.2 Refinement of a joint-transaction to services

A transaction may be refined to a specific protocol of services.
A temporal constraint specifies this protocol.
We are bridging the gap from domain to system specification.
This induces a more detailed model of the system.
The refined model has its own invariants.

5.0 Design — “How”

5.1 Transformations in Design: Reification

Design might re-arrange the model.
We re-arrange this model looking ahead to de-coupling pumps from sales.
We could verify this re-arrangement formally.

5.1.1 Traceability by “Retrieval”

We can show that the re-arrangement is valid and establish traceability.
We can combine formal and informal descriptions of traceability.
This approach supports a variety of architectural choices.
The rigor is available for use when needed.

5.2 Stepwise Refinement and Re-Localization to Implementation

We have discussed several refinement steps so far...
And we can apply them recursively...
Even down to implementation...

5.2.1 Localizing before temporal refinement

Development could take many paths. We re-trace our steps and try another path.
We could have localized just very high-level services, like “dispense”.

5.2.2 Localization within FuelStationSystem

This service could be localized on Pump.
Here, supporting services have been only partially localized to the pump.
And other parts of the design localization have been deferred.

5.2.3 Operations: The Last Transaction Refinement

By applying another refinement...
And get the sequence <offHook;meter*;onHook> by a different development path.
In certain architectures, transactions which are refined may “disappear” in the implementation.
Similarly, specification types may also “disappear”.
And “messages”, at last!
We specify operations down to messages, and continue to use specification types.

5.2.4 Implementing Operations

Objects and operations are finally implemented in an OO language.
And these operations can also be specified .
Until they are all implemented as methods.

5.3 The Role of Refinement in Design

Clear refinement steps support traceability.
Actual development is rarely traced to elementary refinement steps.
Refinement can incorporate significant architectural transformations.

In reality, there are several valid and useful refinements.
 And elementary refinement steps will rarely be documented separately.
 Design will recursively apply these constructs, addressing several concerns
 OMT/Fusion: The Design Process
 Interaction and Visibility Graphs
 Persistent vs. transient is a matter of granularity.

6.0 Conclusions

2.0 Overview: Terms and Process

2.1 Basic Terms and Definitions

Term	Definition
Class	A programming language construct for defining how the state and behavior of a set of <i>objects</i> is implemented. A class may implement multiple <i>types</i> , and vice-versa.
Collaboration	A set of <i>transactions</i> which have some common purpose, with a common level of abstraction or detail, and involving objects playing different <i>roles</i> ; often corresponds to a temporal relation between a set of finer-grained transactions which meets the specification of some higher-level transaction.
Component	An <i>object</i> , possibly complex, which had definite responsibilities assigned to it, and which will have an implementation to support one or more interfaces. Not all components will be implemented as instances of classes.
Design	A recursive process of refinement and decomposition of transactions; a distinct level of description which addresses how the required behaviors will be provided by some pattern of lower-level <i>collaborations</i> and finer-grained <i>components</i> .
Design Pattern	A proven design technique, presented with a discussion of its applicability and trade-offs, which suggests a transformation from a specification to the next level of design, or from one design to another.
Design Type	A <i>type</i> introduced to circumscribe some portion of the (next level of) implementation, with an interface of <i>service transactions</i> ; design types take part in transactions. Members of this type will be identifiable <i>components</i> in the implementation.
Model Type	A type introduced as part of a specification model, purely to support a specification. Model types do not directly take part in transactions, but they can have <i>queries</i> , invariants, etc. Also called <i>specification type</i> .
Object	An individual with identity and behavior; a member of some types; an identifiable <i>component</i> with an interface; an instance of a <i>class</i> .
Query	A hypothetical read-only function modeling the state of an object and used only in pre/post conditions; often depicted as a diagrammatic link or a typed attribute.
Role	A place for a participant in a <i>collaboration</i> ; a <i>view</i> of an object from the perspective of some other object which collaborates with it. The mapping from design object to role is many-to-many at any point in time, and is dynamic.

Snapshot/ Instance diagram	A diagram of an instantiation of a type model at some instant in time, showing the interesting aspects of the objects, and the results of their specification queries (depicted as links and attributes).
Specification	A description of guaranteed behavior of some object, together with the conditions under which that behavior is guaranteed; often described with pairs of pre-conditions and post-conditions in terms of a specification model, sometimes with an associated temporal constraint.
Specification Model	A set of queries – called specification queries – which supports some specification; often depicted as types, attributes, and associations in a set of diagrams.
Specification Type	see <i>Model Type</i>
Transaction - joint and service	A unit of interaction or information exchange between participant objects playing some roles, with a specified effect on those objects. We support both <i>joint</i> transactions (multiple participants, described symmetrically with no distinguished receiver) and <i>service</i> transactions (attached to a distinguished receiver which is assigned responsibility for that transaction). Transactions can be refined in several ways. Our transactions do not require the atomicity and serializability properties required of traditional database transactions.
Type	A specification of externally visible behavior of objects – members of that type are all objects that conform to its behavioral specification. A type makes absolutely no statement about implementation.
Type Model	see <i>Specification Model</i>
Use case	A transaction which accomplishes a meaningful objective to an external user of a system; often a <i>joint</i> transaction, refined to describe the roles that the different participants play in the <i>collaboration</i> .

2.2 Development Process Philosophy

No single development process can be suitable to all situations.

The actual development process used on any project depends on several factors, including the project team, the nature of the problem domain, the schedule, familiarity and development maturity, and external constraints. We do not mandate a single development process to be followed under all situations.

We define relationships between deliverables and a strawman process.

Instead, we focus on the different work-products, their relationships to each other, heuristics for constructing these work-products, and rules for checking them for consistency and completeness. Within this framework, we define one possible development process and provide guidelines for adapting this process to specific project needs.

2.3 Overview of Development Principles and Process

A thumbnail sketch of our development principles.

We apply a set of principles across all levels of description. When constructing any *component*, we first broaden our scope to describe its joint behavior with other objects in its environment. We then negotiate the interface between the component and objects in its context, and localize the responsibilities and services of each. Separately, we design finer grained objects and interactions within the component that will

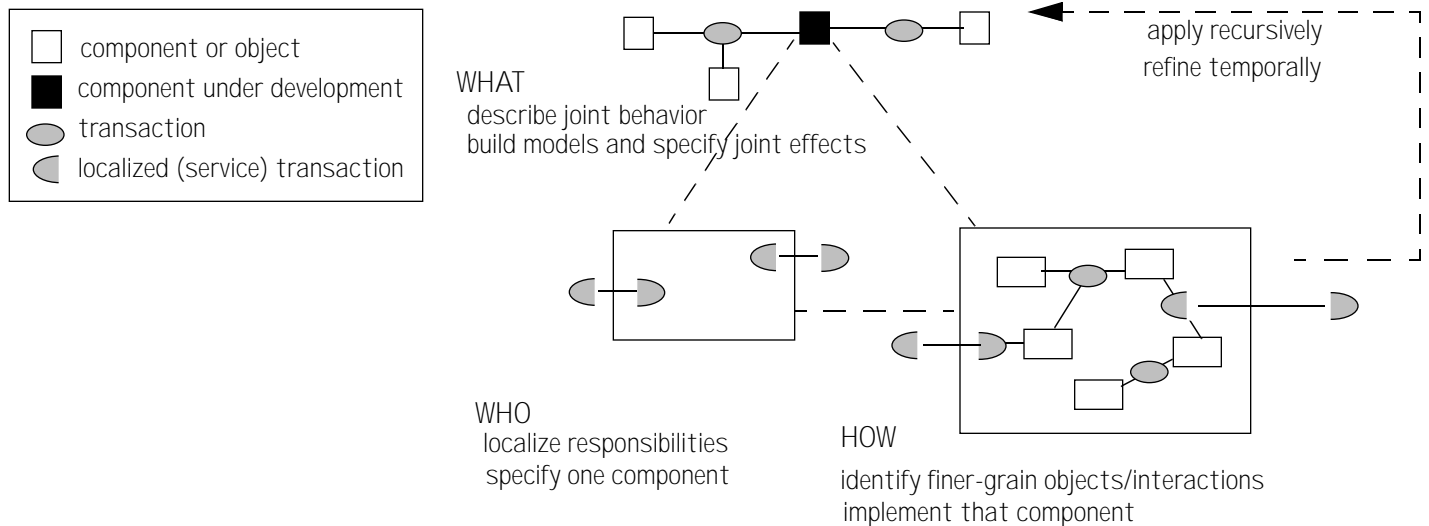


Figure 1 Development Process Levels: What, Who, and How

implement these services, including appointing specific internal objects designed to “receive” external requests on behalf of that component.

We distinguish 3 levels: what, who, how

“What” describes joint behavior

“Who” localizes responsibilities

“How” refines collaborations

This formalizes use-cases.

The development process covers domain models to implementation.

Hence, we are always recursively describing one of the following at a particular level of granularity, and not necessarily in this sequence:

- *What*: the combined behavior of a group of objects which participate in some joint transaction. In order to describe any single component, we must first understand the context in which it operates. Describing such joint behavior requires at least a hypothetical model of each participating component. This level of description often represents a particular *design* for some higher level transaction.
- *Who*: assign and localize responsibilities among the participants in a joint transaction, and design the interactions which will provide the required joint effects. This will eventually define the services provided by each, and compose those services in collaborations to implement the joint transaction.
- *How*: identify finer-grained objects which provide the services on behalf of any component, and refine the granularity of interactions. The resulting description can be recursively refined into further *what*, *who*, and *how*.

These three levels formalize the notion of “use-case” [Jac91]. A use case describes the joint behavior of a set of objects. It may be refined in terms of the granularity of interactions, as well as in terms of the actual objects providing the services required.

Our overall development process includes the following activities, illustrated in Figure 2. There is no strict sequence between the activities, but the deliverables produced have clearly defined and consistent relationships to each other. The overall progression is:

- Understand the application environment, including domain terms, concepts, and metaphors. Build a glossary, and an informal or formal model of the domain, so that subsequently developed terminology has a basis in the domain.
- Sketch out the system boundary and scope by building a system context model. This activity is the start of business (re-)design, since it re-defines roles and responsibilities, and may involve identifying deeper concepts and new metaphors.

- Build formal models of the system which support the needed interactions, and specify required behavior. Refine the interactions as needed, so as to make clear the guarantees and proper usage of the services.
- Develop architecture for implementation. Possibly transform the models, and trace and validate the changed models against the original models and specification.

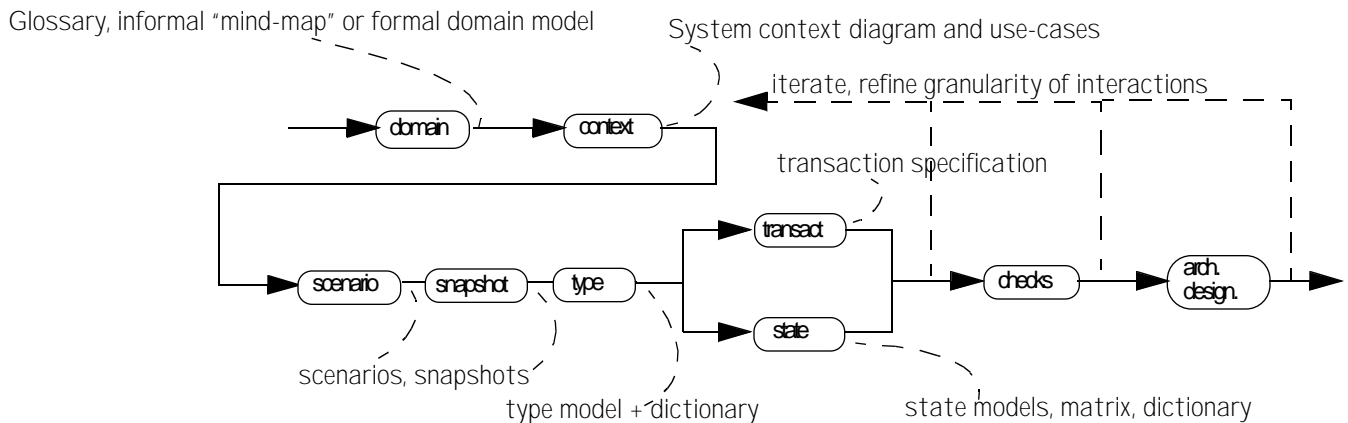


Figure 2 Development Process – Major Steps

3.0 Specifying Collaborations — “What” at the domain level

We will illustrate the key points of CATALYSIS with an example of a software system for a vehicle fueling station, with customers purchasing gas, and suppliers re-filling the station’s storage tanks. For a fuller description of a related problem statement, the user is referred to [Col93].

In this example, we begin by understanding the context within which the software system will be used, and model the objects in that domain and their transactions. We will introduce constructs as we use them in the domain model, but these constructs are applicable recursively at any level of system description. In general, the degree of rigor useful at the domain level will be less than at the system level.

The emphasis in this section is on describing what happens, or is required to happen, jointly between objects in this domain. We do not necessarily allocate responsibilities to specific objects, and may not even fully understand the responsibilities of the target system at this stage.

OMT/Fusion: the Object Model

OMT recommends building an object model to capture “static” objects and associations in the problem domain, and to use this as the basis for system analysis.

Fusion recommends building an object model of the problem domain to capture the “static” objects and their relationships. The model is essentially an E-R model. It describes the structure of problem domain objects but does not capture any of the behaviors in the problem domain which justify those structures in the first place. We address this by considering collaborative behaviors in the problem domain even before attempting to model it.

3.1 Objects and Transactions

Objects interact through transactions.

An object-oriented world is described as a network of interacting objects. The interactions are often a combination of essential interactions and “design” decisions, including design at a business or domain level. A transaction is a unit of interaction between objects. Figure 3 depicts an occurrence of the sale transaction in our domain. In dif-

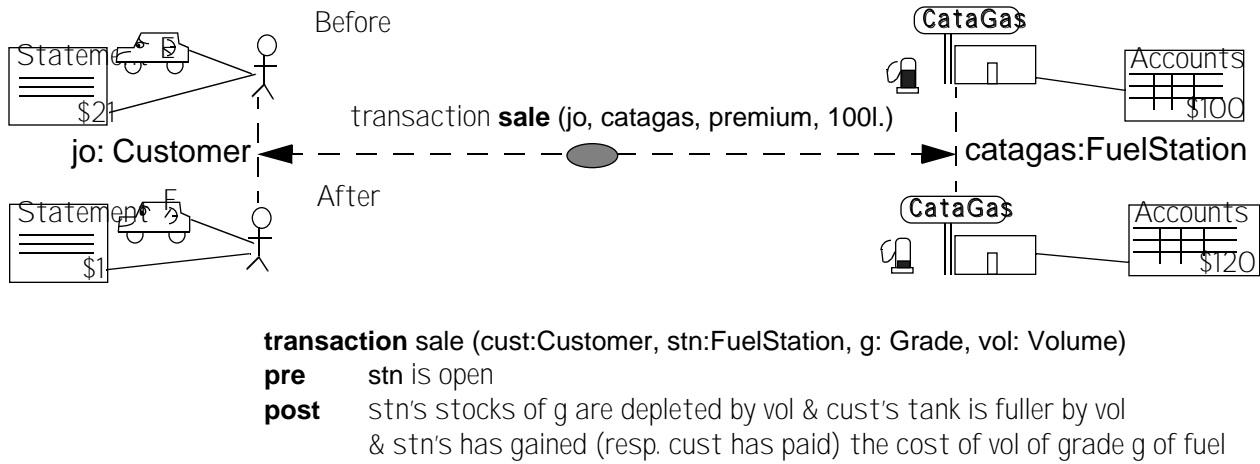


Figure 3 Transaction Example

ferent occurrences of a transaction, each of its roles — the “slots” for its participants — can be played by different objects. The details of the outcome depend on their states and types. Each transaction is specified in terms of its effects on the participating objects. Conversely, each object can be characterized by the transaction it can participate in.

Abstract transactions may be composed of other transactions.

A transaction may encompass a complete dialog of smaller transactions. The common object-oriented programming notion of a *message-response* pair is one special case of a transaction. In Figure 4, sale is an abstraction covering some sequence of authorize, dispense, pay. It may well turn out that sale and refill-storage do not occur as messages

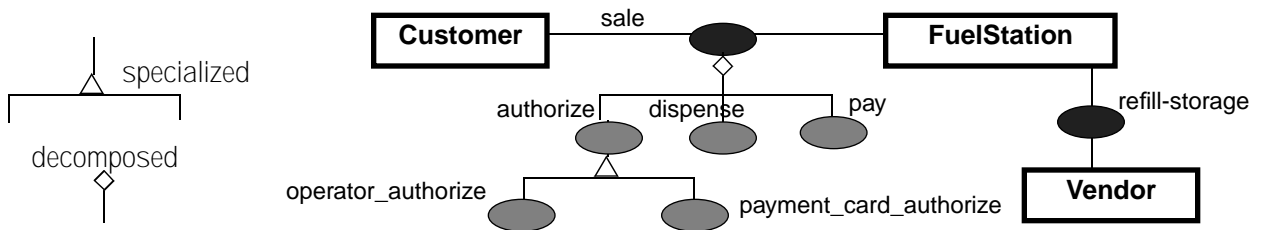


Figure 4 Transaction Decomposition and Specialization

anywhere in the implemented system, but only as abstractions.

Transactions may be specialized.

A transaction may also be specialized. A specialization guarantees that all behaviors of the more abstract transaction are also true of the specialization. Specializations of authorize (which switches on the pump motor) include operator_authorize and payment_machine_authorize (where the customer inserts a card or cash into a machine beside the pumps).

Objects can be “abstract”.

Similarly, an object can also be abstract. The word “object” describes any identifiable component, or indeed a whole system, with a definite boundary which will be crossed by transactions in its implementation. An abstract object may be represented by a co-operating group of less-abstract objects, without requiring a distinguished element to be the ‘head’.

Objects and transactions apply at all levels.

These concepts of objects and transactions are applied to entire systems, and to any components in the design down to the level of programming-language “classes”. We use a refinement process through which abstract objects and transactions can be transformed into classes and methods of a programming language.

3.2 Specifications

3.2.1 Object Specification

A types specifies the behavior of a set of objects.

Objects are characterized by their types. A type T is a specification of externally visible behavior of a set of objects. Type T constrains the possible histories of transactions these objects – *members* of T – can participate in. We write $jo:Person$ to state that object named jo is a member of type $Person$, and conforms to the specification of $Person$. An object is a member of as many types as it conforms to; but membership of a type is for the lifetime of that object¹. A type does not make any statements about the implementation of an object. Thus, a type is specified by a set of transactions in which the object can participate.

Subtypes are subsets of objects.

A subtype is a subset of objects. We write $Woman \subseteq Person$ to mean that $\forall x \bullet x:Woman \Rightarrow x:Person$. We draw $\boxed{S} \dashv \triangleright \boxed{T}$ to indicate that S is defined so that it is a subtype of T . If any transaction is specified with the supertype as a participant, then the guarantees of that transaction will hold for any subtype as well. Subtyping makes absolutely no statement about inheritance of implementation.

3.2.2 Transaction Specifications need Models

Preconditions and postconditions specify transactions.

Each transaction is specified with at least one precondition/postcondition pair. The postcondition is a relation between the states of the transaction participants immediately before and after any occurrence of the transaction. The precondition states under what circumstances this postcondition may be relied on.

A rigorous specification needs a model.

Figure 3 includes an informal specification of the transaction sale. The semi-formal style is readable but potentially ambiguous. In general, we encourage complementing informal descriptions with formal specification, so that any informal clause may be rendered formal if required. In order to do this we need a precise vocabulary, called a specification model, in which to describe the effects on the participants.

OMT/Fusion: interpreting models as queries

We approach modeling in a different way from OMT/Fusion. Our primary interest is in describing *behavior*. In order to describe the behavior of some components, we need to understand the effect of that behavior on those components and on their subsequent behaviors. We need at least a hypothetical conceptual model of the components in order to do this. These models are sets of typed *queries*, which we can depict visually as types, associations, and attributes. We prefer to base these models directly on domain concepts. We then specify behaviors rigorously in terms of the model.

1. We use the notions of state-types and roles to describe more dynamic aspects of an object.

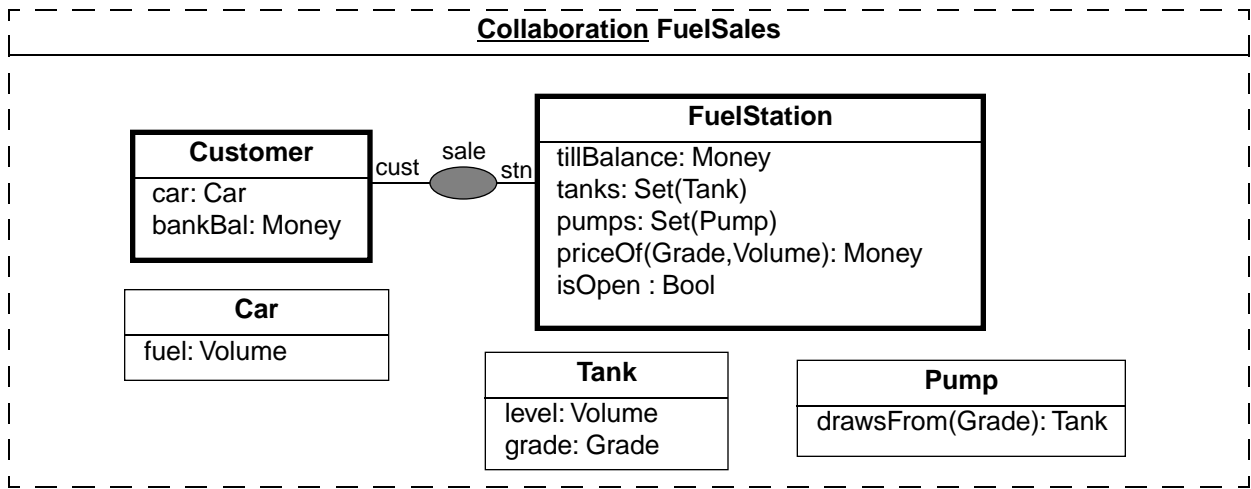


Figure 5 Type Model for a Collaboration

Contrary to OMT, attributes in our specification models do *not* represent stored data or ‘instance variables’.

3.2.3 Specification Models

Models are sets of queries

Since transactions are specified in terms of changes to objects’ states, we need precise descriptions of the states. An object’s state is modelled by a set of hypothetical read-only queries or “attributes”. The implication of “modelled” is that the queries are not necessarily directly implemented in the final design (although they should certainly be computable in any correct design) – their only purpose is to help describe how the transactions affect the object’s state, and hence how transactions interact across time.

Models define precise mutual vocabularies.

Nothing in a model is directly callable by any client code. Thus, a model does not violate encapsulation. In fact, it helps provide a clear shared understanding of behaviors *without* exposing implementation. Figure 5 shows one type model for specifying the transaction sale. We informally read this model as follows:

“We can describe a sale between a customer and a fuelStation if we have a notion of the bank-balance and car of a customer, and the notion of fuel in that car, and notions of the tillBalance in the station, and the priceOf some volume of each grade of fuel at that station, and the tank being drawn from by a pump, ...”

Queries can themselves yield interesting types.

A type-box such as Customer shows the signatures of queries in its model. It may additionally show the transactions which it takes part in. The results of queries are themselves of other types, and may be shown in their own boxes — e.g. Car & Tank.

3.2.4 Formal Transaction Specification

Transaction-specs use predicate logic on queries.

Using these model queries, transactions can be specified much more precisely. We usually complement informal explanations with formal precision:

Sale to cust from stn of v units of fuel of grade g.

transaction sale (cust:Customer, stn:FuelStation, g: Grade, vol: Volume)

pre The results apply only if the FuelStation is open for business.
stn.isOpen

post

If a refers to the FuelStation's current price for volume vol of grade g,

$\exists a:\text{Money} \cdot a == \text{stn.priceOf}(g, \text{vol})$

vol of grade g has been transferred from a tank to the customer's car

$\text{cust.car.fuel} == \overline{\text{cust.car.fuel}} + \text{vol} \wedge \text{stn.tanks}[\text{grade}==g].\text{level} \text{ -- } \text{vol}$

and amount a has been transferred from the customer's pocket to stn's till.

$\wedge \text{cust.bankBal} == \overline{\text{cust.bankBal}} - a \wedge \text{stn.tillBalance} \text{ += } a$

Postconditions relate before and after states.

The bars in a postcondition refer to the results of the queries in the objects' prior states. The postcondition is effectively a test that must be met by the designer; it does not imply any order of execution. The '=' signs are not assignments, but are equality constraints requiring object identity. $x+=y$ is an abbreviation for $x == \overline{x} + y$, and does not represent an imperative assignment (so $\text{stn.tillBalance} == \overline{\text{stn.tillBalance}} + a$), and ! is an abbreviation for "not".

If s is a set, $s[p]$ is the subset (or the only member) for which predicate p is true. Thus $\text{stn.tanks}[\text{grade}==g]$ is the tank whose grade is g. '∃' (≡ 'there is some...') declares local variables which may take a single value in any occurrence of the transaction, and their values are constrained by the conditions specified. The parameters of a transaction-spec are temporary names for the transaction's participants. Any of them may be affected by the transaction. Any parameters may be marked as **out** parameters.

Specification vs. design types

Some of the types in Figure 5 are shown in lightly-outlined boxes because they are *specification types*: that is, the only reason we have them (at this stage) is to understand the result of a query. Thus, the above model only mandates the existence of objects of type FuelStation and Customer. Car and Tank are artifacts used for convenient specification, and are not required to actually exist in an implementation. The former two types are *design types*: we know that objects of those types definitely exist, because they take part in transactions. In contrast, specification types have no assigned operational responsibilities at this level, and no transactions. Their sole purpose is to help understand the specified behaviors of some design types.¹

Multiple pre/post pairs may be combined.

We may have several pre/post pairs for one transaction, often in separate parts of a document. This enables different clients to express their individual views of some transaction, in terms of a model which is suitable to their view. It also allows a subtype to strengthen the specification provided in a supertype, for multiple super-types to specify the same transaction (the designer has to create an implementation which satisfies them simultaneously), and for an elegant description of many "exceptional" situations. Specialization of transactions (such as authorize in Figure 4) uses the same idea to extend the existing specification of authorize with further guarantees. The composition of two specs of the same transaction can be thought of as another spec which guarantees each post-condition under the respective pre-condition.

transaction trans **pre** P1 **post** R1

transaction trans **pre** P2 **post** R2

transaction trans **pre** P1 ∨ P2 **post** ($\overline{P1} \Rightarrow R1$) ∧ ($\overline{P2} \Rightarrow R2$)

1. There is a similar distinction in Syntropy, though Syntropy uses operations in specification types [Coo94]. We have a different semantics for operations on specification types [Cat95b].

In this case, post-condition R1 is guaranteed if $\overline{P1}$ was true; R2 is guaranteed if $\overline{P2}$ was true. We exploit multiple pre/post pairs actively to deal with exceptions and variations in behavior through subtypes. For example, if some customers were regular visitors, and they received a free drink each time they purchased fuel, we simply add another incremental spec for subtype VIPCustomers, implicitly satisfying the original sale:

```

transaction    sale (cust:VIPCustomer, stn:FuelStation, g: Grade, vol: Volume)
post           a free drink has been given by stn to cust ...

```

OMT/Fusion: Transactions

OMT uses data-flow diagrams to express the functionality of the system. These data-flow diagrams have been extended with some minimal object-notations, but they are not well integrated with the other models and are very rarely used in practice.

In Fusion, an operation schema describes the pre/postconditions of an operation. The method permits more than one schema for each system operation. However, the implied behavior with more than one schema is unclear, specially with the use of an explicit ‘changes’ clause for *framing* i.e. describing which parts of the model will and will not be modified by a transaction. Fusion examples seem to use an ‘if-else’ structure to describe post-conditions, and this tends to force consideration of many alternate paths at the same time.

3.2.5 Two Kinds of Transactions

There are two main kinds of transactions.

One important feature of a transaction is its locality – i.e. which object(s) are responsible for which parts of it. Our method uses two broad kinds of transaction: joint and localized.

Joint transactions specify combined effects.

A joint transaction describes an interaction which happens between one or more parties in a symmetric style: it contains no information as to who does what¹. During the development of a system, this provides a way to state the effects of a collaboration before assigning responsibilities and designing the supporting interactions. Figure 5 shows the sale transaction as a symmetric transaction. Note that this transaction, being non-localized, would admit an implementation in which the FuelStation sought out unwilling customers and relieved them of some money, so long as it dispensed the appropriate amount of fuel into their cars!

Localized transactions represent services of an object.

A localized transaction is provided by one object as a service to others. A separate design decision can localize a joint transaction by assigning responsibilities to objects, and defining a corresponding collaboration to achieve the combined effects. Thus, we may decide that a customer must request fuel from a FuelStation, or that an Attendant must dispense the fuel. A localized transaction is always qualified with the type of object which executes it – e.g. FuelStation :: sell_fuel.

Both transactions may be refined temporally.

In both styles, we may specify a transaction before refining it in terms of finer-grained objects and/or transactions, even further separating *what* from *who* from *how*.

3.2.6 Associations

Model queries can be depicted as associations.

When the result of a query is itself of some type with its own interesting queries, it is convenient to show this query as an association in a diagram. In Figure 5 below, we have shown all queries from Figure 5 in this style, yielding a very familiar looking diagram. In practice, certain queries are best depicted as attributes.

1. In many situations, being able to elide details of initiator and “receiver” improves abstraction.

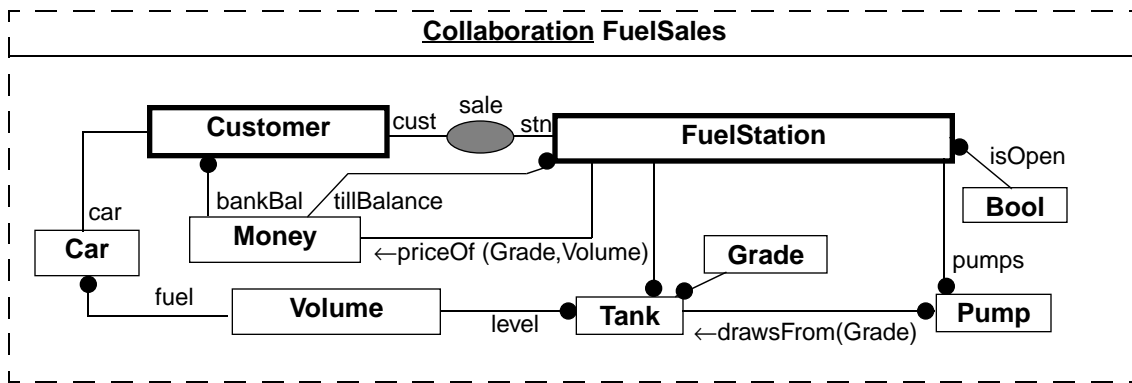


Figure 6 Type Model with all queries shown visually as associations

Associations can be “navigated” by name.

The label is written near the *destination* end of the association, as read. We use simple default naming rules for unlabeled associations, using either destination type names, or prefixing a “~” to traverse an association in the opposite direction.

$$\begin{aligned}
 & (\forall \text{cust:Customer} \cdot \text{cust.car:Car} \wedge \text{cust.car}.\sim\text{car} == \text{cust}) \\
 & \wedge (\forall c:\text{Car} \cdot c.\sim\text{car:Customer} \wedge c.\sim\text{car}.\text{car} == c)
 \end{aligned}$$

The semantics are the same as for model queries.

For every object `cust:Customer`, an object `cust.car` could be found or created, which is a member of type `Car`, and for which we can identify `cust` uniquely using an attribute called `~car`. Remember that we are defining an abstract vocabulary here: there is no implication that there is any stored data, or even any function, named `car` or `~car` in any implementation. In this expression, `a==b` means that the two expressions refer to the same object — that is, their identities are equal, or behaviorally indistinguishable.

Queries can return sets and NIL.

An association of multiple cardinality \bullet implies that the result of the corresponding query is a ‘flat’ set¹. An optional link \circ means the query may have a value of NIL. A query definition may be prefixed with [precondition], which limits the conditions under which that query is defined.

Queries can be parameterized.

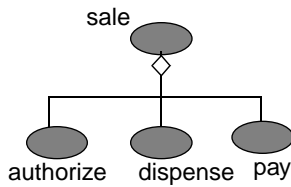
Even parameterized queries can be shown visually — for example `FuelStation::priceOf(Grade, Volume)` with result type `Money`. This usage stretches the notation somewhat since the parameter lists are generally different in each direction, hence we annotate the query with an arrow to indicate its direction. Thus, a pump draws from several tanks; however, for any specific grade a pump draws from a single tank.

Implementations are encapsulated; specification queries do not have to be.

Specification queries need not be encapsulated: a pre/postcondition may refer to any query of any type to which it can navigate. In fact, from a specification perspective, it is usually best to describe the logical net-effect desired as clearly as possible. One major purpose of encapsulation is to make implementation independent of specification, and queries are not necessarily implemented. However, the length of a specification expression like `customer.car.manufacturer.president.spouse.hair_colour` would suggest that the specification should be re-structured. Well structured specifications will localize queries within model types.

1. Flat sets are a very convenient variant of sets, which have the useful property that there can only be one level of containment: $\{1, \{2,3\}_F, 4\}_F == \{1,2,3,4\}_F$; navigating two multiple connections, such as `fuelStn.tanks.pumps`, gives a flat set rather than a set of sets. We also coerce freely between $\{x\}_F$ and x , and distribute queries over flat set members: $\{x,y\}_F.q == \{x.q,y.q\}_F$. In a postcondition, if s is a (flat) set, $s+=x$ means $s == s>>x$.

3.3 Refining Collaborations



Refining a transaction induces a refinement of the model.

In this case, we refine the model with `SaleRecord`.

A collaboration is a set of transactions which have some common purpose, and a common level of abstraction or detail. The transactions in a collaboration may refine those of a more abstract transaction. For example, consider refining our models to permit pipelining the usage of a pump, so that a second customer can begin using the pump before the first customer has completed payment for his fuel. In this refinement we distinguish three steps: enabling the pump (authorize), dispensing the fuel (dispense), and paying the amount due (pay).

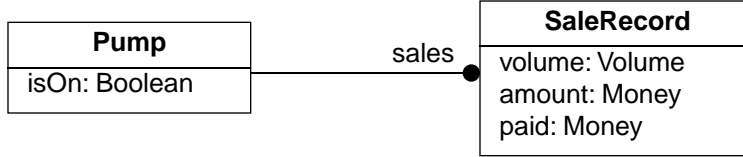
All transactions of a collaboration use the same models of their participants in their specification, so that the effect of each transaction on its successors is clear. The transactions of the collaboration operate at a finer level of detail than the transactions they refine, so more detail is needed in the model to be able to specify them; in particular, we need to add some attributes to `Pump`.

As shown in Figure 7, in order to describe this interaction granularity, our model must also include some notion of a `SaleRecord` which exists (at least) until a transaction has been paid for. The composition of authorize, dispense, and pay transactions is a refinement of sale. Because the heading states that this collaboration refines the more abstract version, we need not restate the previously defined attributes of `Pump`, nor the other parts of the model.

`Pump::isOn` represents whether the pump's motor is running. `SaleRecord` abstracts the idea that we want to remember the details of the sales that are happening or have happened. This keeps details of the sale apart from the pump itself, allowing a new sale to start before the previous one is paid for. It also leaves an audit trail for historical or reporting purposes.

Collaboration FuelSalesSequence refines FuelSales

-- "refines FuelSales", hence the model for FuelSales in Figure 5 is implicitly used



Authorization to use a particular pump, which will start the pump's motor

transaction authorize (stn: FuelStation, c: Customer, pump: Pump)

pre This spec is applicable if the pump is not in use and is one of ours.
 $!pump.isOn \wedge stn.isOpen \wedge pump \in stn.pumps$

post $pump.isOn$ -- The pump's motor is running.

The dispensing of v units of fuel of grade g from Pump p.

transaction dispense (stn: FuelStation, cust: Customer, pump: Pump, vol: Volume, g: Grade)

pre This spec is applicable if the pump is running, and grade g is available for selection at pump.
 $pump.isOn \wedge g \in pump.drawsFrom.grade$

post Vol units transferred from tank to car, pump motor stopped; the history of Sales at this pump now includes a new record with the correct vol, grade, and outstanding balance.
 $pump.drawsFrom(g).level \text{--} vol \wedge cust.car.fuel \text{+} vol \wedge !pump.isOn$
 $\wedge pump.sales \text{+} SaleRecord.new[volume==vol \wedge paid==0 \wedge amount==stn.priceOf(g,vol)]$

Payment for fuel; customer identifies pump to be paid for.

transaction pay (cust: Customer, stn: FuelStation, amt: Money, forPump: Pump)

pre Only applies if there is an unpaid sale for this pump.
 $\exists s: SaleRecord \cdot s \in forPump.sales \wedge s.paid < s.amount$

post Money transferred and applied against an appropriate Sale Record. Note that this does not specify which SaleRecord has been paid.
 $s.paid \text{+} amt \wedge stn.tillbalance \text{+} amt \wedge cust.bankBal \text{--} amt$

Transaction refinement

A sequence of authorization to use a pump in a Fuel Station, dispensing of fuel from the pump, and payment of the station's price for that volume of that grade of fuel, constitutes a sale.

$\forall stn: FuelStation, c: Customer, v: Volume, g: Grade, p \in stn.pumps, a == stn.priceOf(v, g) \bullet$
 $\langle authorize(stn, c, p) ; dispense(stn, c, p, v, g) ; pay(c, stn, a, p) \rangle \Rightarrow sale(stn, c, g, v)$

Figure 7 Transaction refinement: $\langle authorize ; dispense ; pay \rangle \Rightarrow sale$

New objects may exist in postconditions.

In the postcondition of dispense, we use the construction `Type.new[predicate]`, which asserts the existence of an object which did not exist before the transaction occurred, and for which the predicate is true.

A collaboration may refine a transaction.

In the bottom section of Figure 7, we make an important refinement claim and establish a strong traceability link:

$\langle authorize ; dispense ; pay \rangle \Rightarrow sale.$

A collaboration constrains its transaction sequences.

Although this looks like a sequential program, it is really just a temporal constraint¹. It asserts that if a "compatible" sequence is followed for any (' \forall ') combination of fuel station, customer, pump, fuel-grade, volume, and amount of money, and provided

1. An extension of the Fusion lifecycle expression (Col93). We express certain temporal constraints using such extended regular-expressions.

the amount is the appropriate price, then that sequence constitutes what we previously specified as a sale. It is straightforward to simplify this syntax.

This refinement is traceable and can be checked for correctness.

Careful inspection shows that the various parts of sale's specification have been distributed among the three finer transactions, strongly suggesting that the assertion is correct. A completely formal proof would require us to verify a number of other things. For example, we normally use the informal assumption that a correct implementation is one which does not make changes beyond those required to fulfill the relevant postcondition(s), without using 'framing' clauses to explicitly limit which parts of the model may be modified by a transaction. However, in certain critical applications we recommend explicit framing clauses.

Refinement is an essential part of CATALYSIS.

CATALYSIS treats refinement as an essential part of development. In fact, the temporal refinement described in this example is just one of several useful refinement steps we support, improving traceability. In general, any transaction may be refined in this manner. The model – joint or localized – is refined as well, and a sequence expression imposes a temporal constraint on the collaboration.

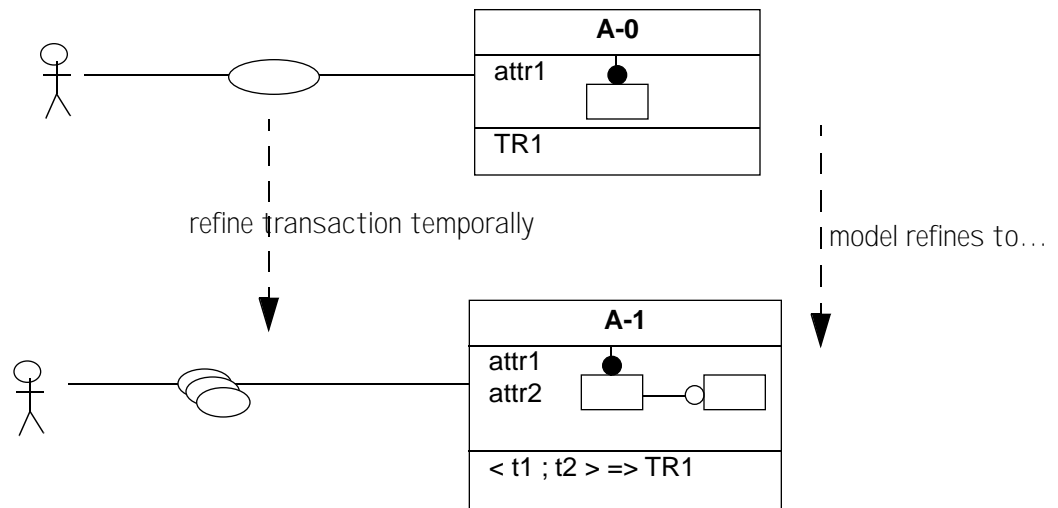


Figure 8 Refined interactions induce refined models

OMT/Fusion: Refinement

OMT does not utilize any notion of refinement of models, and the models are not clearly derived from a particular level of granularity. Fusion does not use a strong notion of refinement. A Fusion operation schema describes system behavior at a single level of granularity. While lower levels of granularity are sometimes best deferred, there are often times when we must describe the system at more than one level. However, Fusion does not encourage the analyst to describe such refinements, on the grounds that they are implementation dependent.

Fusion and framing

Fusion uses an explicit form of "framing". An operation schema has a **changes** clause, which lists those parts of the model which may be modified by the operation. If a schema has its preconditions satisfied, then the only part of the system which will change is that identified in its **changes** clause; all others are guaranteed unchanged.

3.4 Roles and Collaboration Patterns

An object can play many roles; each collaboration may need a different model.

The places for participants in a collaboration are called its *roles*. One object may play several roles, in one or more collaborations. Each collaboration in which an object takes part may be described using a different model. These different models represent

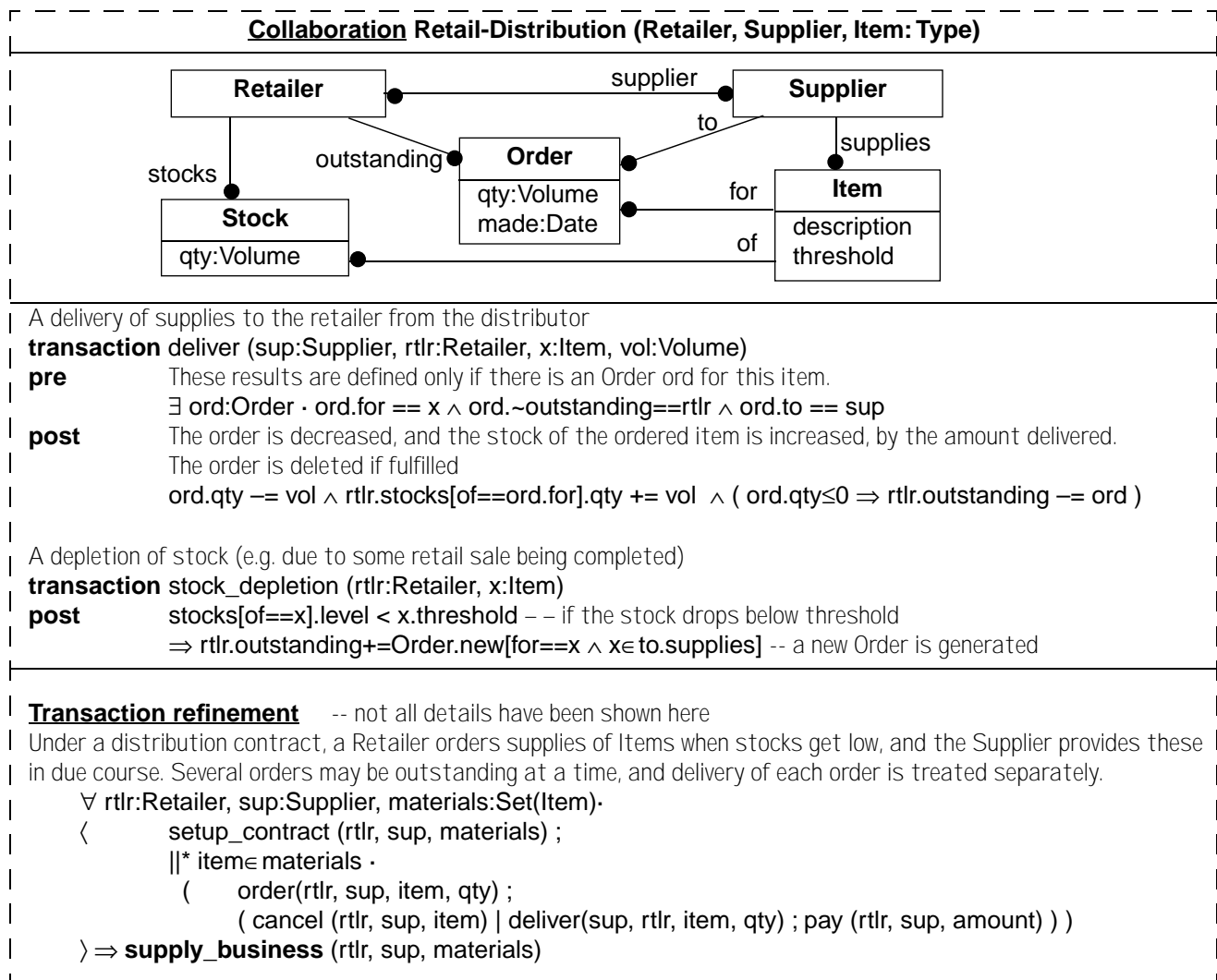


Figure 9 A Different View: The Retailer-Supplier Collaboration

different ‘views’ of the object. Each model represents a mutually agreed conceptual view of transactions by the collaborators.

e.g. Any retailer plays a certain roles with respect to its suppliers.

For example, we have described the FuelStation’s dealings with its customers separately from its dealings with suppliers. Moreover, the retailer-distributor collaboration may be expressed in a generic way, independent of the precise kind of item being exchanged. We build a separate model to show replenishment of stocks by suppliers (Figure 9). This model is *parameterized* by the actual type of the Retailer, Supplier, and Item, and can be re-used in a variety of different modeling contexts by instantiation with appropriate types for the parameters.

This view has detailed two transactions: deliver and stock_depletion. The latter specifies that an order is placed whenever inventories drop below a certain threshold. It appears as a ‘spontaneous’ transaction in this view, even though it is explicitly initiated by a customer in Figure 7. The model shown is not complete, and actually describes a more complex composition of transactions including cancellation and delivery, which refines a very high-level transaction for the supply-business itself.

Temporal constraints are complex, and do not impose concrete sequences.

We now need complex sequence constraints — for example, in the bottom section of Figure 9. f;g within a sequencing constraint $\langle \dots \rangle$ means that f must be *eventually* followed by g; any intervening transactions are not permitted to change the result of g. $\|$ * var:Type \cdot f(var) means any number of interleaved (concurrent) occurrences of f with

different values of var. It is used in $\|* \text{item} \in \text{materials}$ to indicate concurrent lifecycles for different items being re-stocked. $\langle f | g \rangle$ means that either f or g occurs.

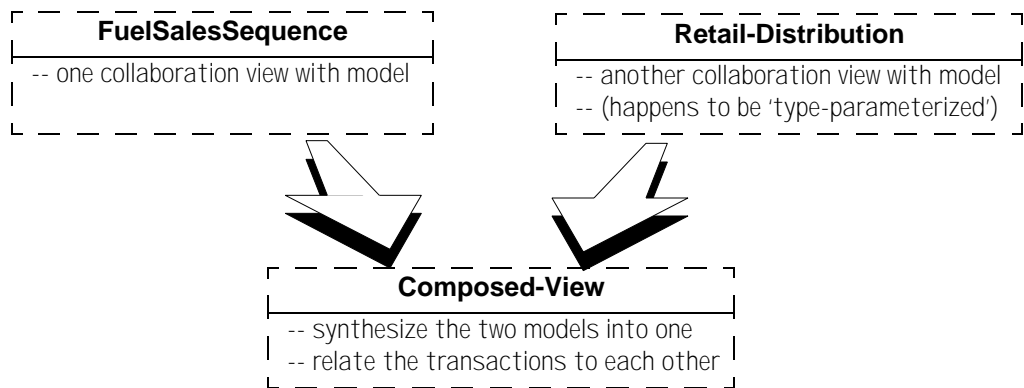
OMT/Fusion: Lifecycle model

OMT does not have an explicit notion of lifecycle models of transactions. It does utilize state-charts to express sequencing behaviors, but these state-charts are built on a per-class basis. Fusion's sequence expressions are somewhat limited in the presence of certain common kinds of multiplicities and interleavings. For example, it would be very difficult to describe the fact that multiple cancellation/delivery/payment cycles could be interleaved, but not for the same order. The extended lifecycle model of CATALYSIS alleviates many of these limitations.

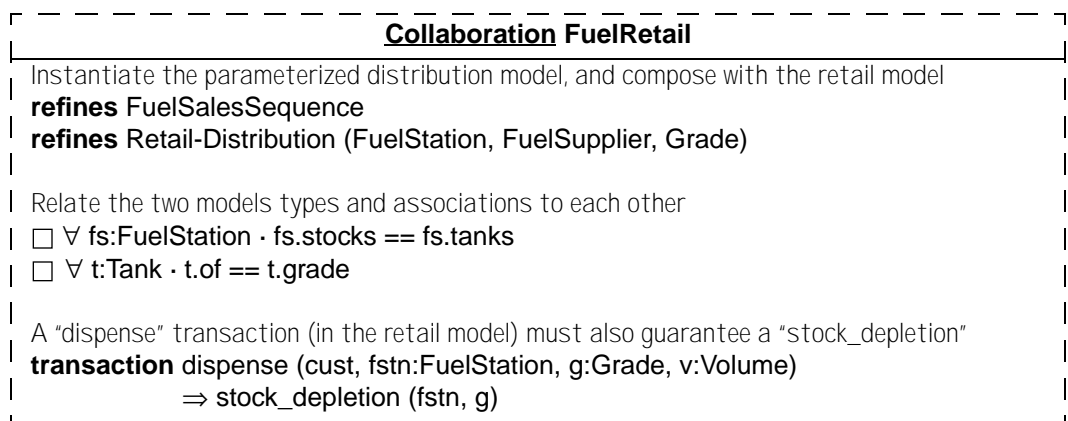
3.5 Composing Collaboration Patterns

We can compose these two views.

Of course, we now must describe how these two views of a FuelStation are composed together. We instantiate the type parameterized collaboration by defining Items to be a



Grade, and Retailer to be a FuelStation. We compose the resulting supply model with our original model for a FuelSalesSequence to construct a joint collaboration with FuelStations playing roles of both consumer and supplier. The composition, defined below, can also be depicted visually as a combination of two views. The \square depicted is an “invariant” i.e. a condition which should never be violated.



The two models are combined and constrained.

This collaboration implicitly has all the definitions of its parents, subject to renamings, like Retailer to FuelStation. We relate the two models, identifying stocks with tanks.

The behaviors in both views must interact.

Most importantly, we need to specify that the stock_depletion transaction-spec is observed whenever a sale occurs, in particular, whenever dispense occurs. This is done

by asserting that dispense must observe all the pre/post-condition pairs of stock_depletion stated in Retail-Distribution, in addition to those already stated in the Fuel Sales Sequence collaboration.

This refinement retains all previous guarantees.

Notice that none of the previous specifications in the model or transactions are superseded: anyone who has only read the Fuel Sales Sequence collaboration will still be able to understand those aspects of the FuelStation's behavior with which it deals — although it may be necessary to also read the collaborations to which it refers. Refinement of collaborations retains previous guarantees, and is a powerful tool. The resulting joint model shown below could be generated automatically by a tool if necessary.

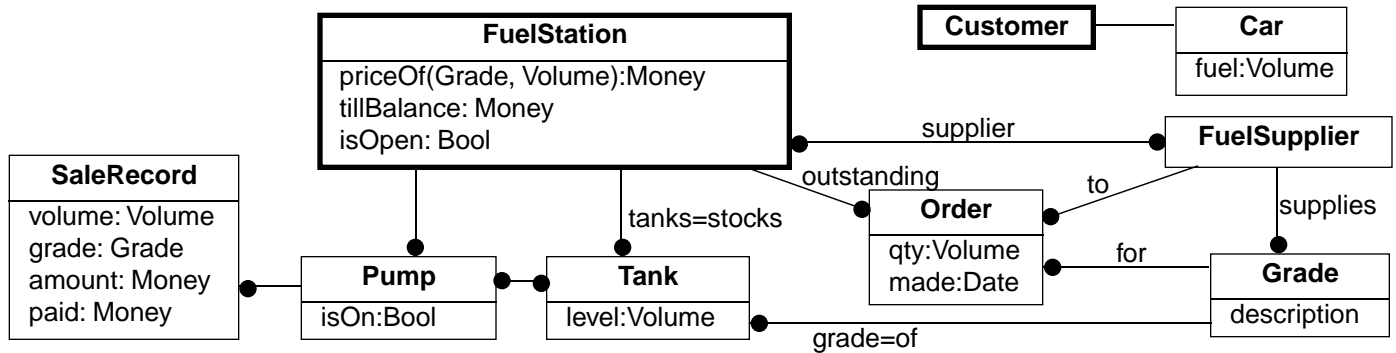


Figure 10 A Composite View and Model

We combined views and instantiated a parameterized model.

Roles are combined by composing collaborations — whose models describe interrelated types — and relating the types with further constraints¹. We have illustrated a composition in which a key step is the instantiation of a type-parameterized collaboration. A less powerful variation is to use un-parameterized collaborations, but create within the composition a type which is the common subtype of types selected from the different collaborations.²

Multiple views on an object are essential to support Design Patterns.

A component offers multiple interfaces, each supported by a different model if necessary. This approach is essential to support the integration of *design patterns* [Gam94] into a development method. Using this approach, which is based on objects playing multiple roles, designs are synthesized from known interaction patterns. This is also in line with trends in complex components, such as in Microsoft OLE [OLE93], OpenDoc [Ope95], and Java [Jav95], and is a significant extension to modeling capabilities.

OMT/Fusion: and multiple views.

Since CATALYSIS is based on describing collaborative behaviors, and using those behaviors to induce supporting models, it is quite natural for us to utilize multiple views of a component. OMT and Fusion, like most existing methodologies, do not support this approach.

1. Loosely speaking, this is the specification equivalent of a “framework”.

2. We are developing a richer model of role-playing in which many instances of the same type of role can be played by one object — e.g., one person can be an employee of several companies; and in which an object can dynamically change the roles it plays.

3.6 Specifications, Collaborations, Roles and Design

A quick review of the constructs used so far.

So far we have used the domain — the real world outside any computer system — to illustrate the modeling principles involved, including:

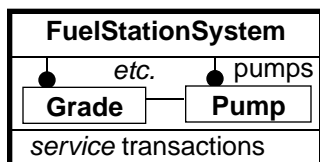
- specification of transactions using pre/postconditions and a type model;
- use of collaborations, which are sets of transactions between collaborators with a common model, to capture interactions between members of a set of types;
- refinement of collaborations to a finer level of temporal detail, in which one abstract transaction turns out to be composed of several smaller ones;
- definition of multiple views of a component, each with its own collaborations;
- composition of collaborations to make one object play several roles.

These principles can be applied inside a software system, as we show in the next several sections. They also apply down to code, as shown in Section 5.2.

4.0 Specifying Services — “Who”

We can effectively localize responsibilities after understanding joint behaviors.

Localizing behaviors defines “services”.



In order to define a component’s services, we need a model of it.

We distinguish system specification types from their domain counterparts.

4.1 Services

When the joint transactions of a component have been identified and specified, we can *localize* them: that is, decide which parts of the precondition and postcondition will be met by the component, and which by the other participants. In practice, this means refining joint transactions to localized *services*.

A service is a transaction for which the responsibility of achieving the required effect has been assigned to one participant (see Figure 1) — the “receiver” in standard terminology; another participant is the sender; and yet others become involved through services invoked in turn by the implementation in the receiver. Just as with a transaction, a service may also be refined to a sequence of transactions.

The name of a service is prefixed with the type of objects which provide it — e.g. `FuelStationSystem::selectGrade` might represent the selection of a grade at a pump, as supported by `FuelStationSystem`, the software which runs the fuel station itself. The services a type provides can be listed in its type box. Only design types represent components which provide services. Recall that design types are indicated by dark-bordered boxes. In this case, we have decided that the `FuelStationSystem` will be a separately implemented component in its domain.

The component will need a corresponding model. It is useful to show the entire model of a component, including associations, inside the model section of the type box: this emphasizes the distinction between types representing the real customers from the system’s internal model of those objects. This visual containment is also useful in representing certain invariants: navigating links which do not travel outside the type box always lead to the same containing object. Thus, a pump at a station can only be associated with a grade available at that station.

Notice that at this stage we are specifying the behavior of the component providing the services, `FuelStationSystem`, and not of any of the objects within its model. For example, although there may be a specification object representing a physical `Pump` within the system’s model, we do not assign the service of recording an event such as the replacement of the pump nozzle (`onHook`) to that object, but to the system as a

whole, with an identification of pump as a parameter. Moreover, as long as the pump parameter in this service remains a specification type, we even defer the means by which the identification of this pump is conveyed. The design of the internal structure of the system, and assignment of responsibilities to its components, are design steps, although they may be taken early in some situations.

This model is drawn from the domain model.

This model represents the view this component needs of its domain concepts in order to adequately specify its services. It is based, as far as possible, on the domain model. In determining the complete list of transactions in which the system takes part, we must take into account those from all the roles it plays. In this example, FuelStation-System has a model which is based on the domain model we have evolved so far.

4.2 Snapshots

Snapshots depict configurations of objects.

A snapshot is a picture of linked objects at a given instant, with all links conforming to the associations of a given model. It is a useful tool for discussion, especially to understand the implications of a transaction-spec. Often the best way of formalizing a spec is to draw a two-color before-and-after pair of snapshots. The example shown in Figure 11 is based on the type model in Figure 13.

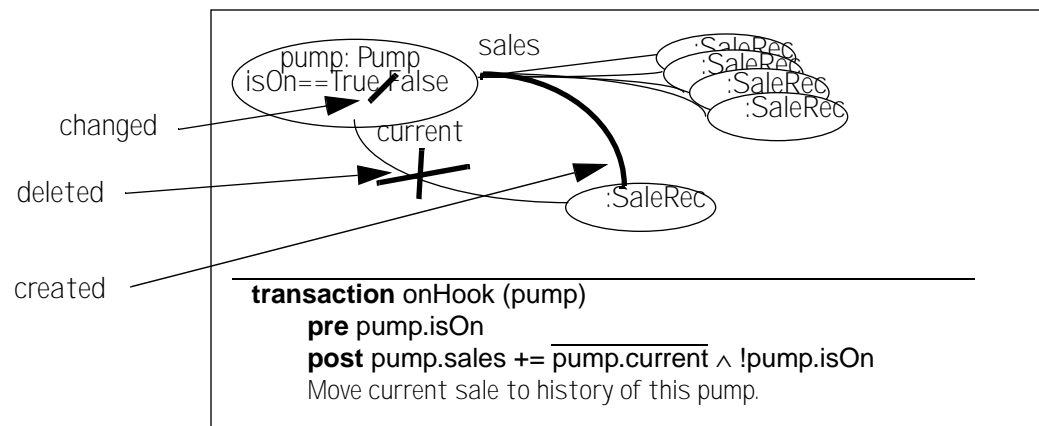


Figure 11 Snapshots: a thinking tool

Scenarios, with accompanying snapshots, depict sequences of transaction occurrences.

A *scenario* is a particular prototypical sequence of transaction occurrences. An *interaction diagram* (Figure 12) illustrates a scenario, and is often accompanied by a series of superposed snapshots (resembling a film-strip) illustrating the evolution of a system's state through the scenario. Each role in a collaboration is shown as a vertical bar, and transactions are shown in sequence down the page. Flow of control follows a path vertically down the page, as each transaction performs its tasks. The initiator of each transaction is indicated by a darkened oval. The corresponding snapshots express how the instance configuration evolves.

All the models can be derived from scenarios and snapshots.

As Figure 12 illustrates, the type model abstracts all the snapshots, and each snapshot is an occurrence of the type model. A transaction specification abstracts all the pre/post pairs of snapshots for any of its transaction occurrences in any scenario. A state diagram for any model type is a projection of the snapshot progression onto a single model type. Each state constrains some portion of the snapshots, and the transitions correspond to transaction occurrences. Thus, states of a pump might be on, off, and its state transitions correspond to the authorize and dispense transactions.

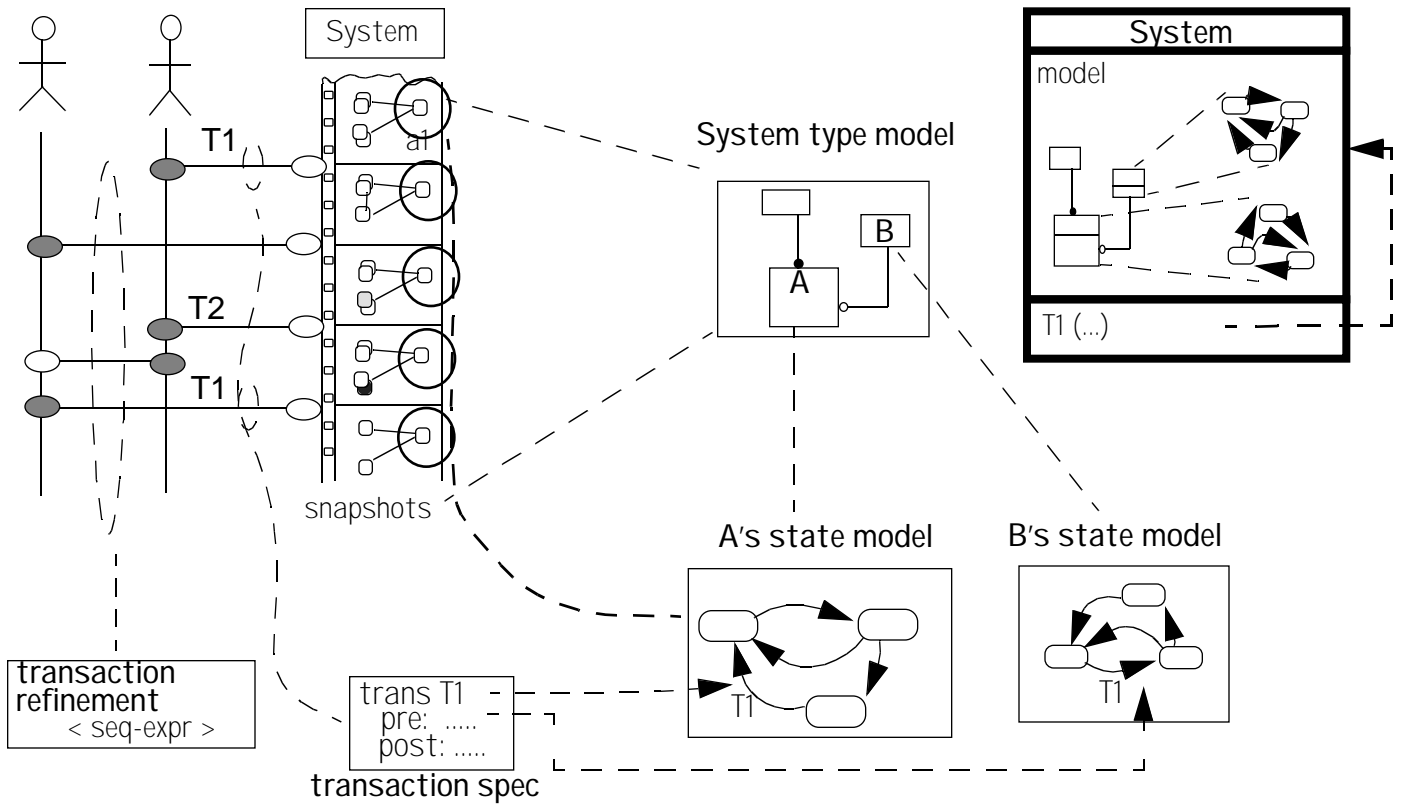


Figure 12 Scenarios and Models

Snapshots also help understand model consistency.

The snapshots also provide a vivid “film-strip” metaphor for the underlying consistency rules between the models [Dso94], depicting configurations of model objects before and after each transaction occurrence¹. At any level of refinement we can check that:

Consistency rules are simple and clear.

- the type model admits every snapshot required at the current level of refinement
- the states in the state model reflect pre/post conditions on the snapshots
- the states of a model type are defined in terms of the model queries
- the transitions on a state model are triggered by transactions on the system
- the transaction specification is expressed in terms of the type and state models
- every part in the type model is created and read by some transaction

CASE tools can readily support snapshots.

Some of the available CASE tools for CATALYSIS illustrate a sequence of snapshots as a film-strip, and check the links you draw against the type model. Of course, snapshots and interaction diagrams only depict a specific trace of interactions at a time, and cannot be considered to be complete descriptions. Transaction specifications and state models account for the general cases.

OMT/Fusion: scenarios, system interface, and models.

Fusion utilizes scenarios to define the system interface. CATALYSIS uses a similar approach, except that we can have mixed levels of description. The usage of snapshots as a metaphor to relate all the models is unique to CATALYSIS. Describing the state of a complex component in terms of state models of its model types, without resorting to the design of design internal messaging and intermediate computational

1. This “film-strip” view actually forms a strong basis even for how we teach Fusion and OMT. We have found that our strict interpretation of the snapshots to have a very positive effect on the clarity and value of the models produced, as it provides beginners with an intuitive tool for a rigorous approach.

states, has also proven very useful in CATALYSIS. OMT uses event-traces prior to defining the state-models, and does not relate them to the object or functional model. OMT has recently discussed use-cases for requirements description.

4.3 System Model

System models should follow the domain.

Specification types (depicted in diagrams with light borders) are used extensively in describing transactions. At the start of the development process, the only “real” objects we know about are the system we propose to develop (or component within a system), and the objects with which it interacts. These are ‘design objects’ at the business level, since they represent specific responsibility allocations made in that context. We will need some queries (specification attributes and associations) to define the transactions. Since queries are hypothetical and defer details of implementation, we prefer to relate them closely to the domain. Our initial specification of the behaviors required of a system, and the supporting model of the system itself, therefore draws strongly on a model of the concepts in the domain. Many system specifications may be based on one domain model.

Design models should ideally follow the domain, but will not always do so.

In the subsequent design phase, the network of transactions between collaborating objects may remain faithful to the layout of associations in the specification model; this provides for rapid response to requirements changes, since the classes in the implementation then correspond closely to the types identified in the domain. However, where performance or aggressive re-use of existing components is important, the model may go through a number of refinements and transformations first, re-arranging the model to be closer to a practical implementation, while still providing the required behavior.

Some development heuristics for system modeling.

We have found these steps useful in generating a specification model of a system, but we shall not illustrate all of them in detail in this paper:

- Analyze and model the domain, either informally or formally, to build a well-defined vocabulary and understand relevant domain transactions. In our example we have created a fairly rigorous model to illustrate the concepts.
- Write informal scenarios or storyboards in which the system is used, identifying the points at which the outside participants interact with the system. Interaction diagrams are often helpful at this stage. The system is treated as a single object, even if it will be eventually designed as a distributed system.
- Outline the overall responsibilities of the system in informal terms, and in terms of the collaborations in which it takes part.
- Describe the identified transactions informally at first, and then more formally as a system model is created. The system model draws on the domain model as far as possible. The transactions are with the system as a whole at this stage, and pre and postconditions are written in terms of the links and attributes of the model as a whole. Transactions may be refined temporally and specialized¹.
- Derive the system model. This model describes what the system needs to know about the world around it, in order to behave as required. It can be derived, beginning with a blank strip of snapshots, by adding just those types needed to help describe the system’s interactions with the objects external to it through the course of a scenario. When we need a type, we prefer one chosen from the domain model. Thus, the system model grows as a selection and extension of the domain model.

1. These refinements are a formalization of the notion of “Use-Case” in [Jac91]

- Exploit snapshots. Visually depicting the effects of a transaction on the model queries is very useful for understanding and exercising the model through a scenario, and for helping formalize the transaction specifications.
- Utilize state charts (only briefly illustrated in this paper, though described well elsewhere, for example in [Coo94]) to clarify the states of some model types within a complex component, and consequently for understanding the states of that complex component.

4.3.1 Inputs and outputs of the System

Services and their inputs have specified effects on the receiver's model.

When a joint transaction (specified as part of a collaboration, but without a specific receiver) is refined to a sequence of services, it is necessary to specify the effects of each service on its receiver. However, if the receiver is human or outside the scope of our modeling, we may omit it. To specify a system which interacts with human users, it is therefore usual just to concentrate on specifying the services of which the system is receiver — the stimuli from the outside world. Each incoming service transaction has a specified effect upon the model of that system.

“Out” parameters might effect the sender's model.

System outputs are dealt with in several ways. A service may have **out** parameters, which define results returned to the sender — implemented in a programming language by output parameters or by return-values with procedure-call semantics.

Outputs can be specified by the desired post-condition on external objects.

Services invoked *by* the system can be specified both in terms of an effect on the sender's model (the system model), as well as on the receiver's model. A state-change on the sender records the completion of some task, such as passing information to a user, should that knowledge be significant for subsequent behavior of the sender. For example, a postcondition of a `selectGrade` request may require that ‘the operator has been informed of the customer’, reflecting a state-change in the external object itself.

Outputs can also be specified by requiring output events to be generated.

There might also be a corresponding output invocation, called `notifyOperator`, invocable by the system upon the operator alarm, which might guarantee the same postcondition, and we might know the system will be wired to the operator's alarm. However, it would normally be left to the designer of `selectGrade` to choose to satisfy its specified postcondition by using the second operation. In other circumstances, such as in the context of certain “open-systems” [Lea95], we might prefer to require that specific services be invoked by the system, without requiring that specific effects be achieved.

Invariants can relate system model states to states of external objects.

The `FuelStationSystem`'s model of a `Pump` has an `isOn` status which represents whether the motor is on; transaction-specs for the system can refer to it without worrying about making the real motor and the attribute conform: that can be specified with an invariant which crosses the boundary of the system.

GUI presentations can be defined by such invariants.

The same applies to user displays of all kinds, which would initially be treated as a separate “user-interface” domain with its own set of models. We would use invariants to relate them to each other. For example, we might say:

$$\forall p:\text{Pump} \bullet p.\text{isOn} \Leftrightarrow p.\text{GUI_Icon}.\text{isHighlighted}$$

Implementing such invariants is an issue for architectural design.

The protocol and mechanics to control any external objects which are directly reflected in the system model is usually a design step. For example, we may make architectural decisions about the “connectors” between the system model and the ‘real’ domain objects, such as using *model-view-controller* to connect the objects in the system's type model to the user through a graphical user interface, and about the communication of object-ids across this interface by visual or software means.

4.3.2 Refinement of a joint-transaction to services

A transaction may be refined to a specific protocol of services.

We next refine the dispense joint-transaction between Customers and FuelStations, which accomplishes transfer of fuel from an active pump (Figure 7), to a service protocol of select a grade, squeeze the nozzle for some time, and hang-up the nozzle. In partitioning responsibilities, we decided that the customer must explicitly select a grade¹, and that a sequence of pulses will be sent to the system by some flow-metering device. We show both originator and receiver for each service interaction, using:

originator → receiver.service(parameters)

Either one may be omitted if irrelevant, as used below for the meterPulse service. Assertions which should be true at any point in the sequence are shown in brackets.

A temporal constraint specifies this protocol.

```
∀ stnsys:FuelStationSystem, pump:Pump, grade:Grade, vol:Volume ·
  < [pump.isOn]
    cust→stnsys.selectGrade (pump, grade); -- cust selects Grade
      →stnsys.meterPulse (pump)* ; -- measurement pulses sent
    cust→stnsys.onHook (pump) -- cust hangs-up
  [!pump.isOn]
  > ⇒ dispense (stnsys.stn, cust, pump, vol, grade)
```

We are bridging the gap from domain to system specification.

In refining the joint-transaction to this service protocol, we refer to several objects from the domain model. These include cust, pump, and grade. In addition, we have now committed to having the FuelStationSystem as a ‘real’ object in the domain, and relate it to the actual fuel station itself by “stnsys.stn”. As we will see, some of the domain objects will be reflected in the specification model of the new FuelStationSystem component, to unambiguously describe its services.

This induces a more detailed model of the system.

As usual, the refined sequence needs a more detailed model, shown in Figure 13. Since selectGrade is a separate transaction, where previously grade was simply an input parameter to the abstract dispense transaction, we need the notion of the selected grade for a SaleRecord. In this model, FuelStationSystem::authorize activates the pump, and FuelStationSystem::selectGrade creates the new SaleRecord with an amount due of zero, and records the selected grade. In order to specify the effect of meterPulse, we model SaleRecord tracking the volume of fuel based on meter pulses, and the amount due based on the selected grade and volume. The model allows us to specify both the current as well as previous sales on a pump. Some details are elided in the figure.

1. Once again, this constitutes an external “business design” decision. An intelligent system might have chosen a grade by recognizing the model of the customer’s vehicle.

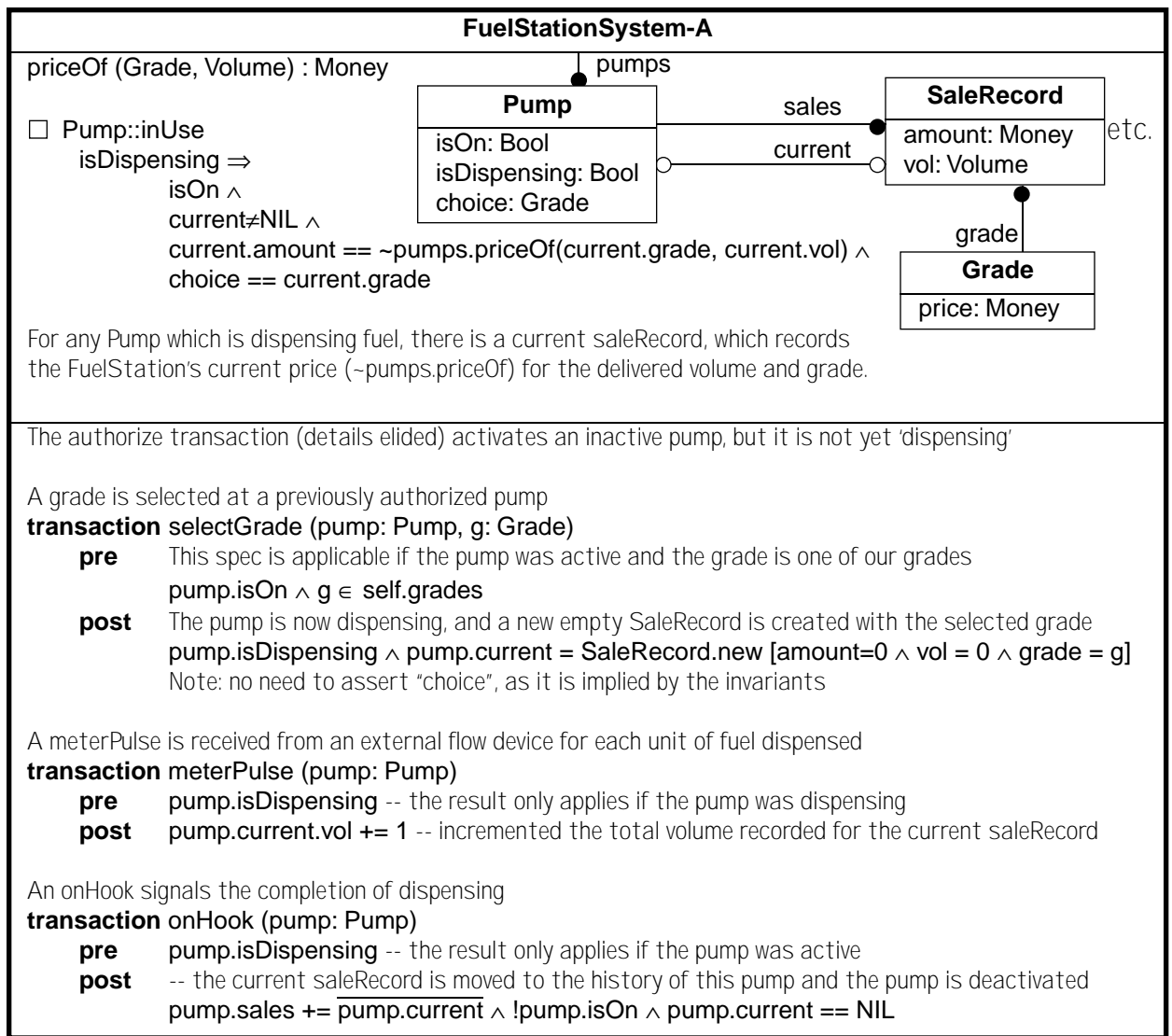


Figure 13 Refined and Localized Services

The refined model has its own invariants.

The model also includes an invariant indicated by □ Pump::inUse. It describes a boolean condition which must always be true when any pump is in use: it must have a current sale, and the amount due on that sale must reflect the volume pumped and the chosen grade price. Invariants may be “anchored” in this manner to any type in the model. Invariants may remain unnamed.

5.0 Design — “How”

5.1 Transformations in Design: Reification

Design might re-arrange the model.

During design we may have to re-arrange the information in the model so as to factor in implementation concerns e.g. distribution, efficiency, re-use of existing components. Such a step is called a reification, and it may involve the introduction of new types into the model, in anticipation of patterns of internal collaborations requiring new roles in the design. The transformation may be more striking if the implementation architecture is based on a different, highly parameterized view of the specification types. For example, a problem domain networks of objects may be mapped to a graph-based design structure with the corresponding graph algorithms.

We re-arrange this model looking ahead to de-coupling pumps from sales.

In the design we are envisaging, named FuelStationSystem-B, pumps will not need to know about sale records: they will maintain volume and amount information for the current sale while the pump is active, and will have this information transferred to a new SaleB object when the pump becomes inactive. We adapt our model towards this end by adding vol and amount attributes to PumpB¹. We also decide to keep details of filled but unpaid sales in one queue, and completed sales in an archive, and add two new associations for this. The re-arranged model, shown in Figure 14, is more suitable for distribution between separate processors in pumps and the main station since it only involves the transfer of values from the pump to the new SaleB upon onHook.

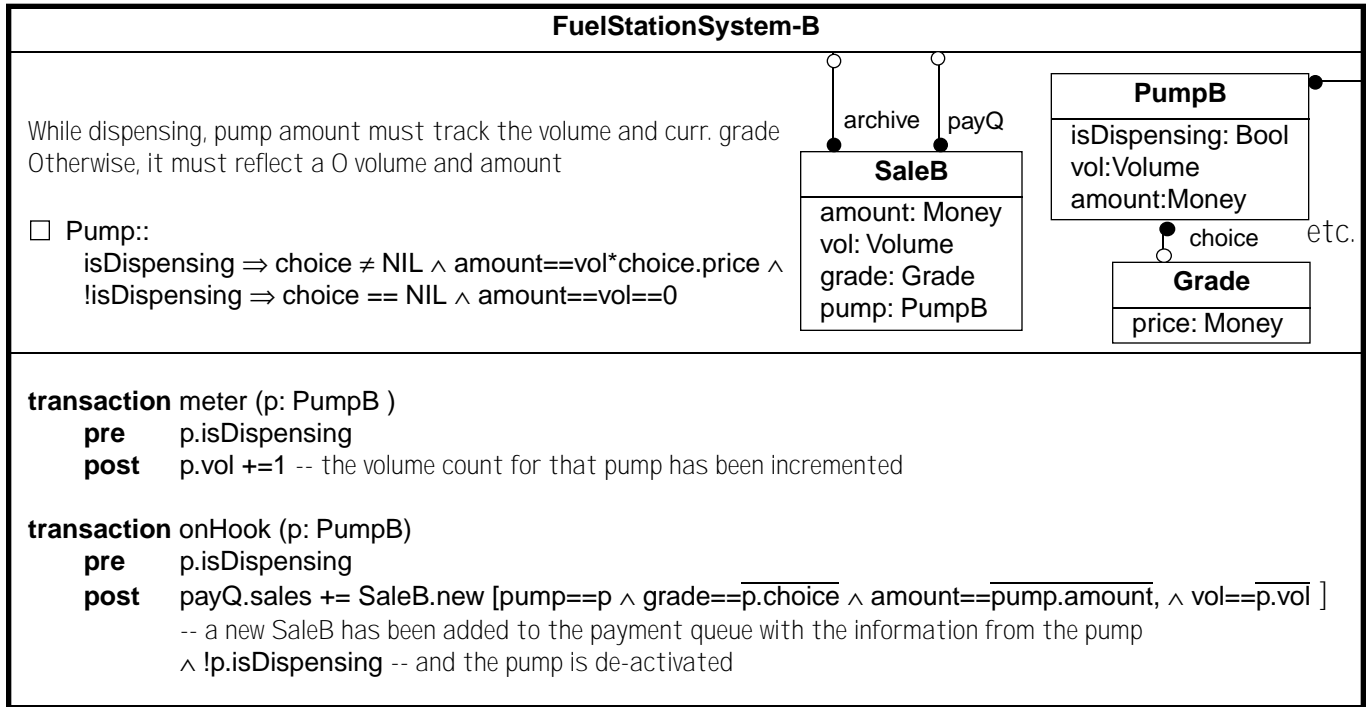


Figure 14 A re-arranged model more suitable for distribution across pumps

We could verify this re-arrangement formally.

How is this new model traceable to our previous models? How do we know it is a correct change? Strictly speaking, all the transactions on FuelStationSystem should now be re-specified for a FuelStationSystem-B. And (strictly), we should check that the transactions specify at least what they did in the more abstract vocabulary. Formally, we describe these guarantees by a “retrieval” or “abstraction” function. A retrieval defines how each query is defined in a design, ‘retrieving’ the information we expect from it. Since we claim FuelStationSystem-B ⊆ FuelStationSystem-A, we should be able to define the relationship between queries on the two models.

5.1.1 Traceability by “Retrieval”

We can show that the re-arrangement is valid and establish traceability.

Since every query on the abstract model FuelStationSystem-A should be traceable from the next design, FuelStationSystem-B, we can define the invariant relationships between their models. We can depict this on a combined diagram to visually depict

1. This makes no difference to the specification itself, as PumpB is still a “model type”, and not a true component to be implemented. However, we are thinking ahead to an implementation in which Pump will become an encapsulated and implemented “design” object.

the “retrieval”, as shown in Figure 15. The double-crossed lines relate the types from the two models, and provides traceability between the models.

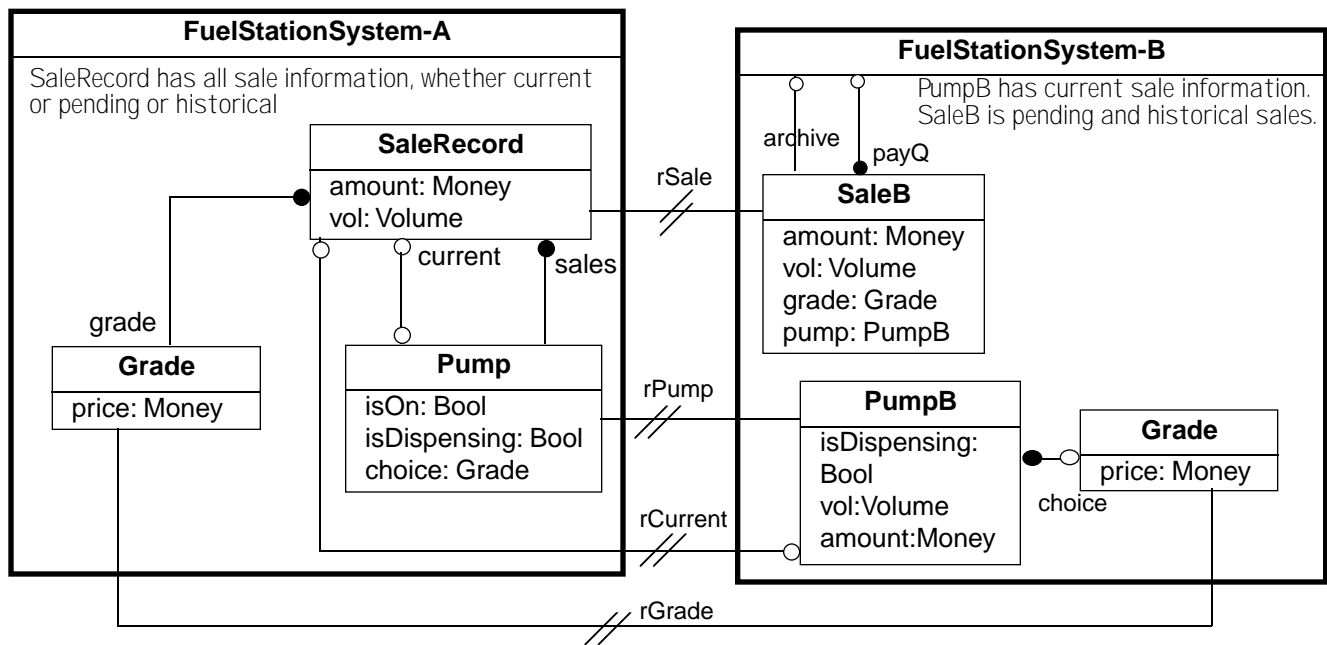


Figure 15 A composite model showing traceability

We can combine formal and informal descriptions of traceability.

The informal relationships are straightforward. *rSale* relates a *SaleB* to a *SaleRecord*, *rPump* relates a *PumpB* to a *Pump*, *rGrade* relates the two grades trivially, and *rCurrent* relates a *PumpB* to a *SaleRecord*, since a *PumpB* (with its *vol* and *amount* attributes), represents a *SaleRecord* until the dispensing is completed (hence the ‘optional’ association). These links are formally described below. Note the combination of informal with formal, and start by reading the informal:

PumpB:: -- A *PumpB* represents both *Pump* and its current *SaleRecord* (if any).
 -- either there is no ongoing sale, in which case there is no corresponding *SaleRecord*
 $rCurrent == NIL \wedge choice == NIL$
 -- or *PumpB* attributes match attributes of the current *SaleRecord* on its corresponding *Pump*
 $\vee rCurrent.grade == choice \wedge rCurrent.amount == amount \wedge rCurrent.vol == vol$
 $\wedge rCurrent.\sim current == rPump$

SaleB:: -- *SaleB* captures a completed *SaleRecord* in the abstract model
 -- the attributes must match up
 $rSale.amount == amount \wedge rSale.vol == vol \wedge rSale.grade == grade \wedge$
 -- and the pumps match up, for any completed *SaleRecord* ($\sim current == NIL$)
 $rSale.\sim sales == pump.rPump \wedge rSale.\sim current == NIL$

-- *SaleRecord* is represented either by *SaleB* or, if current, by *PumpB* attributes.

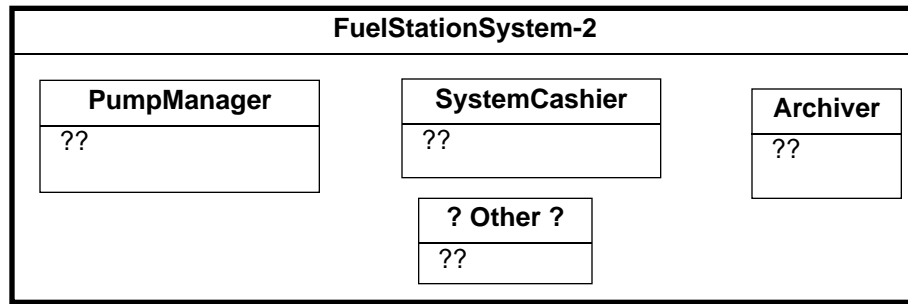
Sale:: $\sim rSale \neq NIL \vee \sim rCurrent \neq NIL \wedge current \neq NIL$

-- A *Pump*'s sales are represented by the current *Sale* which the *PumpB*
 -- represents, together with all the archive and *payQ* sales for this
 -- *FuelStation* which refer to (the *PumpB* equivalent of) this pump.

Pump::
 $sales == \{ \sim rPump.rCurrent \} \cup \sim pumps.payQ[pump == \sim rPump].rSale$
 $\cup \sim pumps.archive[pump == \sim rPump].rSale$

This approach supports a variety of architectural choices.

This approach to traceability works even if we chose a very different architectural partition in design. For example, we could have decided that the fuelstation would be itself built of three major components, one for managing the pumps themselves, another for the payment of sales, and a third for archiving historical information.



These three components would themselves have mutual interfaces and models, and the original specification model could be “retrieved” from this design as well.

The rigor is available for use when needed.

Note that formally establishing this traceability can be cumbersome and impractical without strong support from a CASE tool. However, like many aspects of CATALYSIS, the rigor is available for use when appropriate. The informal versions of the formal relationships are both applicable and very valuable in most situations. In some critical situations this degree of formal traceability might be justified.

5.2 Stepwise Refinement and Re-Localization to Implementation

We have discussed several refinement steps so far...

As our running example illustrates, we recommend the following approximate sequence of development steps:

- Build a collaboration model of the domain, with more than one view and several refinements. Understand the interactions between the “real components” in the business or problem domain. We used joint transactions, and did not happen to describe localized services in our specific example.
- Identify a tentative system boundary, and partition some responsibilities between the external actors and this software system. This is actually a design step, albeit at the level of domain or business design.
- Build a model of the system which permits description of the service transactions which are being localized to that system. Wherever appropriate, adequately describe the behaviors this system will provide in terms of this model, rather than prematurely design internal components and their interactions.
- Bias the model towards anticipated design choices, for example by localizing and partitioning parts of the model. Check that the model is still accurate.

And we can apply them recursively...

In the next few sections, we will have gone full circle and will be back to having a set of “real components”, except that they will now be inside our software system:

- Decompose the type model into internal “real components”. Partition the responsibilities between them. If the components are complex, build models of the components so the interfaces between them can be clearly specified, understood, and negotiated by all developers.

Of course, the pragmatics of a specific development effort will vary widely depending on the “true” (explicit and hidden) goals of that effort. For example, within the same modeling framework, we would follow a somewhat different progression for well

understood domains vs. projects which were truly novel in nature, and we would also approach things differently for projects which were building re-usable frameworks as their primary objective.

Even down to implementation...

The principles of stepwise refinement and localization can be applied further, and the same tools and techniques utilized down to the level of implementation code in an object-oriented programming language.

5.2.1 Localizing before temporal refinement

Development could take many paths. We re-trace our steps and try another path.

We will next illustrate briefly how localization of responsibilities and temporal refinement can be freely mixed and matched. In the previous section, we had localized behavior only to the level of the FuelStation, but had done temporal refinement down to the level of meter pulses. We could decide to stop at a coarser temporal refinement, then directly do some the design of some internal components by assigning behavior to them, and only then proceed with the temporal refinement.

We could have localized just very high-level services, like “dispense”.

For example, FuelStation must deal with the dispensing of a given volume of a given grade of fuel at a particular pump. Suppose we had this behavior localized only to the level of the FuelStation, and that our model was analogous to FuelStationSystem-A (Figure 13), with the addition of a cashLimit on dispensed fuel¹. We might specify dispense as:

```
transaction FuelStationSys:: dispense (vol: Volume, g: Grade, pump: Pump)
  pre    pump.isOn                                -- pump was authorized
  post   pump.current.amount < cashLimit (pump)  -- sale amount within cash limit
        ^ pump.current.amount == vol * g.price  -- correct price on the Sale
        ^ pump.dispensingDone                  -- pump changed state
```

We used a model of Pump with exclusive states isOn, and dispensingDone.

5.2.2 Localization within FuelStationSystem

This service could be localized on Pump.

We may decide to localize this behavior on the Pump itself, thereby introducing a *design object* for Pump, with specific responsibilities to be implemented within it.

```
transaction Pump:: dispense (vol: Volume, g: Grade )
  This "design component" Pump will be implemented with real responsibility for 'dispense'
  pre    self.isOn
  post   self.current.amount == vol*g.price
        ^ self.dispensingDone
        ^ self.amount < ~pumps.cashLimit (self)
```

Here, supporting services have been only partially localized to the pump.

We localized isOn, current, and dispensingDone. By localizing behavior to Pump, we have decided that it is now a “design type” i.e. a component to be implemented as such. It will have a definite implementation boundary (perhaps even as a single instance of a class), and it will control this transaction on behalf of the FuelSystem. Note that SaleRecord is still being used as a pure “specification” type, while Pump will now definitely be implemented to support this interface (or some refinement of it).

And other parts of the design localization have been deferred.

However, notice the use of ~pumps.cashLimit(self), instead of self.cashLimit. We have not decided whether or not the pump itself will have a notion of its cashLimit, so we merely specify in terms of “the station’s notion of the cashLimit for this pump”. The

1. We have added the detail of a cashLimit on the pumps to illustrate some interesting subsequent design issues. Dispensing cannot exceed the cashLimit for that pump.

cashLimit query is **not** localized. We might even localize cashLimit to some other object which checks the limit and stops the pump when needed, as we will proceed to do next.

5.2.3 Operations: The Last Transaction Refinement

By applying another refinement...

We next decide that dispense itself refines to a sequence of finer-grained operations, conforming to the following lifecycle expression:

```
Pump:: dispense (v: Volume, g: Grade)
  post    < offHook ; meter*; onHook >
  where v == the count of the number of occurrences of meter * some unit volume
```

And get the sequence <offHook;meter*;onHook> by a different development path.

In this refinement, offHook and onHook are the signals that the delivery nozzle has been removed and replaced, and meter signals the dispensing of some small unit of volume as measured by some physical flow-measurement device. Note that we have arrived at a similar internal design by a different refinement path. Both paths are traceable to the specification and back to the domain models.

In certain architectures, transactions which are refined may “disappear” in the implementation.

Whether dispense will now appear explicitly in our code or not depends on our architecture. If we are using co-routines or one process per object, dispense might well be the name of the sub-routine which listens for the events of the dispense lifecycle. However, if the transactions translate into function calls, then dispense is not explicitly visible: it is just a mode/protocol used for calling the meter and on-hook functions.

Similarly, specification types may also “disappear”.

In an analogous manner, specification types might also sometimes not be directly represented in an implementation, although we always prefer to retain a direct representation wherever possible. However, design types will always have a clear boundary in the implementation, even though it might not correspond to a “class”.

And “messages”, at last!

As finer transactions are specified, we eventually arrive at a level which will be directly implemented in a programming language. We change “transaction” to “operation” to indicate services which may be invoked by message-sends, and which will not undergo any further temporal refinement.

```
operation Pump::offHook (g:Grade)
  pre    -- pump had to be authorized
         isOn
  post    -- changed state, and current grade recorded
         isDispensing  $\wedge$  grade == g

operation Pump::meter
  pre    isDispensing -- pump had to be delivering
  post    current.vol += 1  $\wedge$  current.amount == current.vol * grade.price
         -- price and volume updated
          $\wedge$  (current.amount > cashLimit (self)) => isDone -- and cashLimit not exceeded

operation Pump::onHook
  pre    isDispensing
  post    isDone
```

We specify operations down to messages, and continue to use specification types.

We are using a *model* of Pump with exclusive states isOn, isDispensing, and isDone, for the Pump. Each meter pulse increments the volume and price during isDispensing. Note that up to this point we have not implemented these operations, but have simply specified them in terms of localized model queries. Also, while we do use the notion of current sale record, we have not yet decided whether or not SaleRecord will be a *design* object, implemented with its own responsibilities and services.

5.2.4 Implementing Operations

Objects and operations are finally implemented in an OO language.

Finally, we can actually **implement** our first few behaviors e.g. Pump::meter. Note that implementations appear syntactically different from specifications, and that we distinguish imperative assignments to variables from the equality-checks of our specifications. We reify the model type SaleRecord into an actual implemented class, and provide Pump with instance variables for vol, grade, and sale. Pump will track the current volume and grade, but the amount due will be tracked within SaleRecord:

```
class Pump {
  data: vol: Volume; grade: Grade; current: SaleRecord;
  services: meter();
}

Pump::meter () { vol := vol + 1; current.updateAndCheck (vol,grade); }
```

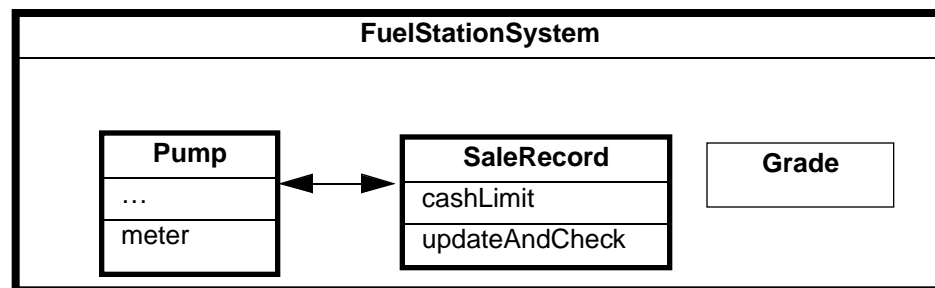
And these operations can also be *specified*.

This requires an operation on SaleRecord, which we may **specify** as:

```
SaleRecord :: updateAndCheck (vol: Volume, grade: Grade)
  post amount == vol * grade.price ^ (amount ≥ cashLimit(pump) => pump.isDone)
```

Until they are all implemented as methods.

We next localize cashLimit as an attribute on SaleRecord itself, presumably initialized by the FuelStationSys when that sale was created for that pump. Eliding details, our decomposed system has at least the two design components shown below, with a *directed* association to support their required interactions.



```
class SaleRecord {
  data: amount: Money; cashLimit: Money; pump: Pump;
  services: updateAndCheck (vol: Volume, grade: Grade);
}

SaleRecord::updateAndCheck (vol: Volume, grade: Grade) {
  amount := vol * grade.price;
  if (amount > cashLimit) then pump.stop();
}
```

We have recursively refined interactions, refining the corresponding specification model, and then assigned responsibilities to new “design” objects.

5.3 The Role of Refinement in Design

Clear refinement steps support traceability.

‘Refine’ (verb) means making a design decision, carrying development a step towards an implementation. In theory, separating what/who/how decisions leads to a step-by-step process of constructing models. ‘Refinement’ (noun) means the relationship between two descriptions, in which it is claimed that one correctly fulfills the other. A refinement should be documented with some explanation of why the designer believes this claim is justified, and why these decisions were made rather than others.

Any development can be thought of as a string of refinements, from tentative specifications at the beginning to solid code at the end.

Actual development is rarely traced to elementary refinement steps.

Any refinement can be understood as a sequence of elementary refinement steps; they fall into a number of different kinds, some of which we have discussed. In practice, one rarely performs a refinement step by step. It should nevertheless be possible to understand a refinement in those terms, and it should be clear to designer and reviewers that it *could* be pulled apart into those elementary pieces, should any critical questions arise.

Refinement can incorporate significant architectural transformations.

Refinement is often based on the discovery of existing components or patterns of interactions which fit the needs of the application, rather than their creation from scratch. For example, architectural decisions may be a strong force in some refinement steps, as major existing components or patterns of interacting components may be re-used. The main task then is to capture the justification — i.e. to verify that the components discovered, configured, or newly invented perform as required.

In reality, there are several valid and useful refinements.

This paper illustrates a few of a repertoire of refinements which include:

- strengthening a specification
- transaction decomposition: adding temporal detail to a transaction
- localization to role: assigning responsibilities to the participants in a transaction
- signature reification: refining specification types in transaction signatures
- role composition: one object playing several roles, often based on design patterns
- object decomposition: implementing an object by a set of objects whose collaborations fulfill the original object's assigned responsibilities
- localization to specification types: using specification types as convenient hangers for services, instead of attaching all behaviors at the "system" level
- reification of *connectors*: designing the mechanisms which will support interactions

And elementary refinement steps will rarely be documented separately.

Of course, real development may use larger steps which are compositions of these refinements, iterative changes to specifications, etc. Separate documents are not produced for every little step, but only for the main ones — initial specification, major design decompositions, specifications of components, smaller designs.

Design will recursively apply these constructs, addressing several concerns

Our entire design process is recursive, and our heuristics help address the following issues:

- Specify and "implement" (by decomposition) recursively: We start with a set of "design objects" at the business level, with joint behaviors and specification models. We then assign responsibilities, possibly adding new "design" objects. We decompose some or all of these "design objects" into component design objects, specify their joint behavior, then recursive apply the same principles.
- Identify essential dependencies across these design components. Then, strive to parameterize these components in such a way that the essential dependencies can be realized simply by a specific configuration of the generic components. We arrive at a set of *directed* associations between these design components, representing query connections which must be 'remembered' at that level of refinement granularity.
- Design by contract: Each joint or service transaction uses pre and postconditions and invariants, even at the design and implementation levels. We exploit concepts analogous to polymorphism and late binding for both specification and implementation. We also use multiple views and multiple types per design object.

- Consider design criteria: Responsibilities and interactions are designed based on coupling, cohesion, and reliability. We factor in OO-specific forms of coupling and solutions to them, and exploit objects for exception handling.
- Apply design patterns and heuristics: Each step of decomposition might incorporate known interaction patterns as part of the architecture at that level.
- Construct using frameworks: Frameworks define “plug-points” for extensibility, and provide flexible compositional mechanisms in implementations.
- Make decisions about *technical architecture*: hardware and software platforms, local processing capabilities, communication topology and bandwidths, and the mapping of domain objects to this architecture.

OMT/Fusion: The Design Process

OMT recommends two stages to design: system design and object design. System design defines architecture, including sub-systems, processes, data-storage, etc. Object design combines the three analysis models, assigns operations to objects, and selects directions for associations.

Fusion addresses the design of object visibility carefully in the design process, and produces two models: an interaction graph which describes object interactions (including creation) for specific behaviors, and a visibility graph, describing the nature of inter-object connections.

Interaction Graphs

CATALYSIS uses a notation similar to the interaction graph, sometimes preferring the time-line version used with scenarios. In CATALYSIS the sequences of interactions can themselves be constrained by extended sequence expressions describing protocols, and specific interactions graphs can be automatically checked against this. In addition, we have identified several heuristics for good distribution of responsibilities and clean interaction design, including patterns for flexible and configurable objects. We use pre/post conditions and mutual models even at this level of work, and carry this approach down to implementations in C++ and Smalltalk. These techniques are used quite uniformly in CATALYSIS from domain description to code, varying mainly in the degree of rigor that is practical or recommended at each level.

Visibility Graphs

The Fusion visibility graph captures several aspects of inter-object connections. Due to the nature of refinement and the clear notion of time-granularity in CATALYSIS, not all of these are necessary. Where helpful, they can frequently be generated automatically from the refinement relationships themselves.

Persistent vs. transient is a matter of granularity.

Fusion distinguishes between dynamic and permanent references. However, at the analysis level, Fusion only describes “static” properties in the system object model i.e. things needed to describe the pre/postconditions of operations. Since CATALYSIS can describe many levels of refinement, what is a temporary model association (and hence not even shown in the model) at a coarser level of granularity might be a “permanent” association at a finer level of time granularity, even without decomposing to internal interactions.

Fusion identifies objects which can be referred to by more than one server (shared), and also identifies lifetime dependencies between objects (e.g. delete dependencies). CATALYSIS combines these notions into one. There may be directed associations between the “components” at any level of description, from domain to code. These associations may be annotated with multiplicity symbols and invariants (helping define exclusivity), and with const. Together, these seem to cover common usage of exclusive access and of lifetime dependencies.

6.0 Conclusions

We have described the principle features of CATALYSIS, an advanced object-oriented analysis and design method. CATALYSIS addresses numerous shortcomings and makes significant extensions to methods like OMT and Fusion. The major characteristics of the method are:

- Clear conceptual separation of descriptions into three layers of *what*, *who*, and *how*.
- Support for incremental development and mixed description levels
- Integration of several important forms of abstraction and refinement
- Support for architecture and design pattern transformations in the recursive nature of the process from domain to implementation
- Support for multiple roles per object, enabling support for synthesis based upon design patterns
- Very strong support for traceability based on a variety of refinements
- Suitability for automated tool support and mechanical semantic checks
- Support for both forward-engineering and re-engineering of systems
- Use of several proven descriptive models and visual notations with clear semantics
- Customized approaches for the scope of development efforts for projects, products, and product-families; we try to characterize the variability factors across each, and use these to define units of re-use and configurability.

There are several other aspects of CATALYSIS that we do not have the room to describe here, including specific heuristics, required deliverables, recommended process, transition and training issues, tool support, re-engineering systems, and characterizing variability. More information is available in our forthcoming book, and in technical reports. Current work includes integrating seamless support for meta-types, adopting a more uniform notion of “pattern” to describe most aspects of the modeling, richer treatment of role modeling, and exploiting the “rely-guarantee” basis for concurrency.

7.0 Acknowledgments

The authors would like to acknowledge extremely useful comments and insights shed by Doug Lea, Petter Graff, Vladimir Bacvanski, and Rick van Rein. Thanks also to Derek Coleman for his helpful comments on an early version of this paper.

8.0 References

- [Boo94] “Object-Oriented Analysis and Design with Applications”, G. Booch, Benjamin Cummings, 1994
- [Cat95] “CATALYSIS: *Rigorous Object Development*”, D. D’Souza, A. Wills, P. Graff, to be published 1995
- [Cat95a] “CATALYSIS: *The Meta Model*”: available from authors; URL: <http://www.iconcomp.com>
- [Cat95b] “CATALYSIS: *A complete Case Study*”: available from authors; URL: <http://www.iconcomp.com>
- [Cat95c] “CATALYSIS: *Summary of Notation and Process*”: available from authors; also URL: <http://www.iconcomp.com>
- [Col93] “*The Fusion Method*”, D. Coleman et al, Prentice Hall, 1993

- [Coo94] “*Designing Object Systems with Syntropy*”, S. Cook and J. Daniels, Prentice Hall, 1994
- [Dso93] “*A Comparison of OO Methods*”, D. D’Souza, OOPSLA ‘93 tutorial notes; also URL: <http://www.iconcomp.com>
- [Dso94] “*OMT Model Integration*”, D. D’Souza, Report On Object-Oriented Analysis and Design, Sept. 1994; also URL: <http://www.iconcomp.com>
- [Gam94] “*Design Patterns: Elements of Reusable Object-Oriented Software*”, E. Gamma et al, Addison Wesley, 1994
- [Jac91] “*Object-Oriented Software Engineering*”, I. Jacobsen et al., ACM Press, 1991
- [Jav95] “*The Java Language Specification*”, Sun Microsystems, 1995
- [Jon86] *Systematic Software Development Using VDM*, C.B.Jones, Prentice Hall, 1986
- [Kur90] “*Object-Oriented Specification of Reactive Systems*”, R. Kurki-Suonio et al, Proceedings of the International Conference of Software Engineering, 1990.
- [Lea95] “*PSL: Protocols and Pragmatics for Open Systems*”, D. Lea and J. Marlowe, ECOOP 1995.
- [OLE93] “*Object Linking Embedding, Version 2.0*”, Microsoft Release Notes and Technical Specs, 1993
- [Ope95] “*What is OpenDoc?*” URL: http://www.info.apple.com/dev/du/intro_to_opendoc/iod0_index.html
- [Rum91] “*Object-Oriented Modeling and Design*”, J. Rumbaugh et al, Prentice Hall, 1991
- [Shl92] “*Object-Lifecycles: Modeling the world in states*”, S. Shlaer and S. Mellor, Prentice Hall, 1992
- [Wil93] “*Refinement in Fresco*”, A. Wills, K. Lano (ed), *Case studies in OO Refinement*, Prentice Hall, 1993