Refine:
A Pattern Language for *Catalysis*

Desmond D'Souza and Alan Wills

dsouza@iconcomp.com

Overview

This is a (fledgling) pattern language for the systematic modeling and construction of object systems, based on techniques from the Catalysis method. The language is called *Refine* because it is centered around the notion of refinement and composition of partial descriptions and multiple views. It permits more to be said about an object, type, action, or collaboration at any time, and supports both significant transformations, as well as a much more "elaborative" approach, to modeling.

There are two distinct parts to these patterns. The first set of patterns describes the principles underlying the modeling approach of Catalysis. The next set of patterns describes the modeling constructs themselves, and shows how they support these underlying principles.

The "show how they support" part is not yet complete, but we hope that at least some of the support is evident. A couple of patterns have been referred to but not yet written up. The paper also needs an overview diagram with some introductory commentary -- the short tabular listings should help. Also, an 'example' section for each pattern would help, but that would ruin the 1-page aesthetics!

1. Refine

To describe, build, and maintain a complex system:

1. Build abstract partial models of different views of the system to capture specific aspects while deferring others.
2. Define your development process based on which issues to tackle before others.
3. Make the abstract models precise, even when they are partial and incomplete. Review abstract models with concrete instantiations.
4. Successively refine or compose partial models to capture new aspects. Build explicit design artifacts -- components and traceability links -- for all significant decisions.
5. Focus your models on desired behavior wherever possible.

To effective capture desired behavior:

1. Focus on *type* rather than *class* to describe an object's behavior.
2. Build an explicit model of the terminology underlying its services. Make that model precise by a set of hypothesized typed queries.
3. Depict the queries using familiar visual notations.
4. Describe collaborative object behaviors explicitly. Permit an object to participate in multiple collaborations, hence to exhibit multiple types.
5. Explicitly capture the *refinement* relationships between abstract and concrete models, even if significant transformations are involved, by documenting a *retrieval*.
6. Distinguish between the refinement of a *collaboration* -- all involved parties are refined; and the refinement of a single type -- clients are not affected by the refinement.
7. Distinguish between type/subtype -- a technique to factor and compose partial behavior specification; and class/subclass -- a technique to define partial implementations subject to extension and overriding.

1 *Principles*

The first set of patterns describes the principles underlying the modeling approach of Catalysis.

Name	Problem	Solution
Decision-Driven Process	We want to expose decisions that need to be made early. At the same time we can only focus on key issues if we consciously defer others. On what basis should we define our process and models?	Define your "process" to prioritize decisions to be made and strategies for recording decisions. Build abstract models to effectively capture decisions of interest while not precipitating others.
Refutable Abstraction	Abstract models are often vague. Code is precise but humans need more abstract descriptions. Abstract descriptions do not have a tradition of precision and often cannot be checked. Not trusted or maintained, they erode over time.	Make all descriptions, from requirements to abstract models to architecture through code, precise enough to be refuted. Maintain the most abstract description refuted by any design change.
Concrete Review	Abstract descriptions can be difficult to review and validate with any degree of confidence.	Review concrete instantiations of any abstract model.
Design Capital	Many key insights are emerge during development. Code does not capture these aspects well.	Treat every significant design element as a design component. Treat every major design decision as a traceability link from concrete to abstract.
Bootstrap	A design activity will have to start somewhere. Modeling will sometimes stall and need to be kick-started. How to bootstrap?	Provide concrete steps for initial modeling, with some redundancy and consistency checks between models, and definite deliverables and checklists. Compose models bottom-up from others.
Behavior-driven	The crucial and most difficult aspect of modeling is to describe required behavior.	Use behavior considerations to uncover a supporting abstract Type Model. Retain different focuses at different times, and keep behavior described partial.
Others	Tower of "why", mixed-levels, traits	

2. Decision-Driven Process

See: Alan Hopley's PLOP-95 paper

Problem On what basis should you define a software-development process?

Forces We want to expose decisions that need to be made early, At the same time we can only focus on key issues if we consciously defer others.

Strategy Focus your "process" on prioritizing and ordering decisions to be made, and on strategies for making and recording those decisions effectively.

Select the models/deliverables for your process so that they effectively capture the decisions you make without precipitating those you want to defer. This means building abstract models which focus on the decisions to be captured.

Result A development process which is, at any time, focused strongly on making and propagating certain key decisions while deferring others. These decisions are captured in abstract models, omitting aspects of decisions which are deferred, using *Refutable Abstraction*.

3. Refutable Abstraction

Produce descriptions which are both abstract and precise.

Context Developing by focusing on key issues while deferring other decisions.

Problem Abstraction means separating the most important aspects of a problem from the details, enabling us to tackle first things first. As such, it is essential. Yet, abstraction often implies "fuzzy" or "vague". How should we approach building abstract descriptions without losing much of their value to vagueness?

Forces Humans need abstract descriptions of problems to be able to tackle complexity. Unfortunately, abstract descriptions do not have a tradition of precision and often cannot be checked in any meaningful way. Requirements specs or high-level architectures, while successful at deferring details, often indulge in astonishing amounts of hand-waving, avoiding complex but critical issues under the ruse of *"...oh, that is a design detail!"*. Consequently, problematic issues are not uncovered except in code. The abstractions are thus not trusted or maintained, and erode over time.

Code is precise, as it must be executed by a machine. Unfortunately, code is too detailed for many purposes.

Inappropriate or excessive formalism can stifle creativity. However, absence of precision can hide serious misunderstandings and ambiguities until very late in a development cycle, with costly results.

Strategy Precision does not imply implementation detail, and abstraction does not imply ambiguity. Make all recorded descriptions, from requirements through architecture to code, precise enough to be refuted e.g. *Model Based Behavior*.

When modifying the system, always maintain the most abstract level of description which is refuted by the changes made. Use this precision to focus insights and clarify ideas and uncover questions, but not as an end in itself. For any model required by the development process, clearly describe how that description can be reviewed and refuted.

Result We will have precise yet abstract descriptions of our problems e.g. using *Type Model*, *Collaboration*, and *Abstract Queries*. Since these descriptions are abstract, they will be subject to *Refinement*. Because they are precise, we can maintain well-defined traceability to them using *Retrieval*. Use *Concrete Review* to check the models.

4. Concrete Review

Review any abstraction by concrete instantiations.

Context We have built abstract models of some system or component, including user-tasks, system services, etc., using *Refutable Abstraction*.

Problem How can we communicate, validate, or test our models?

Forces It is very easy to hand-wave and agree about abstract descriptions (particularly pictures; they look impressive yet pictures do not crash!). As a result, we run an increased risk of overlooking important errors in our descriptions.

On the other hand, the abstract description is precisely what we need to validate. If abstractions are not validated, they will only serve as interim products for temporary communication, to be discarded or become irrelevant. For example, demonstrating working code does not, in itself, validate an architectural model unless the link between them can be established.

Strategy For any abstraction in the model, review instances of that abstraction together with the abstraction itself. As part of the review, present examples of how the abstractions might be refuted. Conduct the review in a vocabulary appropriate for the *context* of the current components being reviewed [*Absorb Context*].

For example, for each user-task (use-case), generate scenarios of interactions. For each occurrence of an action in the scenario, review an instantiation of the *Type Model* before and after that action.

Show how concrete exceptions or illegal examples are either handled or excluded by the abstract descriptions.

Result Abstractions validated or refuted by concrete examples.

5. Design Capital

A design repository should contain not just code, but type and collaboration specs, and traceability links to justify design choices.

Problem Code does not effectively capture all important design choices and problem domain insights of a system. How then should we record and maintain these design intents and insights?

Forces During iterative development or maintenance, previous design decisions must be understood, re-visited and possibly revised, due to changing requirements or new knowledge about issues like performance and flexibility. Each such pass often generalizes and then re-specializes previous designs. This needs very clear traceability links between our levels of description to help us maintain our designs and models effectively. However, the purpose of most iterations is primarily to produce an enhanced version of the implementation.

Strategy Build libraries to include not just code, but interface descriptions, design specifications, abstract patterns, and frameworks. Your development tools should support re-use of the more abstract descriptions as well as of implementation code from this library.

During a development cycle, in addition to populating or extending such a library, also leave a trail of design decisions. Treat each significant design element as a component, described either as a *Type* or as a *Collaboration*. Describe each significant design decision as a traceability link, using *Refinement* and *Retrieval*. Treat each major change to a design as a new *Configuration*.

Result A library of implementation and design components with refinement and retrieval relationships between parts.

6. Bootstrap

Problem A design activity will have to start somewhere. Modeling will sometimes stall and need to be kick-started.

Forces

Strategy Provide concrete steps for initial modeling, driven by scenarios of use since these are the most readily available.

Support some redundancy between models, so the same phenomenon can be considered from different perspectives. Provide consistency checks between models. Exercising such checks will often uncover changes to a model and trigger further insights.

Have definite deliverables and checklists for them. If developers clearly understand what will be produced they will better maintain momentum.

Compose all models from other descriptions. e.g. compose more elementary collaborations and types to build new collaborations, or build new type models out of other types. Do the same for requirements themselves, using *Trait*s. This way models are being built on top of smaller and simpler models, instead of from scratch.

For example, we might start to tackle a telephone network by independently modeling an individual telephone instrument and a single switch, before trying to compose them together.

7. Behavior-Driven

Focus on describing desired behavior.

Problem

There is a long-standing controversy between "data-driven" and "behavior-driven" approaches to modeling and design. Focusing just on the data in a system can lead to fragile, often implementation-oriented descriptions. However, describing the behavior of a system without any discussion of the information in that system can be very difficult.

Strategy

The ultimate objective, of course, is to describe the desired behavior of any part, or collaboration between parts.

Keep behavior as the primary focus. Use this behavior to induce a *Type Model* which supports an unambiguous description of the behavior in terms of this type model. Be careful to interpret the type model as an hypothesized model consisting of a set of *Abstract Queries*, rather than a concrete stored-data model.

If you are describing the behavior of a single object, use *Type*. If describing a joint behavior of a group of objects, use *Collaboration*.

Exploit *Multiple Types* to permit the same object to be seen in different ways, so you can build partial behavior descriptions and then compose them more readily.

2 Models

The next set of patterns describe the modeling constructs themselves, and attempt to show how they support these underlying principles.

Name	Problem	Solution
Type	We want to describe the behavior required externally of an object. This is normally done using a class, but there are many ways to implement behavior in a class, and to factor that implementation with sub-classing or composition.	Use a Type to describe the behavior of some objects without constraining the implementation of those behaviors. It is minimally an interface, and ideally a type model with behavior specification and rules.
Model-based behavior	An interface consisting solely of signatures is too shallow. Describing strictly "visible" behavior is difficult. Individually commented services tend to diverge in vocabulary unless a concrete artifact keeps their terminology focused.	All related behaviors must be described in a common vocabulary. Informally, a dictionary; formally, a type-model with behavior specs.
Type Model	How to make model-based behavior description precise, checkable, refinable, and traceable.	Formalize the model as a set of abstract queries use to specify behavior i.e. a type model.
Abstract Spec and Query	We often need to only partially describe behaviors, to allow for variations. We would like the corresponding models to be more "open" and extensible, yet precise to constrain extensions.	Use parameterized queries, convenience queries and invariants, and "relational" as opposed to "functional" description of behavior.
Visual Type Model	Text is sometimes best, diagrams sometimes preferred. However, diagrams are too often interpreted in implementation terms.	Standard "object-model" notation has type-model semantics.
Collaboration	Any design (architecture) is defined by the collaborative behavior of a group of components: roles and responsibilities, interactions, and rules within and outside the collaboration.	Collaboration: a set of actions between objects playing different roles. These actions are based on a shared model.
Client's Model	Many alternative models can be used to describe the same external behavior.	Bias the model towards the client (or at least "mutual") vocabulary.
Model from Domain	If we have different clients, their problem statements may be very different.	Use a problem domain model.
Absorb Context	In a collaboration, all objects act on the same context type-model. In isolation, each one sees a more abstract view of that shared type model.	Project into an individual type model just those parts of the contextual joint type model required to describe its behavior.
Time Granularity	Actions and interactions can be described at many different levels of granularity.	Treat time-granularity as just one form of collaboration refinement.
Loose Role and Protocol	Collaborative behaviors must be described in a very "open" and extensible manner.	Only specify essential dependencies in the collaboration interactions.
Multiple Types	What type am I?	Objects have multiple types, depending on context/collaboration/client.

Name	Problem	Solution
Composing Collabs	Collaborations are partial descriptions, describing just some parts of an object's interactions.	Compose collaborations so the different roles that an object plays are described in different collaborations.
Generic Collaboration	Common patterns of interactions show up with completely unrelated types of objects and actual invocation names. How can we describe the common aspects?	A generic collaboration.
Sub-types	What is the difference between a sub-type and a sub-class?	Sub-types do not override behavior specifications.
Refinement and Retrieval	With multiple levels of abstraction and models how do we retain traceability and maintain the abstract models?	Explicit refinement and retrievals can make *dependencies* very visible, without any theorem-proving tools.
Class and Type	Classes implement types.	A special form of refinement and retrieval.
Dependency	Identify dependencies across different points of view, different models, requirements vs. implementation, etc.	Retrieval, invariants, traits; all supported by refutable abstraction
Trait		
High Level Overall Process		
Domain Model		
Use-Cases		
System-Context		
Type-Model		
Time-Grain		
Component Model		

8. Type

Use a Type to describe required behaviors of objects without describing implementation.

Context Applying the *Behavior Driven* approach we wish to describe an object.

Problem We want to describe externally required behavior of an object without describing its implementation.

Forces Objects are usually characterized by their class. However, there are many ways to implement the same behavior in a class, and to factor that implementation with sub-classing or composition.

A concrete class provides a complete implementation of all behaviors, but we often wish to only partially describe behaviors. An abstract class is usually used to share a partial *implementation*. It really does not have to be inherited for an object to support the required behavior i.e. even such an abstract class could just serve as a specification of behavior.

If we use classes to describe required behavior we might make implementation decisions, either about a class, or a sub-class structure.

Strategy Use a distinct construct to define externally visible behavior: *Type*.

A Type describes (partially) the behavior of some objects without constraining the implementation of those behaviors. Types will later be implemented with classes, perhaps with subclassing or composition.

A type is minimally an interface i.e. a list of services provided. Ideally, though, it should be a more precise description of required behavior based on a *Type Model* with behavior specification and rules that must be followed by any implementation.

Result We have a new kind of design component, *Type*, which will describe externally visible behaviors of some objects. We can make these behavior descriptions more precise by using *Model-based Behavior* and *Type Model*, yet keep them abstract and extensible using *Abstract Queries* and *Sub-type*. When any class implements this type we have traceability by a *Retrieval*.

A type maps very directly to languages like Java, IDL, and Active-X. In C++ an *pure* abstract class approximates a type. In Smalltalk, a message protocol can be used for the same purpose.

9. Model-based Behavior

Behavior specs need a model of abstract types and attributes.

Problem

How should we describe the intended behavior of the object so we can refute or validate it, uncover problems, capture the resolution of those problems, and produce a precise shared vocabulary?

Forces

Clarifying behavior takes time away from implementation. Clients often think that their descriptions are all that a developer needs. Analysts often equate precise descriptions with "implementation detail". In both cases, serious misunderstandings jeopardize downstream work.

A simple list of signatures leaves too many things unsaid and implicit. While we might augment the list of services with comments, this easily leads to an inconsistent and divergent vocabulary across the services.

We could describe scenarios of interactions, but this mostly helps initiate discussion about behaviors by examples. The levels of detail and vocabulary diverges unless there is a tangible artifact to keep it focused. We could build a prototype of that component. However we still need an abstract description of behavior [see *Concrete Review*].

Strategy

Construct a model describing abstract attributes of the object. Use that model as the sole vocabulary available for describing the behaviors of that object. Describe each service by relating the attributes and outputs upon completion of the service to the inputs and prior attributes. Validate with scenarios. Use the model to re-formulate the user's statements in an increasingly precise manner.

The model itself may take different degrees of formalism, from a structured glossary to a *Type Model*. The key point is that the same model be used as the sole basis for documenting all behaviors of the component. In this way, the model provides a concrete artifact to keep the vocabulary focused and convergent.

For example, we might describe a PBX component using a model which defines PBX attributes of type *Party*, *Call*, *Instrument*, *Message*, and related properties of each.

Result

We have a model of an object, and describe its behaviors in terms of that common model. To make our descriptions more precise we use *Type Model* and *Abstract Query*. To make traceability explicit we can define *Retrievals* to and from this model.

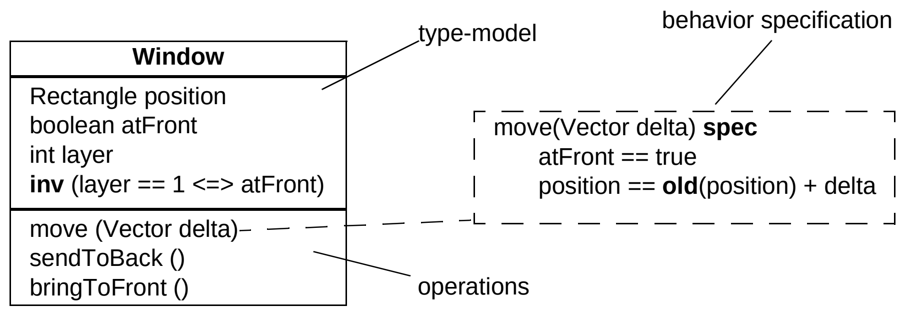
10. Type Model

A model can be formalized as a set of abstract typed queries.

Context We are using a model-based approach to specifying behavior.

Problem How can the vocabulary provided by a model be made precise enough to be checked and to enable clear traceability?

Strategy Define the model as a set of typed queries. The return type of any query is itself another type, which has its own model (set of queries). The operations can now be specified in terms of this model.



It is important that these queries be interpreted correctly. They do not represent stored data, public accessor methods, or even private methods that a designer must implement. Rather, they define just the necessary abstract state for client and implementor to understand behavior.

Any correct implementation of this type must, in some form, achieve the desired results in terms of its particular concrete representation of this abstract state [*Retrieval*].

The type model can contain invariants (**inv**), required to be true after any external operation. These invariants can constrain some of the model queries, or permit convenient but redundant model queries to be defined. For example, we could add a Screen type with a model consisting of a set of Windows. An invariant would then relate the windows and their positions and layers to the appearance of the screen.

Result Every component has a type model, and its behaviors will be specified in terms of that model. The model queries can themselves have parameters [*Abstract Queries*], and can be depicted as familiar diagrams [*Type Model Diagrams*]. As importantly, an object can be described as a composition of types [*Multiple Types*], each with its own model. In this case the different models can be related by invariants.

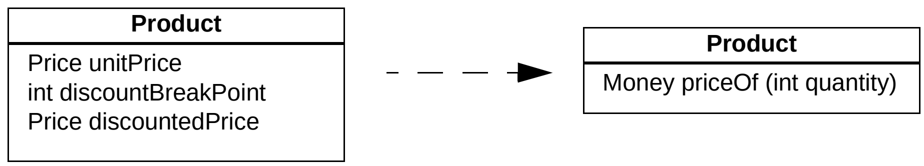
11.Abstract Spec and Query

Build partial specifications using abstract or redundant queries.

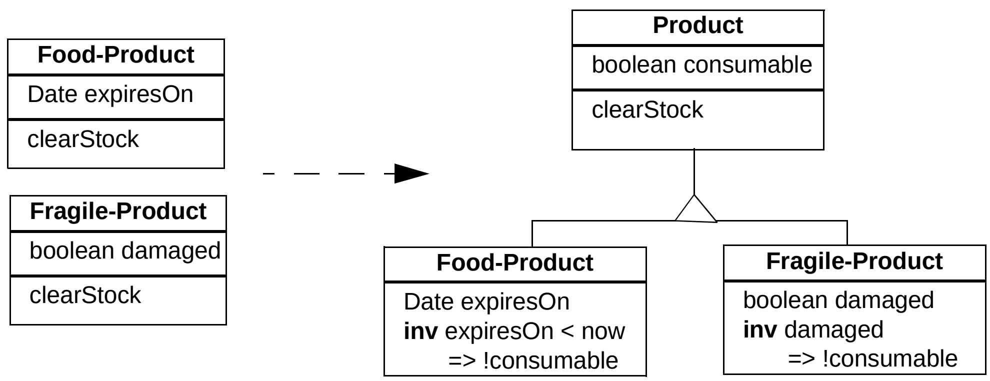
Context Define abstract behaviors with interesting specializations using *Type Model* without losing the benefit of *Refutable Abstraction*.

Problem Most interesting behavior descriptions are partial. Variations, either in the problem domain or in implementations, differ in their effects. We still want to describe the behavior accurately, though incompletely.

Strategy If you find the behavior specification starting to deal with multiple detailed cases, introduce convenient abstract queries with parameters to abstract out the variants e.g. priceOf(quantity) below.



If you find variants in some behaviors and their corresponding models, add a more abstract model with redundant queries. Subsequently you can add the variants (e.g. in sub-types) and relate it to the abstract model using invariants.



Use relations instead of deterministic functions to specify effects partially. Person::buyStuff() **spec**: self.money < **old**(self.money).

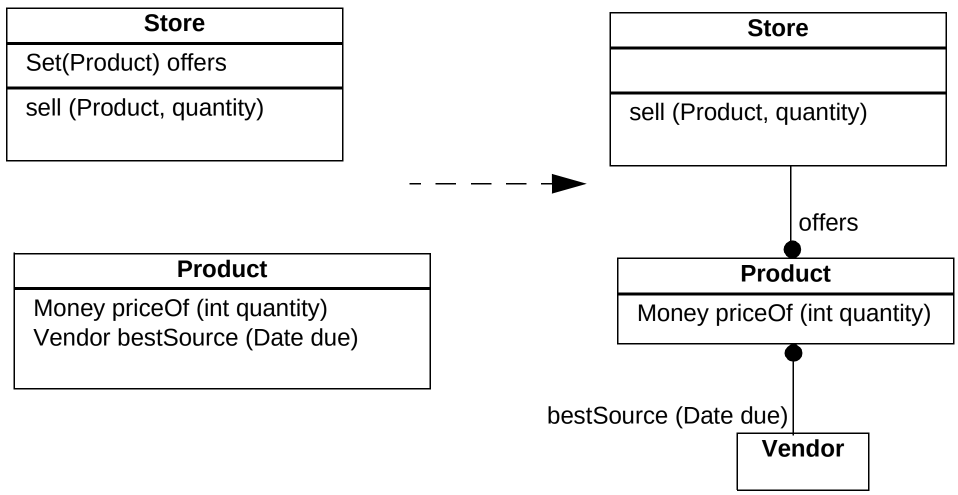
Result A set of abstract type-models with parameterized queries, or with redundant queries, and incomplete descriptions of behavior in terms of those queries. These can be depicted as a *Visual Type Model* if preferred.

12.Visual Type Model

Depict your type models using familiar diagram notations.

Problem People like diagrams. Diagrams can be powerful communication tools, and also powerfully ambiguous. Diagrams are often interpreted directly in implementation terms.

Strategy Use a visual notation to depict the queries of a type model. Where helpful, depict queries as associations connecting the types.



Make sure that this diagram is still interpreted as a type model should be: as a set of hypothesized abstract queries. Associations do not represent implementation pointers, just conceptual queries.

However, do not try to say everything in pictures! Use pictures where they help communicate your intent without change of interpretation or precision.

Result We can now define the component by its Type i.e. define its services in terms of an visual underlying model.

However, understanding a model of the component and each of its services is not enough for the client, since it does not directly address the client's intended usage of that component [*User Tasks Compose Services*]. Also, there are many possible models which can be used to describe equivalent services. We want to use one which reflects the clients view [*Client's Model*].

13.Collaboration

Use a Collaboration to specify interactions between objects.

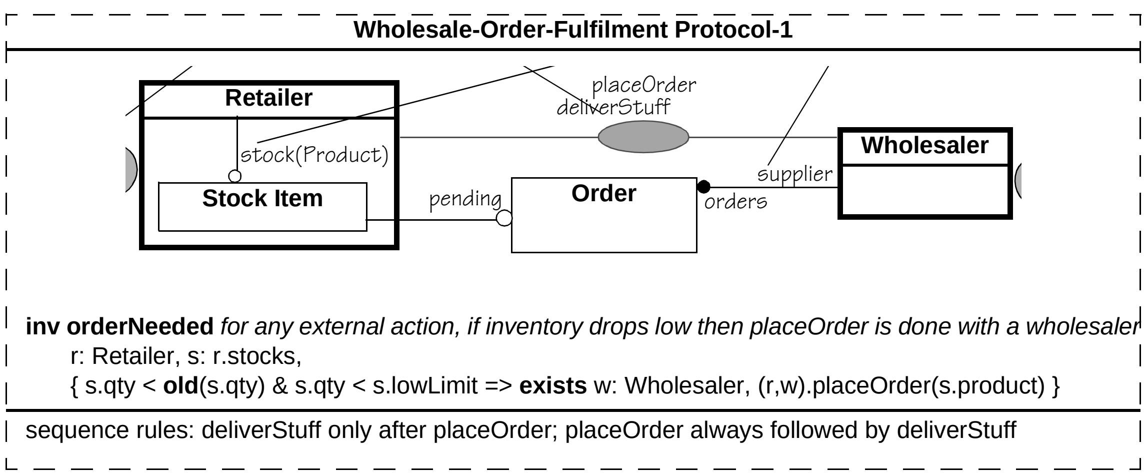
Problem The most interesting part of an object design is the interaction between objects, not individual behaviors. Many interaction patterns recur in slightly different forms. How should we describe such behaviors?

Forces An object may be described by its public and private parts. Objects interact through public interfaces. However, some interactions are more intimate than others, hence public/private is too coarse grained.

An object obeys *invariants* on its *Type Model* -- they hold on completion of any *public* operation, returning the object to a consistent state. However, these invariants may be violated in more intimate interactions.

The usual interactions diagrams simply illustrate messaging, and do not treat the collaboration as a first-class design component, with its own scoping, refinement, and composition rules.

Strategy Use a distinct construct to define a *Collaboration*. A collaboration defines a set of actions between objects playing different roles. Each role is defined as a type. The actions are partitioned into *internal* and *external* sets, and *invariants* apply to the external actions. The actions may be specified in terms of a *Type Model*, where this model now serves as a shared vocabulary for all of the actions in the collaboration.



A collaboration may be refined as a whole; all participating objects are simultaneously refined. A collaboration may be abstracted into a *generic collaboration* (which resembles a “framework”), or composed with other collaborations.

Result

A new kind of design component, *Collaboration*, describing some subset of the interactions and behavior rules of an object. As with any abstract model, a collaboration should be reviewed in a concrete form. Use scenarios for this purpose.

14. Client's Model

The terminology in the type model should not be biased towards an implementation, but should be in the client's vocabulary.

Context Using type models to support behavior descriptions, either of a type of a collaboration.

Problem There are many possible models which can be used to describe the same behaviors. Which one should we choose? A model based on implementation terms (instance variables, private methods) would typically be inappropriate for a client.

Strategy As far as possible, bias the model towards the vocabulary of the client. Depict the model and instantiations of it using names and symbols which might have meaning in the client's problem domain. Exploit *Abstract Spec and Query* towards this end where needed.

If there is no clear "client" in an interaction, then make sure the model is mutually agreed to by all parties in the interaction.

Result A model in the client's vocabulary. This is typically, but not always, the vocabulary of the problem domain which forms the context within which the object will be used [see *Absorb Context*].

Sometimes it is better to model using *Model from Domain* (perhaps a separate model) which is explicitly biased towards the problem domain and away from client-specific terminology.

In either case, we will need *Refinement* and *Retrieval* to relate the different models e.g. the problem domain, the terminology of a specific client, and the design and implementation model.

15.Model from Domain

Sometimes you should avoid the vocabulary of a specific client, and seek the deeper needs and terminology from a domain model.

Context Several clients whose requirements descriptions and terminology vary even though they are all essentially trying to solve the same problem.

Problem Client descriptions often use legacy terms to express deeper needs. We want to uncover these deeper needs and solve them, thereby meeting the client's directly stated requirements.

When we design and implement our system we would like to use a single model and set of terms based on these deeper needs.

Strategy Build a model of the problem domain itself. In some situations this will be a (set of) *Collaboration* describing some aspects of the business. Omit the target system, or planned changes to the system, from this model. Understand the needs against this model.

To extract such a problem domain model, use the *Tower of Why* family of patterns.

16. Absorb Context

The type model of any object should absorb just the necessary portions of the type models of its collaborations.

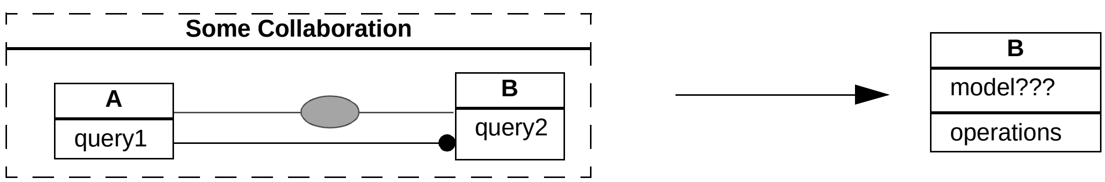
Context Moving from a *Collaboration* to focus on one of its roles.

Problem How should the type model of an individual object be related to the joint type model of the collaborations in which it participates?

Forces We can use the entire type model of the collaboration as the type model of the individual object. However, this will include aspects which could be abstracted away, at least from the perspective of this object.

We can build the individual model separately, then use *Retrieval* to relate it to the joint model. However, this will introduce unnecessary discontinuity between the individual models and its context.

Strategy In collaboration, all participants act upon a shared type model. In isolation, each one may see a restricted and more abstract view of that context (more correctly, a "projection"). Using the collaboration type model as the starting point, begin to define the behaviors of the object in isolation. Draw parts of the collaboration model into the individual model as needed to define just its behaviors.



Use *Abstract Spec and Query* to make the individual model more generic.

Use *Multiple Types* to permit this object to see the type model shared with its context through a more abstract view.

Result The context from the collaboration has been "absorbed" as needed into the type models for each role in that collaboration.

17. Time Granularity

One of the most common refinements is the granularity of interactions.

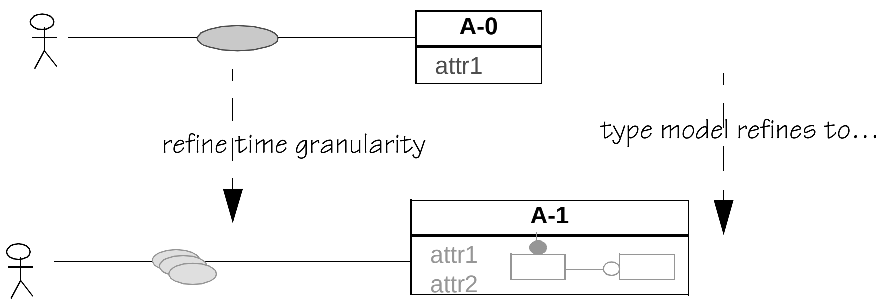
Context An approach which uses use-cases at the requirements level and object messaging at the lowest level.

A use-case is a very abstract action. For example, buy is a use-case for a customer at a gas-station. It represents a high-level action in the collaboration between customer and station.

At the lowest level, the customer must press some buttons, swipe a credit card, and so on, in order to complete a buy action.

Problem How do we relate these levels of detail.

Strategy These are two different levels of time-granularity. We have refined the collaboration between customer and station -- both customer and station are affected. In such a *collaboration refinement*, each refined object type is *not* a sub-type of the original; because the interactions between the participants is being refined, all participants are affected.



Collaboration refinement is a very common and powerful refinement.

Treat time-granularity as a form of refinement. Use *Action Reification* to build the refined model. Use *Refinement* to relate the models.

Result A set of refinements and retrievals from use-cases to code.

18.Loose Role and Protocol

Collaborations, being partial views of what an object does, have to be described in very open and extensible ways.

Context Describing the actions in a collaboration

Problem Each collaboration is a partial and incomplete description of the behavior of the objects involved. We do not want to rule out variations that might arise from other collaborations the object is participating in, or from implementation decisions made downstream.

Forces We could leave the behaviors unspecified except for illustrative examples. Of course that does not give us the benefits of *Refutable Abstraction* that we are seeking.

If we unduly constrain the behaviors we are ruling out useful extensions such as composition with other collaborations.

Strategy Describe only essential dependencies between actions. For example, avoid explicit linear sequencing unless you are sure it is required. Instead, simply specify what partial ordering of actions is permitted, and which actions have to precede or follow which others.

Similarly, do not overcommit to *who* does what. For example, if it is required that object *B* receive a request *Mb* after object *A* receives request *Ma*, do not needlessly require object *A* to send this request.

Similarly, use *Joint Actions* to further defer assignment of responsibility when necessary.

Result Collaborations which are abstract and extensible, yet precise in the rules they impose.

19.Multiple Types

Objects are seen as different types by different clients.

Context Describing an objects behaviors using a model.

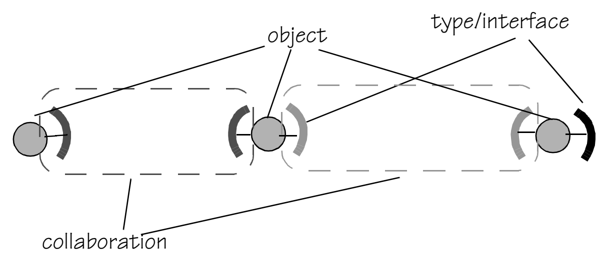
Problem Objects participate in multiple collaborations. Different collaborators view the same objects as being of a different type.

The different roles an object plays will interact in subtle ways e.g. through the state of the object, or dependencies between the actions across collaborations. Any class which implements those behaviors must implement, directly or indirectly, all the required roles and the interactions between those roles.

Yet, most useful descriptions of behaviors are abstract (partial and incomplete).

There can be many implementations of the same behavior, both in terms of the methods and supporting data members, as well as the choice of class inheritance to help re-use that implementation.

Strategy Adopt multiple types per objects as an intrinsic part of your modeling and design approach.



Each type represents one view. The behaviors in that view are expressed in terms of a model best suited to describing those behaviors. Hence the models for the different types may differ.

Construct a new model for the object so it is a *Sub-Type* of its different types. In this sub-type, relate the different models to each other using an *invariant*. If necessary, *Retrieve* the actions in the different types.

Result A view of the world well suited to Corba, COM, and Java-like systems.

20.Composing Collabs

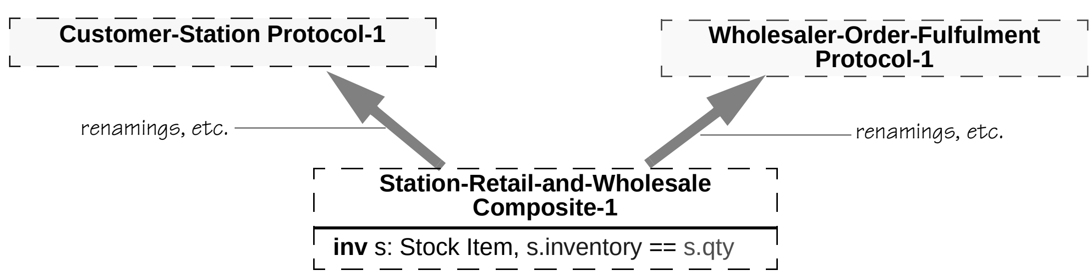
Context We have a repository of collaborations, either from a library, or describing separate and partial views of some objects in the system being built.

Problem How to combine them?

Strategy Compose collaborations. In the composition, relate the models and actions of the collaborations being refined. An object in the resulting collaboration plays zero or more roles in each of the original collaborations.

This approach supports the use of some of the design patterns in [Gamma]. For example, an object in the implementation might simultaneously play the role of *Subject* to some observer, and a *Proxy* to some remote object.

Fuller support for design and modeling patterns needs the notion of *Generic Collaboration* i.e. a collaboration which is itself parameterized by the types of some of the participants. It acts as a specification framework.



Result By composing collaborations we can better abstract recurring abstract patterns of interactions between objects, specially when we combine it with *Refinement* and *Generic Collaborations*.

21.Sub-types

Problem How to handle the fact that many languages treat sub-types as synonymous with sub-class, and permit overriding of methods in a sub-class?

Strategy Treat a sub-type as being very different from a sub-class. Sub-types do *not* override any parts of the behavior specification of their super-types. To do so would lose much of the advantage of the abstract specification of the supertype.

Use *Abstract Spec and Query* in the super-type to allow sub-types to be defined without violating guaranteed behaviors. Also, distinguish clearly between refinement of a collaboration, in which all participants are affected, and true sub-typing, in which a collaborating client will not be affected at all.

Use *Class and Type* to relate the implementation hierarchy to the types.

Result

22.Refinement and Retrieval

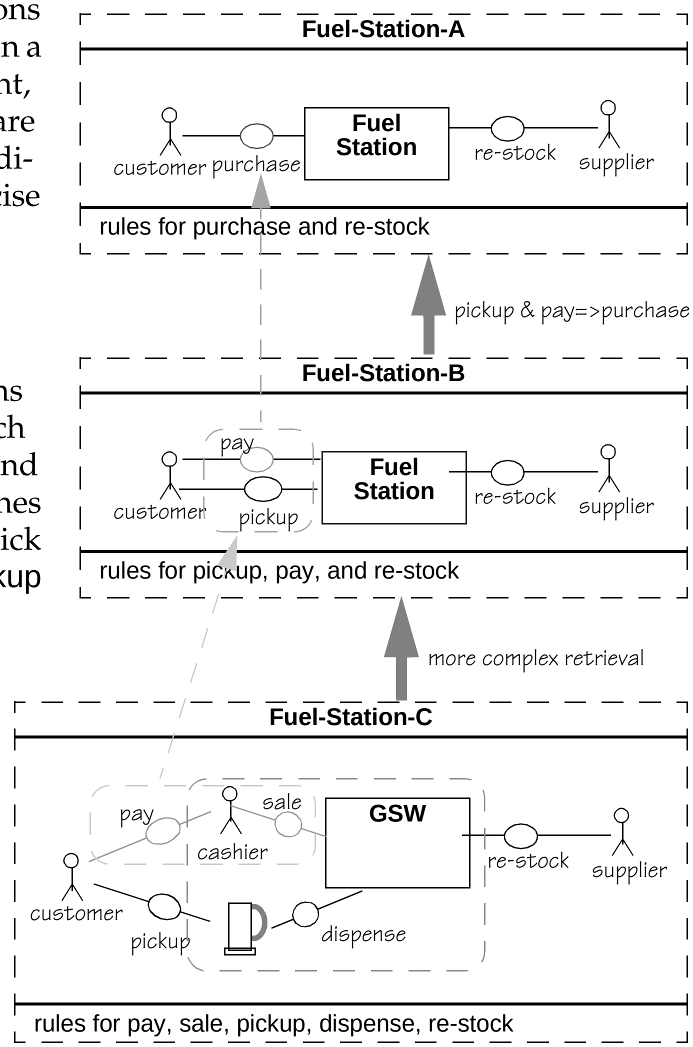
Problem With multiple levels of abstraction and models how do we retain traceability and maintain the abstract models?

Strategy Maintain explicit refinement relationships between different levels of models, for types as well as collaborations. For any refinement, document how the concrete conforms to the behaviors and rules of the abstract with a *retrieval*. A retrieval will typically relate the concrete type-model to the abstract, and sequences of concrete actions to abstract actions. In the example below the thick grey lines represent refinements, annotated with the appropriate retrieval.

The abstract problem domain interactions are described by a collaboration between a customer, the Fuel Station establishment, and the supplier. The primary actions are buy and order, with the participants indicated. Each action has abstract but precise descriptions based on a type model of Stock Items, prices, volumes.

We factored the buy action into pickup and pay. The supporting model contains additional terms to describe items which have been picked up but not yet paid and vice versa. The entire collaboration refines the more abstract — indicated by the thick grey arrow — by mapping a pair of pickup and pay to the abstract purchase.

We next decomposed the Fuel Station into constituent roles — cashier, pumps, and the Fuel-Station software system itself, GSW. We have again refined the collaboration; this time a composition of customer-cashier and cashier-system actions comprises a pay[B].



Result

- A layered set of models with explicit traceability.
- A screaming need for a serious CASE tool!

23.Class and Type

Problem How to bridge the gap between class/subclass and type/subtype?

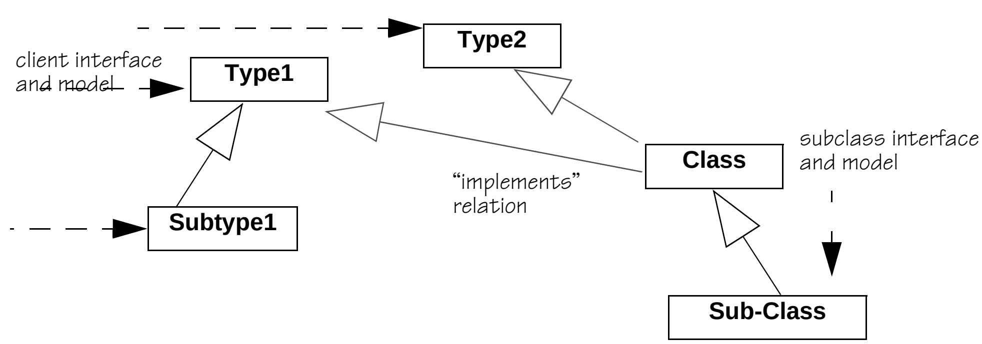
Strategy A class implements multiple types. The class selects a particular set of instance variables to be its *"model"* for all the types it implements.

Hence, the class is a refinement of any of those types. If necessary, we can document this fact by a retrieval which shows how the abstract models of each type is represented in terms of the instance variables of the class, and how the effect of methods on the instance variables corresponds to the specified effect in terms of the type models.

A sub-class inherits the complete "instance-variable model" of its super-class. It can override methods. There is a distinct interface between superclass and subclass, which becomes even more important in the presence of *template method*s.

If the superclass uses framework techniques like *template-method*, then all superclass methods which call the template method must document this call as part of their specified behavior for the subclass interface. There is a corresponding type-model required for this super-sub interface as well. Of course, this call is not relevant to the "client" interface for that superclass.

Hence, we have two specifications of the same method: one for the client, one for the subclass. Overriding of methods will still satisfy the client interface provided the overrides are consistent with respect to the subclass spec and type-model.



For frameworks design and implementation with multiple types, use abstract and generic *collaboration* to describe the framework.

3 *Acknowledgments*

Thanks to Petter Graff and Doug Lea for useful comments on the structure of these patterns. Thanks to Al Hopley for an excellent description of decision-driven development process in PLOP-95. Thanks to Doug Lea (again) for the insights provided by his work on PSL, and to OORAM for its work on “role modeling”.