Carnegie Mellon University
Research Showcase @ CMU

Department of Electrical and Computer Engineering

Carnegie Institute of Technology

1995

# Strictly class-based modeling considered harmful

Birgitte Krogh
*Carnegie Mellon University*

Carnegie Mellon University.Engineering Design Research Center.

Follow this and additional works at: http://repository.cmu.edu/ece

Recommended Citation
.

This Technical Report is brought to you for free and open access by the Carnegie Institute of Technology at Research Showcase @ CMU. It has been accepted for inclusion in Department of Electrical and Computer Engineering by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

**NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:**
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

# Strictly Class-Based Modeling Considered Harmful

**Birgitte Krogh, Sean Levy, Allen Dutoit
and Eswaran Subralunanian**

**EDRC 05-96-95**

*To appear in the Hawaii International Conference on System Sciences - £9, January 1996, Afaui, Hawaii.*

# Strictly Class-Based Modeling Considered Harmful

**Birgitte Krogh**
*Department of Mathematics and Computer Science, Aaiborg University, Aaiborg, Denmark*
**Sean Levy, Allen Dutoit & Eswaran Subrahmanian**
*Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA, USA*

## Abstract

*Many object-oriented methods have assumed class-based approaches, without considering prototype-based ones. Some authors, while admitting prototypes as a useful concept, only envision their application in early phases of systems development Others consider only the use of the prototype concept in programming environments and not with respect to the whole systems development process. We propose that these omissions are neither necessary nor useful and often can be harmful, in that methods grounded in purely class-based assumptions do not reflect the nature of problems inherent in modeling.*

*In this paper, we argue, using the principle of limited reduction, that both class-based and prototype-based approaches should be used in object-oriented methods, respectively for controlling complexity, and capturing enough detail for evolution.*[1]

## 1. Introduction

Object-oriented software development methods originated from the popularity of object-oriented programming languages, which provide powerful tools for dealing with the increasing complexity of software systems. Object-oriented methods enable the developer to build and refine a series of class-based models of both the problem (e.g. managing accounts in a bank) and the solution (e.g. software that help a bank manage its accounts).

The idea of modeling the world with class-based techniques has its roots in Aristotelian concepts, where a class consists of a set of defining properties, a set of characteristic properties, and a set of names used for the class. A class can be considered as a "template" describing the set of objects which fit that template. The contrasting notion of modeling the world with prototypes[2] have been gaining popularity in the world of object-orientation[6,13,19]. A prototype is a class-free object, completely defined by the properties which it possesses as an individual, independently from any other object in the system. At the implementation level, prototypes have been found to better support experimentation and evolution which are intrinsic characteristics of complex engineering and software products. Modeling problems that prototype-based approaches are better suited for include: multiple classification hierarchies based on orthogonal properties[6,18], unanticipated changes in the classification and changes in classifications over time [14,19].

The Principle of Limited Reduction[ll,12]proposes a relationship between the amount of information present in a system development project *{complexity)* and the reliability of that information *{uncertainty):* attempts to reduce either complexity or uncertainty will necessarily be limited by an increase in the other. In the context of system development, reducing uncertainty means increasing the amount of available information (e.g. requirements elicitation, interviews, field tests). However, while reducing uncertainty by increasing knowledge, one also increases the amount and scope of information that one needs to deal with, i.e. increases complexity. The same situation holds when attempts are made to manage complexity, i.e. by crystallizing information into a set of specification documents. If such documents are to serve their function, they must necessarily abstract away much from the total corpus of information in order to provide clear, concise guidelines on what the desired end results of the project should be. In doing so, complexity is reduced, in the sense that a mass of information has been, effectively, replaced with a more concise description that is intended to provide what was really important in the first place. However, this necessarily produces an increase in uncertainty since, especially for ill-defined problem domains, one can never be sure that the specifications capture the model needed to support the eventual users of the system.

In this paper we argue, using the principle of limited reduction, that the use of a strictly class-based approach to software development, which effectively deals with complexity, can unnecessarily increase uncertainty. We propose that using a combination of both class-based (or generally Aristotelian) and prototypical approaches could lead to similar reduction in complexity while providing better representations for managing uncertainty, since

1. This work has been supported by the Engineering Design Research Center, an NSF Engineering Research Center.
2. We wish to stress that in this paper, unless otherwise indicated, we mean "prototype" as a class free object, as opposed to the act of prototyping, i.e. creating a prototype of a piece of software for review and feedback.

prototypes allow a less restrictive model of reality than classes. We further propose that during the modeling process, prototype-based concepts *must* be used by the modelers, even while attempting to abstract the phenomena at hand into more manageable, class-based structures. Consequently, we recommend that object-oriented methods which wish to support the broadest possible slice of the lifecycle must acknowledge this fact, and provide somehow for the incorporation of models which use prototype-based object concepts, in some form or another, regardless of whether or not the target implementation language supports them.

This paper is structured as follows: Section 2 summarizes the essential characteristics of object-oriented methods in terms of their activities and products. Section 3 describes the differences between Aristotelian and prototypical concepts in modeling, and observe that object-oriented methods generally adopt an Aristotelian view of the world. Section 4 introduces the abstraction process, by which models are created and modified. Section 5 introduces the principle of limited reduction and argues the use of prototypes in that light. Section 6 is a short example illustrating the points made in Section 5

## 2. The Nature of Object-Oriented Modeling

This section summarizes the main activities and products of object-oriented methods. We first describe the software system development process in terms of phases. We then examine the activities occurring during each phase as well as their products.

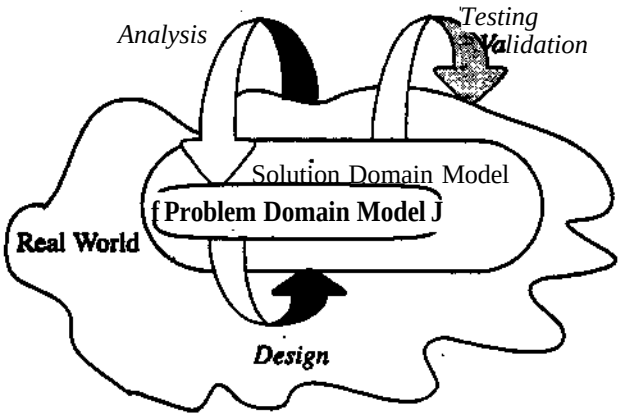
### 2.1 Object-Oriented System Development

Object-oriented modeling is the creation and modification of problem domain models (representing the relevant parts of the real world) and solution domain models (representing the system being built). In object-oriented methods, the analysis phase is the construction of an object model representing the problem domain. The design phase is the refinement of the analysis model into an object model representing the system. Ideally, the resulting design model is a super set of the analysis model (the additional classes representing non-domain related phenomena, such as user interface and storage entities). The implementation model (i.e. the code) is the realization of the design model. The consequence of this approach is that the same concepts are used at each level (analysis, design and implementation) to represent the same set of phenomena (the functionality available to the user). This provides traceability across all models which enhances the maintainability and understandability of the system. The test phase is the validation of the software system against the design model, the analysis model and the real world.

The relationship between these models and phases is illustrated in Figure 1 (inspired from [12]). Each model is represented as areas, and phases as arrows. The analysis design and test phases are pictured as transformation between areas.

All phases have effects on the models; if they do not prescribe the modification of a model, they involve getting new information on which to base or validate modifications. Since, according to Figure 1, models are created and modified in the analysis and design phases, a rigid view of the process would infer that modeling is restricted to these two phases. However, in reality phases overlap, feedback is provided to the products of early activities by iterating, and each activity that involves modifying a model requires a "counter-activity" in which the new representation is assessed[14].

**Figure 1. Phases and models involved in a system development project**




In the following section we describe the typical elements of the analysis and design phases in object-oriented system development processes.

### 2.2 Object-Oriented Modeling Activities

Even though we claim all phases influence modeling, we focus on the analysis and design phases, since this is where modeling is addressed explicitly. The parts of a process where the purpose is to define the problem or perform tests is definitely relevant to our discussion, but the details about these activities is beyond the scope of this paper. The concept of phases provides a high level and theoretical view of a process. At a more detailed and practical level, we now take a look at what kind of analysis and design activities are recommended and supported in prevailing object-oriented methods.

Monarchi and Puhr[12] identify the activities listed in Table 1 as critical during the object-oriented analysis phase. The analysis activities are categorized according to whether they consist in finding relevant phenomena (identification), organizing them based on properties (placement), or specifying dynamic behavior. However, some object-oriented methods may not emphasize all of these activities

in the same manner, or may not even provide support for all of them.

Table 1 OOA activities

| | |
|---|---|
| 1. Identification of | 2. Placement of |
| a. Semantic classes | a. Semantic classes |
| b. Attributes | b. Attributes |
| c. Behavior | c. Behavior |
| d. Relationships (generalization, aggregation, other) | 3. Specification of dynamic behavior |

The activities under 1 and 2 primarily address structure, whereas activity 3 addresses behavior. It should be noted that methods generally advise that most of those activities be performed concurrently once a core set of classes has been identified. This usually allows the analysts to explore trade-offs, e.g. whether to express specific properties as attribute or as behavior (activities l(b) and (c)), whether to abstract similar behavior into abstract classes (activities I(d)and2(c)),etc.

According to the same authors, the design phase encompasses the same set of activities applied to the solution domain (vs. the problem domain), as listed in Table 1.

Table 2 OOD activities

| |
|---|
| 1. Identification of solution domain classes, attributes, behavior and relationships |
| (same as 1. in Table 1 applied to solution domain classes) |
| 2. Placement of solution domain classes, attributes and behavior |
| (same as 2. in Table 1 applied to solution domain classes) |
| 3. Specification of dynamic behavior |
| 4. Optimization of problem and solution domain classes |

The additional classes which are identified during this phase model the user interface, the storage entities and support code (e.g. low-level objects such as queues, stacks and tables). In addition, the developer also takes into account non-functional requirements (e.g. response time, throughput and space requirements) during the design phase (activity 4). At this stage of the lifecycle, the behavior of classes is more formally described.

## 2.3 Object-Oriented Models

The result of both object-oriented analysis and design phases are class models and behavior specifications. Monarchi and Puhr [12] identify the modeling constructs listed in Table 3 as critical. Again, not all methods make use of all these constructs

Table 3 OOAD modeling constructs

| Static view | Dynamic view | Constraints |
|---|---|---|
| Objects | Communication | On structure |
| Attributes | Control/Timing | On dynamic behavior |
| Behavior | | |
| Relationships (generalization, aggregation, other) | | |

Given that most object-oriented methods primarily descend from data modeling, the modeling of structure is relatively similar across methods. However, object-oriented methods differ in their representation of behavior. For example, OMT[17] and Booch[2] use state transition diagrams to specify behavior, while OOA/OOD[3,4] use control flow diagrams. OOSE[8], which puts more emphasis on behavior as early as the analysis phase, introduces the concept of *use cases.* Use cases capture functionality as natural language descriptions of possible flows of events through the system. Each use case describes a specific set of related functions that the user of the system may accomplish.

# 3. Aristotelian and Prototypical Principles

Modeling is the representation of some chosen real world phenomena, physical or in the mind, in a way that is separate from the real world phenomena themselves. In the terms of the Beta Project[9], a program execution is a physical model simulating the behavior of a real or imaginary part of the world. Thus, the system development process is a modeling process, the purpose of which is to represent the relevant part of the real world (the problem domain) and the program execution (the solution domain). In each of these domains, abstraction processes form the connection between concepts and phenomena — the latter referred to in the model system as objects. The aspects of the real world in which the phenomena can be modelled as parts of program executions have certain qualities: substance, measurable properties thereof, and transformations of the substance and hence the properties. From an object-oriented modeling perspective, concepts of the problem domain are represented in the solution domain as class hierarchies.

A concept consists of the extension, a collection of phenomena; the intension, a collection of properties; and the designation. The latter is the collection of names used to refer to the same concept. There are two extreme views on how the intension characterizes the extension of a concept:

According to the *Aristotelian view,* the intension of a concept has two sets of properties: the defining properties that all the phenomena it refers to must have, and the characteristic properties that the phenomena may have. The extension of an Aristotelian concept is the set of all phenomena that have the defining properties, and it is objectively decidable whether or not any given object has

these properties.

According to the *prototypical view,* the intension is a set of example properties that the phenomena in the extension may, but do not need to have, and a set of prototypical phenomena that are covered by the concept. It is not objectively decidable whether or not a given phenomenon belongs to the extension of a prototypical concept.

Most modeling constructs used by object-oriented methods are Aristotelian. Classes are identified by their name and defined by their attributes and behavior. The extension of a class is objectively decidable. State transition diagrams and control flow diagrams are also Aristotelian: these representations unambiguously specify which behaviors are legal (i.e. that the system should implement).

Prototype-based programming environments are now emerging which allow the creation of class-free objects (e.g. SPLINTER[6], SELF[19], BOS[13]). It has been argued that prototype-based systems better support evolution and experimentation: class-based systems require the addition of a new class whenever the developer only wishes to modify the properties for a proper subset of the instances of a class; in prototype-based system, however, the modifications need to be done only to the relevant objects, since each individual carries a description of its structure.

Prototype concepts have not yet become part of object-oriented methods. Most object-oriented methods use Aristotelian concepts for representing structure (i.e. classes) and behavior (e.g. state diagrams). Interestingly enough, the few instances where prototypical concepts are used (timing diagrams Booch [2] and interaction diagrams OOSE[8]) are in behavior modeling activities, which are traditionally recognized as the weaker side of object-oriented methods. Interaction diagrams can be roughly described as high level call graphs: given a use case (or event flow through the system) an interaction diagram shows the sequence of all the message sends and their answering methods. This representation is prototypical, given that interaction diagrams only represent the context in which a method is invoked together with an informal description of what the method does in that context. Once all the interaction diagrams have been built, the designer may review all the contexts from which a method is invoked, and build from that information an Aristotelian view of the method (e.g. a pseudo code description, a state diagram or a control flow diagram).

## 4. The abstraction process

Abstraction processes are important to modeling activities. They take place both with regard to perceiving and structuring existing phenomena and concepts in the relevant part of the real world, and when creating and assessing the structures that form the model in the system being built. We see abstraction as the cornerstone of rational, i.e. complexity reducing, behavior. An abstraction process is the organization of knowledge, and can be seen as having three levels [9]:

- At the *Level of Empirical Concreteness,* phenomena are conceived individually, as they appear. Prototypical views (e.g. drawings, examples, simulations) are typically used at this level. The focus of this level is on understanding the problem without necessarily attempting to reduce its complexity.
- At the *Level of Abstraction,* phenomena are analyzed and concepts are formed to capture their common properties and organization. Concepts are typically expressed using the Aristotelian view at this level. The focus of this level is on expressing what was understood. For this purpose, complexity is reduced through classification and detail removal.
- The *Level of Thought Concreteness* expresses a further understanding of the total system of concepts and phenomena and provides through the structure of the concepts a deeper insight into the interrelations and events of the phenomena. At this level, the focus is on verifying the representation developed at the previous level.

Object-oriented methods mostly provide support activities at the level of abstraction. Once relevant phenomena have been identified, plenty of techniques exist for structuring and detailing class models. However, we observe that little support is provided for understanding the domain and the validation of concepts (level of empirical concreteness and the level of thought concreteness). This lack of support is especially visible in the behavior modeling activities where more traditional (and non object-oriented) representations are used. However, in a few instances mentioned previously (Booch and OOSE) prototypical representations are provided and appear to make it easier to understand and verify behavior.

## 5. The Principle of Limited Reduction

Mathiassen and Stage[ll] introduced the Principle of Limited Reduction to express the effectiveness of a design effort. It is based on the relationship between, on the one hand, what characterizes the situation at a given point in a system development process, and on the other hand, the nature of the operations to be performed in order to make progress, i.e. to get closer to the desired end product:

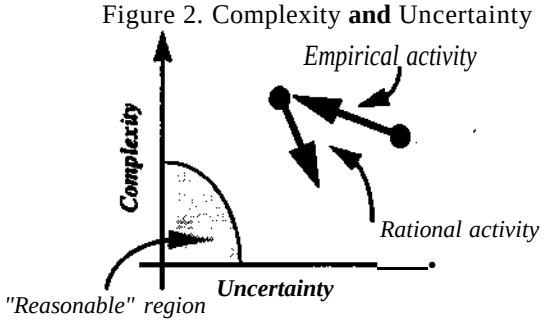
> "Relying on rational behavior to reduce complexity introduces new sources of uncertainty requiring experimental countermeasures. Correspondingly, relying on experimental behavior to reduce uncertainty introduces new sources of complexity requiring rational countermeasures." [11]

*Complexity* is used to mean the amount of relevant information that is available. *Uncertainty* is a measure of

the availability and reliability of relevant information. In a situation that is mainly characterized by uncertainty about system requirements or about the correctness of the available data, building and testing prototypes[3] of the system, or just important aspects thereof, is an often recommended strategy, as it can lead to a better knowledge base for the further development (i.e. "experimental countermeasures"). By contrast, in a situation where the main problem at hand is overwhelming or incomprehensible amount of data, the typically observed approach is analytical, with the goal being to abstract, simplify, and describe relevant information in specifications (i.e. "rational countermeasures")- The Principle of Limited Reduction proposes that the approach best suited to one type of situation invariably brings a project closer to the other type of situation.

Figure 2 shows a simple illustration of this principle. One dimension represents the degree of uncertainty in a given situation, the other the degree of complexity. Most projects presumably start out with a relatively high degree of both. The desired place to be is as close as possible to the origin — marked in the diagram as the "reasonable" region. The Principle of Limited Reduction proposes that it is not possible to move along a direct line towards the desired region; each step in the process will, depending on the situation, be of a rational or experimental nature. And while decreasing the degree in one dimension, that step will invariably increase the other dimension. Please note that time and other measurable entities are not depicted in the diagram. It is merely meant as an illustration of a principle; not of, say, the lifecycle of a project.

Figure 2. Complexity **and** Uncertainty




The extremes, where the whole emphasis is on either the rational or the experimental approach, axe referred to in [5] as the construction and evolution - approaches. Constructionism assumes a well defined problem that can be treated in an analytical manner and where it is objectively determinable whether or not requirements are met. Evolutionism assumes a dynamic environment with unstable requirements and confrontation as the only feasible way to assess a given design.

We focus on system development settings where the requirements are not well defined, where subjective assessments are crucial, and where problem definitions are part of the process. For the reasons expressed in the principle of limited reduction, the main problem when working top-down must be expected to be die lack of user feedback and hence the risk of "building the wrong system the right way". Similarly, in a bottom-up approach, the main problem will be a superficially satisfying system with a messy "interior" — due to lack of rationally designed abstractions — and hence "building the right system the wrong way".

Models of the system development process that use mixed approaches which combine the benefits of rational and experimental activities are generally accepted as better than each of the extremes. A good example of this is the Spiral Model presented in [1]. It takes into account the most prominent aspects of the problem at a given state in the development process, and defines the most efficient next steps to be performed accordingly. Purely experimental approaches (e.g. evolutionary prototyping) and purely rational approaches (e.g. the Waterfall model) can both be viewed as being subsumed by the Spiral model, and are generally only recommended when the situation is sufficiently well or ill defined for them to be applicable.

It is quite clear that there is a large gap between being able to state the principles and concepts by which modeling should be done and actually being able to operationalize them in a real system development environment We can, for instance, agree with any of the many object-oriented modeling methods that arriving at a class hierarchy that models the domain is a laudable goal, but when we encounter difficulties, how are we to proceed?

The answer lies in attempting to maintain a balance, instead of strict adherence to any one approach. Shifts in orientation between, for instance, Aristotelian and prototypical views of the modeled world are often necessary, if only in order to clarify the direction that the design of a class hierarchy should take next. Rather than focusing only on the end result of a given modeling process, i.e. a class hierarchy, attention should also be paid to the actual modeling process itself, as well as "by-products" that might result and which are not otherwise accommodated for in a given method. Put in terms of uncertainty and complexity, an Aristotelian view decreases complexity significantly more than a prototypical view while introducing more uncertainty. When validating concepts, this larger uncertainty is more difficult to reduce given that the Aristotelian view is a more restricted model of the real world than the prototypical view. We suggest that the use and alternation of both views enables a significant reduction of complexity while managing uncertainty.

**3. In the sense of a piece of software.**

## 6. An Example

In this section, we will focus our attention on an example, in order to illustrate the processes that take place during modeling. This is not meant to be a complete analysis model or any other sub-product of a systems development process, let alone empirical "evidence" for our points. We have merely attempted to present the modeling of part of a problem from the real world in just enough detail to illustrate the points relevant to our arguments without having to descend into the depths of the modeled domain. In addition, as much as possible, we have recorded the events in the exercise in the order that we did them, so as to retain the flavor of the process.

The problem domain being modeled consists of equipment and processes in a chemical plant. The purpose of the system considered is to simulate the behavior of the plant and provide users with predictions about the effect of changing the value of various factors. In particular we are interested in modeling breakdowns, i.e. the causes and consequences of failures in the equipment. The partial modeling process presented here focuses on this aspect of the system, and, where necessary, we have sacrificed chemical engineering details in the interests of exposing interesting modeling problems.

### 6.1 Pipes, Valves, and Explosions

First, we form an idea about the domain by identifying relevant objects and their relationships in terms of a class hierarchy, which serves to organize our thoughts on the matter; the initial result is depicted in Figure 3. It shows classes for the various identified objects and a superclass, equipment, for modeling the properties all the other classes have in common. The cardinalities show the number of instances involved in a particular relationship, e.g. a pipe is connected with zero, one, or two valves whereas a valve always connects two pipes. Next we want to fill in more details for the different types of objects.

In order to see how well our classification of the world works, we decide to think about how some actual objects instantiated from this hierarchy might look. We do this in order to get a better idea of what properties and behaviors the objects in our problem domain need to have to satisfy our requirements. More specifically, we look at what kind of things can happen in the pipe-valve relationship and what this means for the model. For lack of better diagrammatic conventions, a style of drawing the actual instances and their relationships developed more or less spontaneously on the blackboard, shown in part in Figure 4. Here we see the interconnection of two pipes via a valve. Each of the ovals in the figure represents an actual object (i.e. not a class) because we are interested here in testing one specific situation and finding out what influence it must have on the class model. There are three distinct kinds of references involved: the valve *connects* to the two pipes, i.e. its input and output, and the pipes have *input* and *output.* The side-bars attached to pipe 1 and the valve are lists of messages that such objects might understand and/or properties they might have. As we are still in the early stages of modeling the problem, the distinction is not completely clear yet.

For instance, we can posit with reasonable certainty that pipes and valves both have a diameter, a thickness and are made of some material. What is not necessarily clear yet is whether, for example, diameter is an entirely static property or one which must be (at times) calculated (can the pipes in our world become occluded?).

**Figure 3. Partial Class Diagram**[4]

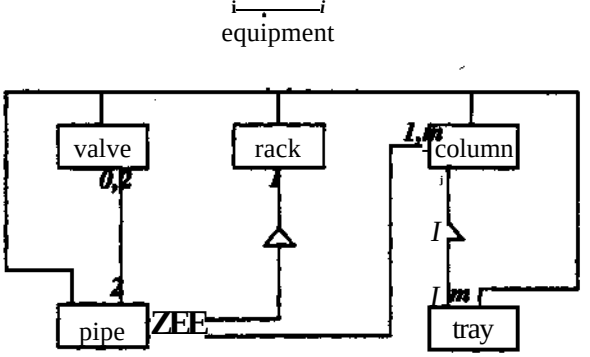
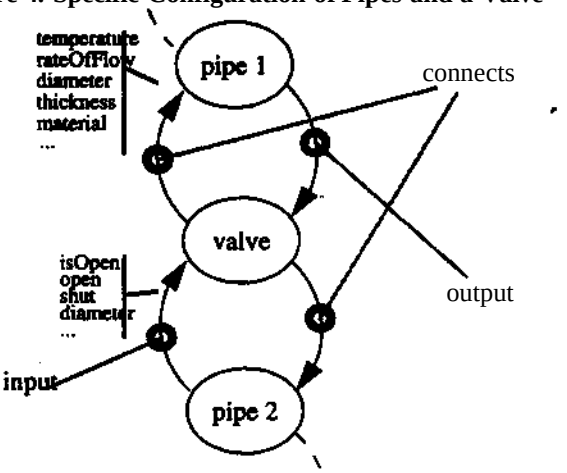


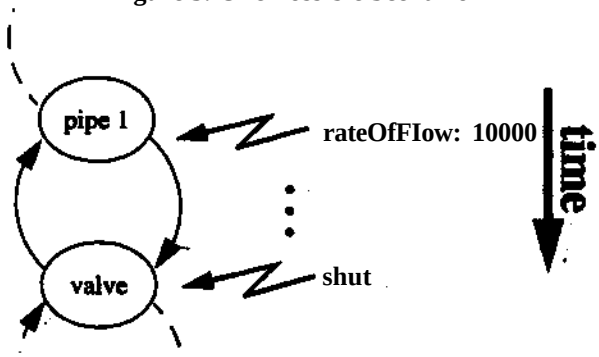

**Figure 4. Specific Configuration of Pipes and a Valve**




Given this crude model of a piece of the real world, our next step is to think about how parts of it respond to messages. As before, we start this out by depicting a specific situation since we do not have an overall generic description of the system's behavior. Figure 5 shows pipe 1 being told to set its rate of flow to 10000 units followed, after some interval of time, by a message to the valve to shut itself.

4. simplified Coad/Yourdon notation; a dotted line means an abstract class, i.e. a class with no instances

**Figure 5. One Possible Scenario**



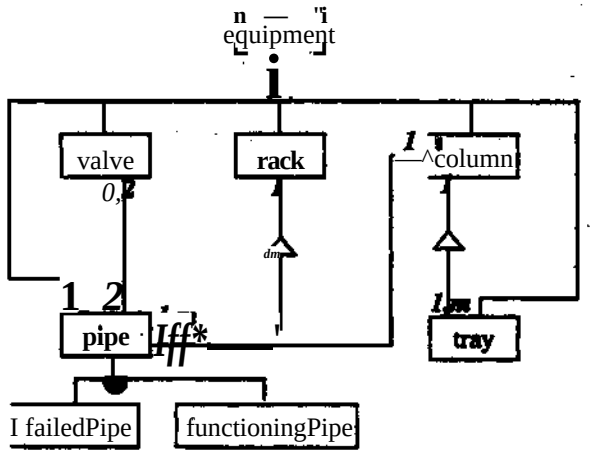

Suppose that our model requires that when the valve is shut, the incoming pipe should experience a failure, e.g. explode. Clearly, a pipe which has exploded should answer at least some of its messages differently than a pipe which has not. Further, an exploded pipe can not make a transition back to a "normal** state, as opposed to, for instance, a pipe which has been exposed to extreme cold. This would seem to indicate that there are at least two kinds of failures, which might be *calledfatal and fixable.* So we want to incorporate this aspect of the pipes, having different kinds of failure states, into the class model.

One possible way of dealing with the problem is to add a property to all equipment, such as *failureMode,* and push the solution into the behavioral part of the model. For instance, for each message which must have a response that varies with respect to failure, we may model it as a multi-way decision based on the value stored in *failureMode,* which might be one of *ok, fatal ox fixable.*

Although at first this may seem like a reasonable solution, it quickly becomes apparent that there are disadvantages to this approach, on several levels. If we ignore problems of implementation and restrict our attention to problems that appear at the design level, we see two main difficulties. In an ideal object-oriented model, the presence of many multi-way decisions in the specification of behavior is a sign that the model itself is not rich enough, i.e. the class structure does not adequately model the domain. In other words, we are burying important structural information in the description (or even the implementation) of classes in such a way as to make the actual relationships between the classes much richer than the diagram might indicate. Further, such a strategy would mean that any message that needed to be sensitive to failure would need to be modeled as behavior, i.e. a method, instead of as a property. This might, in the extreme, force the addition of "methods" for calculating every property of a piece of equipment, based on hidden internal state, a move which, at the very least obfuscates the meaning of the model a great deal.

An alternate approach would be to add a property to all equipment which refers to a *Failure* object Functioning equipment would refer to a dummy failure object whose methods are null operations. A failed pipe would refer to a failure object which represents the particular failure it experienced. Then, all messages answered by a failed pipe would be forwarded to the failure object it refers to, instead of being directly answered by the Pipe class. This approach would have the advantage of removing the multi-way decisions introduced by the prior solution. However, this solution still suffers the problem that all properties of all types of equipment would have to be represented as methods.

**Figure 6. Revised "class" diagram**




A better solution would be to consider the pipe objects as belonging to different "classes" over time, e.g. *&failedPipe* class and *afuncHoningPipe* class, depending on their failure status. Better yet, individual pipe objects could be dynamically changed depending on the kind of failures which would happen to them. The concept of dynamic inheritance and partial inheritance have been introduced for at the programming language level ([19] and [6], respectively. Use of such concepts could be extended to the design or even analysis phases of an object-oriented method. The revised "class" diagram is pictured in Figure 6.

It should be noted that the inheritance between *failedPipe, functioningPipe* and *pipe* is not equivalent to a standard class-based inheritance relationship. An instance of *pipe* retains its identity through failures and repairs, i.e. it is not replaced by a new instance of another class. This has the advantage that other objects referring to the failed pipe (e.g. a valve, a rack) need not change their references to the failed pipe. This inheritance relationship is not equivalent to multiple inheritance either, given that the pipe cannot be a *failedPipe* and a *functioningPipe* at the same time.

## 6.2 Discussion of the Example

Revisiting to the three levels of modeling processes presented in Section 2.5, we can categorize some of the

activities that occurred during modeling. It can be seen as a series of informal sub-tasks each of which was focused on either: further understanding the problem domain based on the already available knowledge; structuring and otherwise describing the properties of the elements of the problem domain; or comparing the understanding and the description in order to put both to a test. The former two tasks were the most straightforward to perform, using common sense aftd the recommendations and notations of the method; the latter task were more critical and called for more creative measures.

On the level of empirical concreteness, we followed the prescriptions of at least the more widely accepted methods in attempting first to identify the *concepts* in the domain ("look for the nouns," as e.g. [3] and [16] refer to this activity). In fact, it was relatively easy, after some consultations with chemical engineers, to produce a simple class structure that described the concepts in question with respect to their defining properties. This brings us to the level of abstraction: the initial construction of the class hierarchy of the identified phenomena. Our difficulties began when we incorporate behavior into an initial model[5]. Further, the line between what was a defining property and what, in Aristotelian terms, were characteristic properties became rather blurry when an actual situation was considered. This attempt to verify or modify the model is on the level of thought concreteness; we see the already identified relevant phenomena in a new light, we obtain new insight into what is and what is not captured in the model.

In this manner we continued to shift our view between understanding the actual domain, expressing this understanding in an abstract way and, always, "confronting" these two levels with each other, which usually caused both the understanding and the representation (the current model) to change.

Although the "final" model may be a class-based model (possibly augmented with prototypical concepts such as dynamic inheritance[19], or partial inheritance[6]), the prototypical representations created during the modifications should become part of the outcome of the process (i.e. made part of the model, such as interaction diagrams in the OOSE method). The prototypical representations may then be used during the next modification of the model and provide a better context by reducing uncertainty.

## 7. Conclusion

Class-based approaches are good at dealing with complexity, but introduce uncertainty in that they can be too removed from the real world problem that is being modeled. Both Aristotelian and prototypical concepts are useful and necessary in both the modeling process and the model. They are useful at the process level because alternating between understanding and expressing — between operating at the object and class level, between constructing structures and validating them — is an intrinsic part of any modeling activity. They could also *b6* useful at the product level since parts of the real world are much better represented as prototypes[6] than instances of (sometimes artificial) classes[6,13,15,18].

In the overall perspective, both Aristotelian and prototypical approaches are complexity-reducing, but the right use of the prototype concept, along with classes, (re)introduces less uncertainty than a strictly class-based approach. The advantages of having both types of concepts to work with can be seen as an application of the principle of limited reduction, only at a more detailed modeling level, namely the "local" iteration between identifying and abstracting phenomena (rational behavior) and validating (experimental behavior) — as described by the three levels of modeling processes. In some sense, one could think of this as an application of the age-old adage: "As above, so below."

Based on these observations, we recommend that object-oriented methods take this into account by supplementing existing notational constructs with additional structures to express the concept of prototypical objects. These notations could be used, for instance, in sub-tasks when the aim is to build class structures, in which case they might be considered by-products, as well as parts of the final model. We view the construction of such notational devices as interesting questions for future research. Further, the incorporation of prototypes in to the implementation vehicles available to designers could improve the over-all process, by removing some of the artificial constraints on the end-products of modeling that have been posited by various groups. This does not necessarily mean switching to a prototype-based programming language, as some authors have begun to point out [7,13].

## 8. Acknowledgments

This work has been supported in part by the Engineering Design Research Center, an NSF Engineering Research Center. The authors would like to thank the anonymous reviewers of this paper for their insightful comments and their suggested edits.

5. This is not to say either that *(a)* we had the non-behavioral part of the model right or *(b)* that there are not domains in which modeling the actual things themselves is intrinsically difficult.

6. It Is to be noted that the term *instance* is sometimes used in place of *prototype* [15].

## 9. References


[1] Boehm, B. (1988). "A spiral model of software development and enhancement/* *IEEE Computer*, 21(5):61-72.

[2] Booch, G. (1993). *Object-oriented analysis and design with applications.* Benjamin/Cummings Publishing Co., Redwood City, CA, 2nd edition.

[3] Coad, P. and Youidon, E. (1992). *Object-oriented analysis.* Yourdon Press, Englewood Cliffs, NJ.

[4] Coad, P. and Yourdon, E. (1992). *Object-oriented design.* Yourdon Press, Englewood Cliffs, NJ.

[5] Dahlbom, B. and Mathiassen, L. (1993). *Computers in Context.* Blackwell Publishers, Cambridge, Massachusetts, USA.

[6] Demaid, A. and Zucker, J. (1992). "Prototype-oriented representation of engineering design knowledge." *Artificial Intelligence in Engineering.* (UK). 7(l)pp.47-61.

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable object-oriented software.* Addison-Wesley, Reading, Massachusetts.

[8] Jacobsen, I. etal. (1992). *Object-oriented software engineering: A use case driven approach.* Addison-Wesley, New York, NY.

[9] Madsen, O.L., Mfliler-Pedersen, B., and Nygaard, K. (1993). *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, Reading, Massachusetts.

[10] Mathiassen, L., Seewaldt, T. and Stage, J. (1995). "Prototyping and Specifying: Principles and Practices of a Mixed Approach.*' *Scandinavian Journal of Information Systems.* 7(l):55-72.

[11] Mathiassen, L. and Stage, J. (1992). "The principle of limited reduction in software design/* *Information Technology and People.* 6(2-3): 171-85.

[12] Monarchi, D. and Puhr, G. (1992). "A research typology for object-oriented analysis and design/* *Communications ofthe ACM, 35(9).*

[13] fi-dim Group (1995). "An overview of the basic object system." Technical Report EDRC 05-92-95, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA 15213, USA.

[14] Parnas, D.L. and Clements, P.C. (1986). "A rational design process: How and why to fake it/* *IEEE Transactions on Software Engineering.* SE-12(2):251-57.

[15] Parsons, J. (1994) "Instance-based Primitives for Data Modeling", *Proceedings of the Fourth Workshop on Information Technologies and Systems,* Vancouver, Canada, pp. 215-24.

[16] Rubin, K. S. and Goldberg, A. (1992). "Object behavior analysis." *Communications ofthe ACM,* 35(9):48-62.

[17] Rumbaugh, J. et. al. (1992). *Object Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ.

[18] Sargent, P.M., Subrahmanian, E., Downs, M., Greene, R. and Rishel, D. (1992) "Materials* Information and Conceptual Data Modelling" *Computerization and Networking of Materials Databases: Third Volume, ASTM STP 1140,* Thomas I. Barry and Keith W. Reynard, Ed., American Society for Testing and Materials, Philadelphia.

[19] Ungar, D. and Smith, R. B. (1991). "SELF: the power of simplicity." *USP and Symbolic Computation,* 4(3): 187-205.