Dynamic Models in Object-oriented Design

Kurt Nørmark
Aalborg University
Denmark*

Abstract

Dynamic modeling in the scope of object-oriented design is discussed and clarified. A dynamic model is seen as an abstraction of the actual program execution. In contrast, a static design model is seen as an abstraction of the program description in which the structural relationships are emphasized. The central hypothesis behind this paper is that designers and programmers think in terms of objects instead of classes, because objects and object relations are more tangible than classes and class relations. As a consequence it is recommended that dynamic design models are worked out prior to the construction of static design models. In the paper a dynamic OOD model is developed as an abstraction of the actual program execution model. A dynamic exploration tool, which works on the abstract execution model, is also described. The dynamic exploration tool is seen as a medium which matches the properties of the dynamic design model.

1 Introduction

This paper is about the role of dynamic models in object-oriented design (OOD). In this scope a *dynamic model* is an abstraction of a program execution which involves objects and their mutual interaction. The counterpart is a *static model* which is an abstraction of a program that involves classes and their structural relations. In the paper we will motivate our way of dealing with dynamic models and we will describe the approach that we follow in our ongoing research. The main purpose of the paper is to analyze possibilities and to describe the work we are carrying out.

The paper consists of three main sections:

1. Discussion and concepts.
2. Abstract execution models.

*Department of Mathematics and Computer Science, Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark. Internet: normark@iesd.auc.dk. — This research was supported in part by the Danish Natural Science Research Council, grant no. 9400911.

3. Elements of a dynamic exploration tool.

The first section serves as a broad discussion of ideas, approaches and concepts. In the second section, elements of an abstract program execution model is analyzed. The third section outlines our ideas in a concrete form via initial considerations of a tool that supports the exploration of dynamic OOD models.

Finally a description of related work from the leading OOA and OOD literature is given.

2 Discussion and concepts

What is OOD: In our context, a *design* is a plan for construction. In a program design process *models* may be made of the program (static models) as well as of the program execution (dynamic models). The static design models, which we are interested in, are centered about overall structural properties of the program. The dynamic design models focus on overall process properties of the program execution.

An *object-oriented design* (ODD) is a design (following the meaning from above) in which classes and methods play a key role in the static design models, and in which objects and messages play a key role in the dynamic design models.

The level of abstraction in an OOD model is higher than that of an object-oriented program. In an object-oriented design model we disregard a great number of details. These details are taken care of in the programming phase of the development process.

Dynamic modeling before static modeling: At the detailed programming level, it is difficult to imagine that the dynamic model (the program execution) can be explored before the static model (the program). The reason is, of course, that the program execution is governed, described, and controlled by the program in a very narrow sense. It is tempting to conclude that static modeling comes before dynamic modeling also at the object-oriented design level, but in this paper we will argue that this need not to be the case. The starting point of the argument is the following hypothesis:

> *Programmers think in terms of objects, object relations, and object interactions during the creative phases of the design process.*

This should be contrasted with the observation that designers typically start with the construction of a static model. If the hypothesis is correct, it would be more natural to formulate a dynamic model first, in order to capture the intended design at a concrete and tangible level.

At the programming level, the program can be regarded as a description of the program execution (see figure 1). At the design level the static model need not to be a description

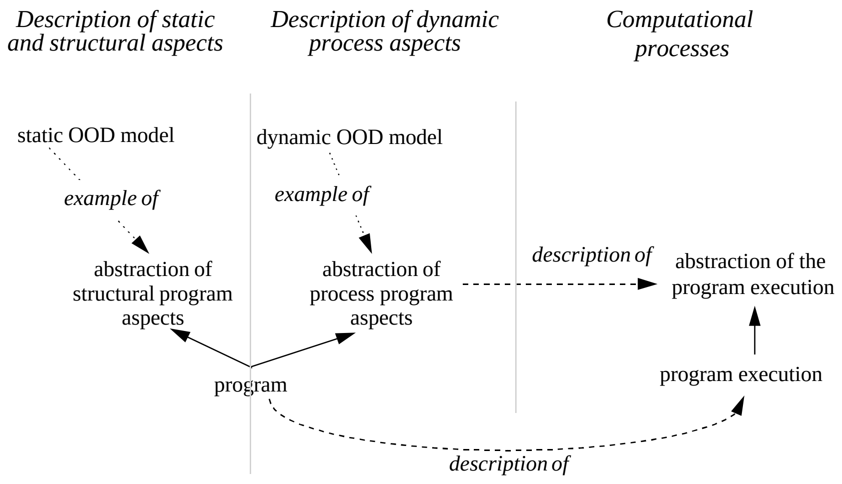

Description of static and structural aspects
Description of dynamic process aspects
Computational processes
static OOD model
example of
abstraction of structural program aspects
dynamic OOD model
example of
abstraction of process program aspects
description of
abstraction of the program execution
program execution
program
description of

Figure 1: *Static and dynamic models in relation to executions.*

of the abstracted program execution. The reason is that the typical abstractions, which are ‘applied’ on the program, disregard the descriptions of process properties in favor of structural properties. However, selected elements of the static model may be extracted from an initially developed dynamic model. This is because of the duality between objects and classes, and between messages and methods; Many object relations induce similar relations among classes, and vise versa.

Despite the hypothesis from above, we most often succeed in constructing static models early in the design process. However, when dealing with complicated design tasks the mental gap between

1. the concrete set of objects that are in play and
2. their class description counterparts

can be great. The “size of the gap” may make it difficult to capture the necessary understanding of the problem and its solution in a static design model alone. As an additional argument in favor of ‘dynamic model before static model’ novice designers may find it difficult to bridge the gap, even when dealing with relatively trivial problems.

Static is easier than dynamic: It seems to be the case that static models are better developed and that static models are more often used than dynamic models i object-oriented development processes. One reason may be that there are some major and inherent difficulties in dealing with dynamic models in comparison with static models. As the term indicates, a “dynamic model” consist of elements which appear, change characteristic, and disappear as a function of time. It is not easy to deal with the temporal dimension of dynamic models when using a static medium for descriptions, such as paper. Often, we are limited to work on a snapshot of a dynamic model which captures the objects at a given point in time. In some models, one of the dimensions in the plane (on the paper) is devoted to time; in other models particular graphical means are related to temporal aspects. A temporal progression of snapshots, which illustrates the dynamics in a movie-like fashion, would probably be more useful, but it is awkward to deal with such unwieldy descriptions on a static medium.

The role of the media: One of the ideas behind this work is to use a computer-based tool as the medium via which to *represent* dynamic OOD models. There is, of course, nothing new in using a computer-based tool for the creation and maintenance of design descriptions. When we deal with analysis and design activities various kinds of CASE tools are in widespread use. Most CASE tools are used for models of the static design and program aspects. Our idea is to use tools for creation, maintenance as well as *presentation and interpretation* of the dynamic aspects at a reasonable high abstraction level. It may be difficult to deal with, and to understand, a dynamic model through a printed text or a diagram. As a consequence we propose that a dynamic model is viewed and understood through a separate tool. As a matter of terminology we call the presentation and interpretation tool for a *dynamic exploration tool* (DET).

We prefer to think of a dynamic exploration tools as a *medium* through which to understand the properties of a dynamic model. The nature of this medium goes perfectly hand in hand with the nature of the dynamic models, which we want to work with. The nature of the DET medium is radically different from a drawing or a piece of text on paper. It takes a great deal of efforts to interpret a description on paper as "being alive" in some sense. As a medium, the paper is "dead". In contrast, a DET is able to convey liveness. In a DET, temporal aspects and dynamic behavior can be dealt with in a natural way, and much more directly than it is possible on a "dead medium".

The means of expression: In the implementation phase of program development the linguistic and textual means of expression dominate. The situation is different in the area of analysis and design. Here the diagrammatic style prevails. In general, diagrams provide overview and only little detail. In addition, a diagram may be easier to understand than a text in a formal language. Although important in some respects, we claim that the means of expression is a rather superficial property of a design artifact. A DET medium, along the lines proposed above, allows us to use a variety of different means of expressions. This is achieved by supporting several *external presentations* based on an appropriate *internal representation* of the information in the underlying tool.

The degree of formalization: Besides the means of expression we can also discuss the *degree of formalization*. A formal description lends itself towards a higher degree of precision than does an informal description. Notice that formalization and the means of expressions, as discussed above, are orthogonal to each other. (It is possible to have both diagrams and textual documents with a varying degree of formalization and precision). Recall that we are interested in dynamic models which are created early in the OOD phase of the development process. One of our goals is to capture object creation and object interaction in a way which enhance the *intuitive understanding* of the object dynamics. Consequently we will argue that considerable elements of informal explanations (in pure, natural language text) is important. We will say that our description of the OOD dynamic model are *narrative*. An informal, narrative account does not necessarily substitute precise and formal elements of descriptions. On the contrary, there may be redundant elements in the model such that both precision and intuitive understanding are enhanced.

Thus we are proposing a description of object dynamics, in which intuitive explanations go hand in hand with more formal and precise descriptions, which are appropriate at the given level of abstraction. When the resulting descriptions are static (on, e.g., a sheet of paper) large amounts of intuitive explanations within the descriptions may easily turn out to be a problem, because it overloads the reader with information, and because it clutters the description. Using a DET medium this is not necessarily the case, because we may provide for a variety of different ways to explore the descriptions. This will be illustrated in section 4 of this paper, where we will outline a dynamic exploration tool for object-oriented design.

The level of abstraction: As noticed above, the level of abstraction is relatively high during the OOD process as compared with the OOP process. This is both the case for the static models and for the dynamic models. It is tempting and natural to compare dynamic OOD models with models of execution of an object-oriented program. It is very important to settle on dynamic OOD models which are more abstract than the actual OOP execution models. In the next section of the paper we will discuss the problem of abstraction in relation to program execution models.

3 Abstract Execution Models

As discussed above we distinguish between the *programs* and *program executions*. These characterize the *static aspects* and the *dynamic aspects* of program development respectively. In this paper we discuss a similar distinction in the area of object-oriented design. More specifically in this paper our interest is the dynamic aspects of object-oriented design.

Abstraction is familiar and well-known when we talk about programs. Many key elements in modern programming languages are related to abstraction. Similarly, static design models do in themselves represent abstractions over a lot of details. On this background we find it natural to study abstraction possibilities over the details in actual and concrete program executions. In the object-oriented paradigm the program execution elements are objects and messages.[1] The purpose of this section is to develop

[1]'Objects' and 'messages' may, in themselves, be seen as abstraction over 'cells' and 'instructions'. Thus with the object-oriented concepts we have already introduced a certain amount of abstraction in the execution model.

a model which elevates the dynamic behavior of objects from the implementation level to a design level.

At the programming/implementation level, the behavior at the dynamic level is governed and controlled by the description, which we call the program. As such, the actual program execution is in part described and derived from a static model. At the design level, it may be difficult to imagine a similar impact on the dynamic model from the static models. The problem is that we in most static design models have abstracted away from the aspects which are related to execution behavior. Furthermore, we are interested in dynamic design models which are independent of static models, such that dynamic models can be studied before we develop any static model of the software.

We will now analyze abstract program execution models, and describe these in relation to actual program execution models. We focus on the following elements in the actual execution model of an object-oriented program:

- Objects
  - Object fields and types.
  - Object relations to other objects.
  - Object relations to classes.
- Messages.
  - Actual parameters.
  - Results: value or impact on the state of the program.
- Start situation.
- Classes
  - Class context.
  - Class attributes and types.
  - Class relations to other classes.
- Methods.
  - Method context.
  - Formal parameters.
  - Bodies sending messages in certain control patterns.

We see that the actual execution model has been chosen as a clean object-oriented model without surrounding procedural or functional elements as such. Objects and messages are the central elements. In the following we will discuss the abstract execution models in relation the elements of the actual execution model, which we outlined above.

	Formal	**Informal**
Fields and types		
Fields		
States		
Nothing		

Figure 2: *A table of possibilities for descriptions of object fields.*

Objects and object fields: It has to be decided to which degree internal aspects of objects, which carry the object state, are part of the dynamic OOD model. In the most concrete extreme the fields and field types from the actual execution model are also included as elements in the dynamic model (with or without type information). In the most abstract extreme the internal object aspects are abstracted away entirely. In between these, one can imagine that we identify a number of different states of the object. Here, a 'state' is a more abstract notion than the vector of actual object fields. One can also imagine that we represent the object state in terms of the operation history. As discussed in section 2, these possibilities may all be formalized, or they may be kept informal. This design space is illustrated in figure 2.

Object relations: The object relations define a logical structure among the objects. The logical structure among the objects is important in its own right, not least because the object relations may induce similar relations among the classes of objects. Thus, relations among objects may give hints to the structure of the software we are working with. In addition, the relations among objects allow us to access one object O1 from another object O2, provided that O1 can be reached via a relation from O2. *Reachability* may be handled via the object fields, if they are part of the dynamic model. Alternatively, reachability may be handled via an object naming scheme, introduced exclusively for the purpose of the dynamic model. The naming scheme may or may not reflect object visibility in the actual execution model.

Relations between objects and classes: The relations between objects and the classes, of which the objects are instances, are important, because they provide the links between a dynamic model and the static models which are to be developed later on. Thus, in our dynamic models, we find it necessary to have a notion of classes, and to relate an object to its instantiating class.

Messages: It is appealing to use the message passing metaphor for object interaction. Messages cause activations of procedures or functions on objects, and as such the messages are crucial elements of a dynamic model. There are two objects involved in a act of sending a message: a sender and a receiver. At the programming level the receiver is specified in a rather direct way, whereas the sender is identified more indirectly (as "the current object"). In a dynamic object model we have the possibility to chose an explicit or implicit determination of the sender and/or the receiver of a message. The actual parameters are in general expressions, but in the dynamic object model they may be limited to be notions of objects. The parameter passing and parameter correspondence mechanisms are also up to discussion in the model.

In the actual execution model, the impact of a message may be an effect on the program state and/or an object, which is the value returned as result of a message passing. The impact of a message may be important enough to warrant some special attention in the dynamic model. This can be done by accompanying each message with an account on its result. This account may be formal, informal, or both.

The start situation: The start situation designates in the most limited version an object which receives an initiating message. In the dynamic model this may be extended to a set of *pre-existing objects* on which the execution may take off.

Classes: Classes are elements of a program, and as such they belong to the static model, but, as already mentioned, it is important to provide information in the dynamic model which let us bridge the gap between the two kinds of models. For our purposes there seems only to be a little point in injecting a complete static model into the dynamic model. One or more static models should probably be worked out besides the dynamic model. As discussed in section 2, it is the idea of this work that the static models are created after the dynamic model. As a consequence, we should probably only include the classes and the class relations that are necessary to make sense out of the dynamic properties, and which let us "build a bridge" from the dynamic model to a static model.

Because of the duality between the set of object and their relations on the one hand, and the set of classes and their relations on the other hand, it may be possible to automatically derive some properties of the static model from the dynamic model. As an example, the class attributes and their types may be derived from the object fields, provided that the relation among objects and classes is present. Similarly, aggregation and association among classes may be derived from the relations ('containment' and 'reference to') among the objects. In contrast, it is probably more difficult to induce the generalization/specialization relationship among classes from the object relations, because this relationship does not leave much trace in the object space.

Methods: Methods are parts of the program description. In most object-oriented design methods, methods are important elements in the central static models. Like procedures and functions, methods are abstractions of commands or expressions, and as such they play an important role also in a dynamic model. At our level of abstraction we do not care whether methods are shared among objects which are instances of the same class. (This is a distinction which, of obvious pragmatic reasons is important in the actual execution model). Given a message from one object to another, it is a method in the receiver-object which is responsible for the state transition in the dynamic model. Selected aspects of the state transition is one of the key issues in a dynamic model, and

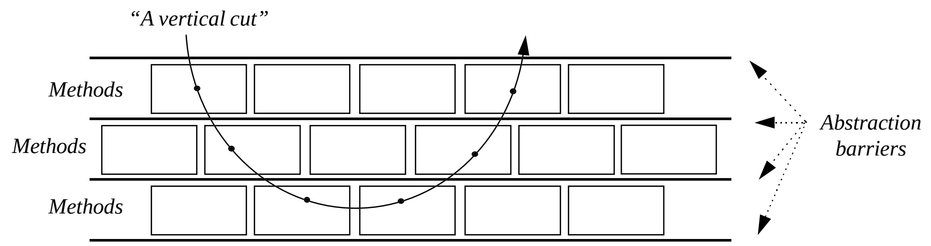


Figure 3: *Layers of methods and vertical cuts of messages.*

consequently the treatment of the methods becomes a crucial theme in our discussion.

In the static model the methods are organized to form "horizontal layers" in relation to the overall program architecture. It is usually recommended only to consider one layer at a time. Thus the methods in a layer can be regarded as black boxes that exclusively are concerned with fulfilling the contractual obligations in respect to their clients (which often are at a higher level). In a dynamic model, where we are concerned with understanding the mutual interplay among objects via messages, it may be the best approach to study "vertical cuts" of methods, which are linked together with message passings (method activations). This is illustrated in figure 3.

As a minimum we want our dynamic model to include object allocation and object mutation aspects. [2] I.e., selected object births and object mutations, as prescribed in a method, should be captured in the dynamic object model. (The level on which to deal with object mutation depends on the level we deal with fields/state of the objects. This has been discussed above). In the one extreme, the description can follow a fixed control course, and as such we will call it a *scenario*. A scenario is an example execution which involves a number of objects and a number of messages. The model builder determines the branch of each selection and the number iterations in each loop in advance, at model definition time. In the other extreme we deal with a prescription which can include the actual variations in the control (selection and iteration), along the lines of the actual program execution; such a prescription will be called a *high level program*. If complete enough, one can imagine that a high level program can be used for a simulation of the program execution at a high abstraction level.

[2]Object deallocation is implicit in most modern run-time systems of object-oriented programming systems. This is the job of a garbage collector. Consequently, object deallocation should not be an issue either in a dynamic design model.

4 Elements of a Dynamic Exploration Tool

In section 2 we introduced the notion of a dynamic exploration tool. A dynamic exploration tool is seen as a medium on which to define, analyze and experiment with a dynamic object model. In this section we will describe the properties of the first version of a dynamic exploration tool. But first we will describe a particular abstract execution model, which we have developed, and on which the tool is based. Again, the abstract execution model is a first version, which is likely to change when we get some experience with it's use in a dynamic exploration tool.

As the starting point we decide that the first version of the dynamic model should be based on scenarios rather than high level programs (see the discussion of these concepts in the previous section). A scenario is a fixed sample execution without any actual dynamics[3]. Chosing the scenario approach gives us more freedom to introduce informal aspects in the model. Such aspects cannot easily be simulated in "automatic execution" of high level programs.

4.1 The abstract execution model

The central concepts in the abstract execution model are *objects*, *messages* and *scenes*. The class concept is also taken into account, but it is less central. The method concept is not part of the model. We will now describe each of the concepts of the model in greater details.

Objects: An object is a abstraction of part of the program state. The abstraction serves an as encapsulation of the state, and each object has its own identity which makes it unique in comparison with all other objects. This reflects the traditional understanding of objects in an OO model. The model is not explicit about fields in an object, but it is possible to declare how new objects are related to existing objects (global to, part of, associated with). In the model, object identity is handled by giving each object a unique name in a flat name space. Hereby each object can be easily referred. As an implication, reachability issues are ignored in the model. The class of each object is registered, but apart from this the class concept is not important in the model.

Objects are born through *object provisions*. Object provisions is a convenient mechanism which establish objects exactly when they are needed. An object provision declares/claims the existence of an object in addition to a certain relationship to other objects. The initial state of the object is described informally. If we need to access a part object P of an object Q, it can be done by 'providing P and as a part of Q'. This compensates for the lack of field declarations in Q, and it allows us to introduce objects and object relations at the point in time, where they are needed in the dynamic course

[3]By 'actual dynamics' we here mean such dynamic behavior which is known from real program execution.

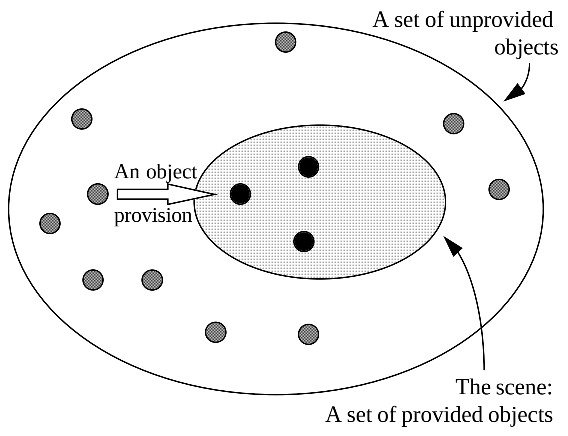

A set of unprovided objects
An object provision
The scene: A set of provided objects

Figure 4: *An object provision brings an object 'onto the scene'.*

of the program. In an actual execution model it works the other way around: When Q is created we should be concerned with the creation of P too. An object provision may also be used to signal that the provided object P becomes interesting/necessary for our purposes at object provision time. In the actual execution model, P may have existed for a long time. The idea is here that an object provision is a somewhat magical introduction of an object "onto the scene". This is illustrated on figure 4.

Messages: Objects interact via message passing. In our current dynamic model only the message receiver is explicitly given. This corresponds to the static description of message passing, in which the sender object is given implicitly as 'the current object'. However, it might be desirable if both the sender and the receiver could be given explicitly. The actual parameters of a message are also taken into account. Actual parameters may either be already provided object, new object provisions, or informally described values. Thus, expressions (here actual parameter expressions) have only limited power in our model.

We focus on "vertical cuts" of message passings in the model. I.e., we directly describe a chain of messages across the boundaries of abstraction barriers. This is shown in figure 5. The figure illustrates that a message M1 is sent from O1 to O2. Next O2 sends messages M2 and M3 to O3 and O4 respectively. Following that O4 sends a message M4 to O5. An additional structure, shown as rectangles on the figure, is introduced among the messages. All the messages in a rectangle belong to the same method. This is the only trace of methods in our model. Thus, when sending the message M1 from O1 to O2, a method is presumably located in O2 which sends messages M2, M3 and M2 (second time) to O3, O4 and O7 respectively.

Scenes: A scene is a pre-existing set of objects which we want to exist as the starting

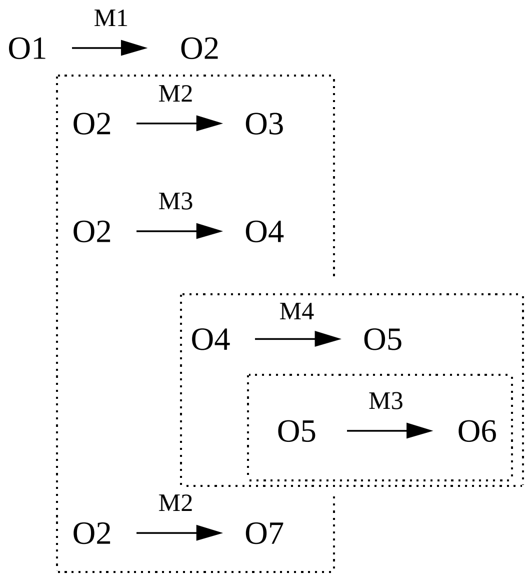


Figure 5: *A sample message passing structure.*

point for one or more scenarios. A scene can be characterized by a set of object-provisions. Object provisions which appear within a scenario add objects onto the scene.

As already mentioned, several aspects are described at an informal level in the dynamic model. Above we have seen that object states and more complicated expressions are not handled formally. Besides this, a number of informal aspects complements the formal aspects, in order to enhance intuitive understanding. Every message, object provision, and result of a message can be (and should be) described intuitively, in pure text. Such descriptions are called *understandings*. The understandings are in addition to the more formalized accounts, which are part of the dynamic model.

4.2 The tool and its parts

The tool that realizes the dynamic medium on which we define dynamic object models can be divided into three conceptually distinct parts:

1. A model builder.
2. A model explorer.
3. A model analyzer.

In an actual elaboration these parts may be more or less integrated.

The **model builder** allows the designer (the user of the tool) to create and modify a dynamic model. Thus, the model builder is some kind of an editor. The means of expression (as discussed in section 3) plays an important role for the model builder. There are basically two different possibilities: A linguistic approach and a 'data structure approach'.

In the *linguistic approach* the dynamic model is formulated as text in a formal language, which later can be processed and interpreted in an appropriate way (we will later in this section come back to what interpretation of dynamic models means).

In the *data structure approach* a dynamic model is represented by an (internal) data structure which may be viewed in different ways on a display medium. The data structure may be directly interpreted without prior parsing. The data structure, which represents the model, is constructed via a *structure editor*.

The 'data structure approach', supported by a structure editor, is preferred. In our case the data structure is a tree which represents objects, messages, object-provisions, actual parameters and understandings (and other concepts) of the dynamic model. One reason for chosing the data structure approach is that it allows for an incremental definition of the various aspects (e.g., formal and informal aspects) in a dynamic model. At first, a scene may constructed which is appropriate for a number of related scenarios. Next, a naked message passing structure may be built on top of the scene. Finally, the scene and the message passing structure may be attributed with informal explanations (called *understandings* in our model). If we use the traditional linguistic approach all of these aspects would be described together, thereby giving us large and unwieldy descriptions that are hard to define and hard to grasp. Moreover, using the data structure approach we may switch between a textual and diagrammatic interaction style. It is clearly the case that while formulating the understandings we should use plain text. However, when defining the message structures, it may be much more convenient to use a grapher tool, in which each object manifests itself as a node from which we can define messages to other nodes as edges.

As the name indicates, the second part of the tool lets the user **explore the dynamic model**. The purpose of 'exploring' is to consolidate the understanding of the dynamic behavior of an object system. To some degree, it may be possible to gain such an understanding by simply *reading* the model (in some linguistic og diagrammatic presentation). The model may also be *traced* in the order of the actual message passings. While tracing, some overview of the objects on the scene may be maintained. During tracing, the explorer (the user) may determine the control course through selections and iterations (if/when they become part of the model). Tracing may be seen as one possible way of *animating* the model.

The purpose of the **model analyzer** is to infer and extract properties of the dynamic model. The analysis results may be used as the starting point of a static model. The analysis results may alternatively be used to enhance the dynamic model itself, e.g.,

```
SCENE
  OBJECT-PROVISION A-POINT: POINT [global]          The initial point of this example.
MESSAGE A-POINT.CIRCLE-ON-POINTS(OBJECT-PROVISION P: POINT
                                 OBJECT-PROVISION Q: POINT )
                                                    Determine a circle through this point and
                                                     the two points passed as parameter.
   OBJECT-PROVISION L1: LINE [local]                The line between this point and P
   OBJECT-PROVISION L2: LINE [local]                The line between this point and Q
   MESSAGE L1.CENTRAL-LINE()                        Find the central line of L2
     OBJECT-PROVISION P1-L1: POINT [part of L1]     The starting point of L1
     OBJECT-PROVISION P2-L1: POINT [part of L1]     The ending point of L1
     OBJECT-PROVISION C1-L1: CIRCLE [local]         Circle with center in P1-L1, radius: length of L1.
     OBJECT-PROVISION C2-L1: CIRCLE [local]         Circle with center in P2-L1, radius: length of L1
     MESSAGE C1-L1.LINE-THROUGH-INTERSECTION-POINTS(C2-L1 )
                                                    We find the line through the two points where
                                                     this circle and C2-L! intersect each other.
     RESULT: OBJECT-PROVISION L-RES: LINE [local]   The line through intersection points of circles.
   RESULT: L-RES , also known as CENTRAL-LINE-1
   MESSAGE L2.CENTRAL-LINE()                        Find the central line of l2.
   RESULT: OBJECT-PROVISION CENTRAL-LINE-2: LINE [local]
                                                    The line perpendicular on L2, on its center.
   MESSAGE CENTRAL-LINE-1.PARALLEL_WITH?(CENTRAL-LINE-2 )
                                                    Check if receiver is parallel with central-line-2
   RESULT: Assume the lines are not parallel.
   MESSAGE CENTRAL-LINE-1.INTERSECTION(CENTRAL-LINE-2 )
                                                    The intersection points between central lines.
   RESULT: OBJECT-PROVISION C: POINT [local]        The point of intersection.
   MESSAGE P.DISTANCE(C )                           Find the distance between the point C and
                                                     one of the original three points.
   RESULT: The distance (a number).
 RESULT: OBJECT-PROVISION C: CIRCLE                 The circle with center in C and radius 'distance'.
```

Figure 6: *A sample traversing of a dynamic model.*

removal of inconsistencies.

4.3 An example

In order to illustrate our approach we will in this section discuss a sample dynamic model, which we have developed with an early prototype of our envisioned dynamic exploration tool.

The problem domain of the example is geometric objects such as points, lines, and circles, on which a number of classical geometric constructions can take place. We imagine that we are about to construct a computer-based system, which support such a construction work. We start the design work by making a number of scenarios, which together constitute our dynamic model. First afterward, and based on this dynamic model, we deal with the static model of the involved classes. The problem, which we will consider here, is the construction of a circle through three given points.

As explained above, we represent a dynamic model as a tree. By traversing this tree we get a *trace* of the messages in the scenario. Figure 6 shows an example of such

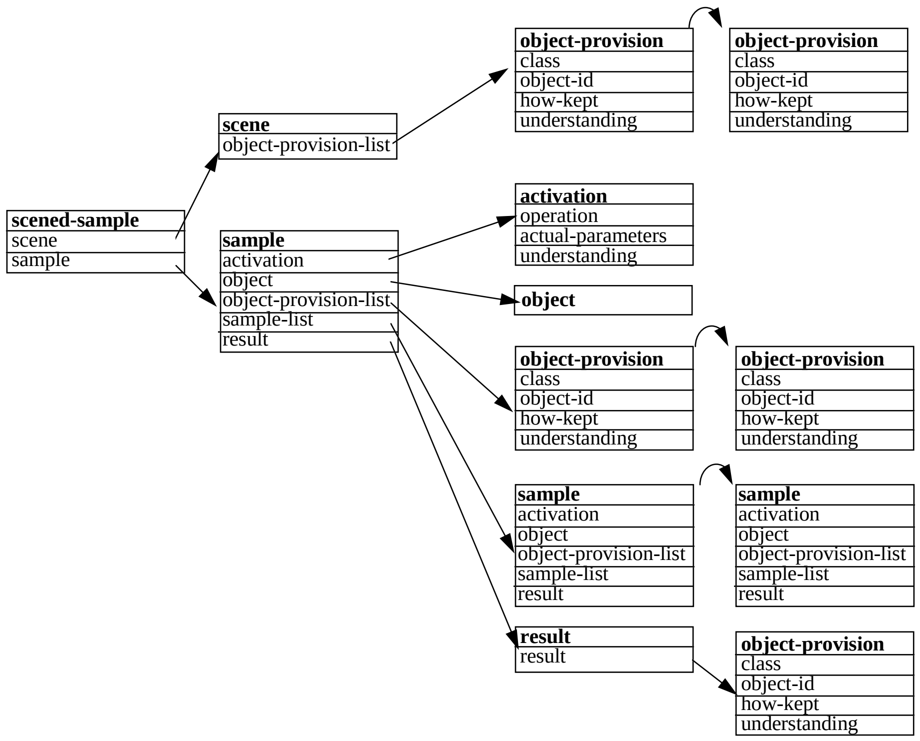


Figure 7: *The data structure behind the trace of figure 6.*

a trace. The explanations of messages and object provisions are also shown on the right-hand side of the figure, in italic font. The example illustrates an object-oriented realization of the circle construction through three given points. We see that the initial scene consists of a single point, called A_POINT. A message circle_on_points is sent to A_POINT with two newly provided points P1 and P2 as parameters. We next provide two lines L1 and L2 through the three points. On each of L1 and L2 we now construct the central lines[4]. The dynamic model includes a trace of the objects and messages which are involved in the creation of the central line to L1. The central line of L1 is constructed by drawing circles in the two extreme points of the line segment. Notice how the start and ending points of L1 are provided as *part* of L1. The two points, where these circles intersect gives the direction of the central line. (The pattern is, of course the same in the construction of the central line to L2, and therefore this line segment is just provided “magically”). Finally, the intersection between the central line of L1 and the central line of L2 gives the center of the resulting circle.

The actual data structures behind the trace is a tree implemented in the Common Lisp

[4]The *central line* of a line segment L is a line segment of the same length as L, perpendicular on L, which intersects L in its midpoint.

Object System (CLOS) [1]. A sketch of such a tree is shown in figure 7.[5] The tree is akin to an abstract syntax tree. In the current, early prototype, the tree is built via a simple custom made structure editor. The structure editor works via a number of specialized commands which augment, change and prune the tree structure. Following each modification the trace is redrawn on outline form.

In the current prototype it is possible to analyze the trees in various ways. The most interesting analysis extracts the list of classes of the dynamic model together with a list of messages, which are activated on instances of the classes. The following is the result of such an analysis for the trace discussed above:

```
((CIRCLE
    LINE-THROUGH-INTERSECTION-POINTS)
 (LINE
    CENTRAL-LINE
    PARALLEL_WITH?
    INTERSECTION)
 (POINT
    CIRCLE-ON-POINTS
    DISTANCE))
```

Such a listing, based on a number of samples, may be seen as a starting point for the construction of the static model.

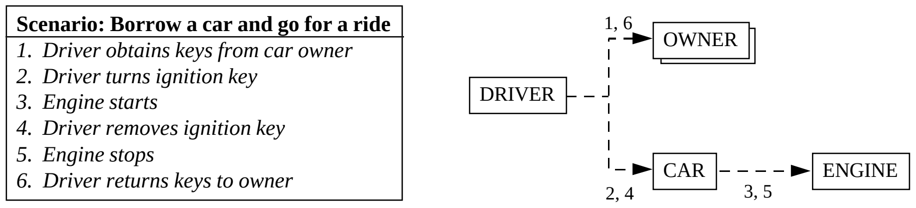
5 Similar work

In this section we will describe and characterize the treatment of dynamic models in some central works on object-oriented design. We will also compare other's work with the work described in this paper.

5.1 The work by Nerson and Waldén.

In the book "Seamless Object-oriented Software Architecture – Analysis and Design of reliable Systems" [4] Nerson and Waldén devote a chapter to description of dynamic models in the BON notation. They first describe a number of "event charts" and other charts (which are just tables with selected relations among elements in a dynamic model). They reject state transition diagrams and finite state machines as the only formalism for description of dynamic models because of the mismatch between static OO models and state transition diagrams. Following that they develop a scenario concept and they introduce some simple diagrams, through which a number of objects

[5] In figure 7 we use the term 'sample' for the nodes which aggregate the information of message passings. Each box in the figure is a node in the tree. The name of the node is shown in bold face. Below the name of the node we show a number of fields, some of which refer to other nodes in the tree.



Scenario: Borrow a car and go for a ride

1. *Driver obtains keys from car owner*
2. *Driver turns ignition key*
3. *Engine starts*
4. *Driver removes ignition key*
5. *Engine stops*
6. *Driver returns keys to owner*

Figure 8: *A dynamic model in the BON notation.*

and 'message relations' (in terms of arrows) among objects are drawn. The temporal aspects are captured by numbers on the message relations.

As an interesting idea, Nerson and Waldén introduce the concept of an object group which can take part in message relations. Object groups simplify the diagrams of dynamic models, and object groups can be seen as abstractions of a number of objects with similar properties. One usage of object groups is indeed to group together objects which are instances of a common, general class. The grouping of objects may be a first step in the direction of classification, and may consequently lead to generalization/specialization in the static model.

In contrast to our approach, Nerson and Waldén's work is quite heavily focussed on the *means of expression*. Their work is primarily a diagraming technique for expressing dynamic models. The *degree of formalization* is rather low in the BON notation. This in also in contrast to our approach, in which formal and informal aspects complement each other. The BON notation is at a slightly higher *level of abstraction* than our proposal. This is mainly due to the generality of the BON message relations, which are (n-n), and due to the object grouping concept.

Figure 8 shows a simple example of a dynamic model formulated in the BON notation. In order to compare this with our approach, we show a trace of our similar representation of the example in figure. We start with the driver and the car on the scene, and we introduce the owner via a provision of a global object at a later time, when the owner becomes necessary for the example. As another difference, we also provide an explicit key object. Notice that it is difficult to deal with object instantiation in the BON notation. The reason is, of course, that a diagram contains a fixed set of object (and a fixed set of message relations). In that sense the dynamic model is static. This is in contrast to our representation of a dynamic model, in which it is possible to prescribe new object creations as a function of time, during the development of the scenario.

The weakness of the BON notation with respect to object creation becomes even more clear if we attempt to make a diagram of the circle construction scenario, which we described in section 4.3. Figure 10 shows an attempt. In this example there are several

```
SCENE:
  OBJECT-PROVISION  C: CAR [global]
  OBJECT-PROVISION  DRIVER: PERSON [global]
MESSAGE   DRIVER.TEST-DRIVE(C )
    OBJECT-PROVISION  OWNER: PERSON [global]
    MESSAGE   OWNER.GET-KEY()
    RESULT: OBJECT-PROVISION  K: KEY [local]
    MESSAGE   C.START(K )
        OBJECT-PROVISION   E: ENGINE [part of C]
        MESSAGE  E.START()
        RESULT: The engine starts
    RESULT: The car starts
    MESSAGE   C.REMOVE-KEY(K )
        MESSAGE   E.STOP()
        RESULT: The engine stops running.
    RESULT: There are now no keys in the car.
    MESSAGE   OWNER.DELIVER-KEY(K )
    RESULT: The driver has no keys any more.
RESULT: The driver is done and happy.
```

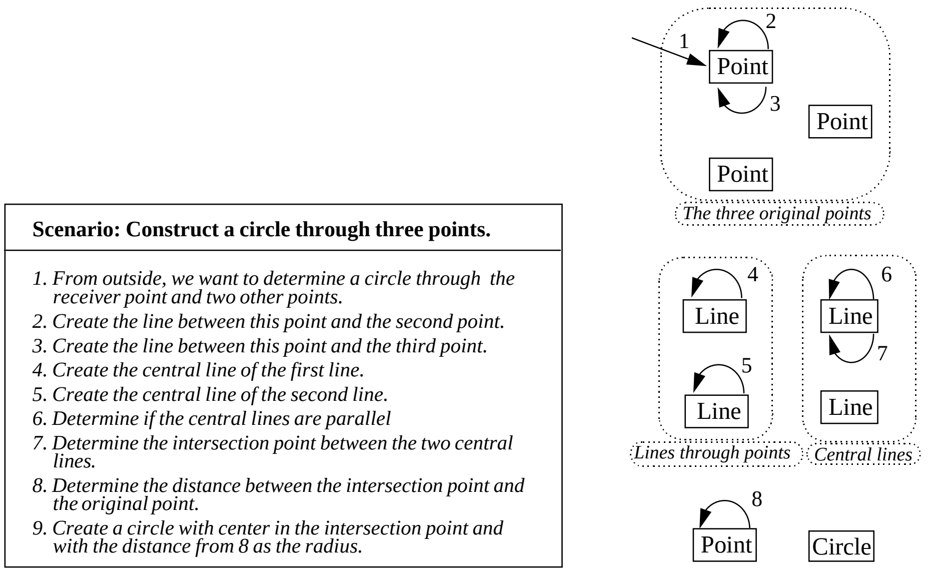
Figure 9: *A traversal of a dynamic model corresponding to the one on figure 8.*

point objects and several line objects involved which all can be seen on the figure. The roles of the different objects are given as group names. As noticed above it is not possible to illustrate how objects are created as the result of messages. This is a severe limitation in this example, because most of the objects are not 'on the scene' at the beginning of the scenario. In addition, there are many objects in this example that never receive any message, with which we are concerned. These objects are parameters to the messages involved. The actual parameters of the messages cannot be shown in the BON notation. In summary we do not think that the diagram in figure 10 gives a good account of the dynamics of the circle construction.

5.2 The work by Jacobson et al.

The OOSE method described in the book *Object-Oriented Software Engineering - A use Case Driven Approach* [3] by Ivar Jacobson et al. has a number of contributions which are relevant for a discussion of dynamic OOD models.

The central concept in the book is *use cases*. A use case is a scenario which involves the computer system and a number actors (users and external computer systems). Thus, a use case describes the borderline between the forthcoming computer system and the



Scenario: Construct a circle through three points.

1. From outside, we want to determine a circle through the receiver point and two other points.
2. Create the line between this point and the second point.
3. Create the line between this point and the third point.
4. Create the central line of the first line.
5. Create the central line of the second line.
6. Determine if the central lines are parallel
7. Determine the intersection point between the two central lines.
8. Determine the distance between the intersection point and the original point.
9. Create a circle with center in the intersection point and with the distance from 8 as the radius.

Figure 10: *A dynamic model in the BON notation.*

surrounding world in terms of the interaction between the system and the actors. A number of use cases, which cover the problem domain, are formulated early in the OOA and OOD process, during a requirement gathering phase. The use cases are stated informally, in plain text and use cases can be specialized/generalized and extended in various ways. In the subsequent design steps the use cases are basis for the majority of the further developments, such as finding the objects (classes) and operations.

In the design phase of the construction process the use cases are turned into *interaction diagrams*, in which objects (called blocks in OOSE) send messages (called stimuli in OOSE) to each other. The temporal relations among the message passings can be seen from the interaction diagrams. In addition the interaction diagrams can be related to pieces of pseudo program, which realizes the interaction.

Figure 11 shows an example of an interaction diagram. The diagram is reproduced directly from (the upper part of) figure 8.11 of [3].

The example is a recycling machine for bottles, cans and crates. When the user of the machine deposits an item (such as a bottle) the machine first checks whether the item is acceptable. If it is, the machines increments a counter which keeps track of this particular kind of item. In addition it increments a counter which keeps track of the total number of items received, again specific to each possible kind of item. All this makes up the basis for printing a receipt to the customer.

The interaction diagram in figure 11 involves four objects: A customer panel (which is

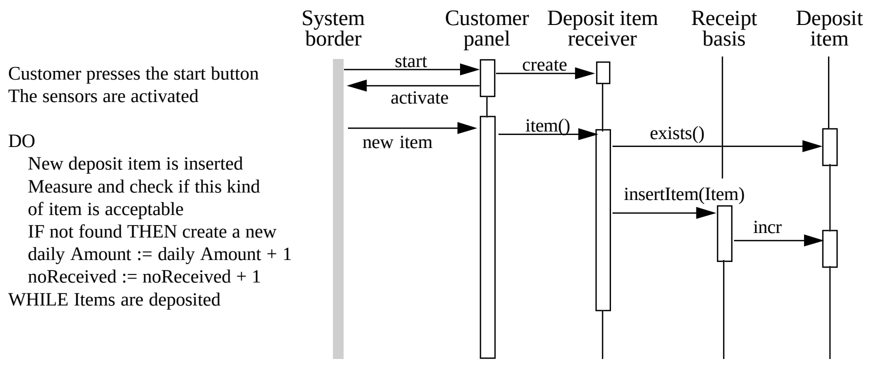

System border
Customer panel
Deposit item receiver
Receipt basis
Deposit item
Customer presses the start button
The sensors are activated
DO
New deposit item is inserted
Measure and check if this kind
of item is acceptable
IF not found THEN create a new
daily Amount := daily Amount + 1
noReceived := noReceived + 1
WHILE Items are deposited
start
activate
create
new item
item()
exists()
insertItem(Item)
incr

Figure 11: *An interaction diagram reproduced from Jacobson et al.*

an interface object in OOSE terms), a Deposit item receiver (which is a control object), a Receipt basis and a Deposit item (both entity objects). Each 'Deposit item' object contains information about (1) the value of the item, (2) the total number received this day, and (3) the physical characteristics of the item. A 'Receipt basis' object is able to count how many items of each kind the customers has deposited. From the diagram and the pseudo code we see that it is first checked whether the deposited item is acceptable. If it is acceptable, it is inserted into the Receipt basis object, and its counter is incremented.

There are several things to notice in the interaction diagram from above:

1. There is no messages shown which cares about acceptance of an item.
2. It is not easy to find out how the individual 'Deposit item' objects are organized in relation to the 'Receipt basis' object.

In addition, it would be awkward to deal with very many different objects in an interaction diagram, simply because of the spatial layout of the diagram.

In the solution which we adopt for this paper there are two central objects:

- An object ACCEPTABLE_ITEMS which allows us to check whether an item is acceptable. This object must contain descriptions of the physical characteristics of each of the allowed items, which the machine recycles. Each of the allowed items are either a BOTTLE, a CAN, or a CRATE. These three concepts are generalized to a DEPOSIT_ITEM.
- Objects of the class ITEM_SET, which counts the number of items of each kind that have been delivered. This object contains a collection of ITEM_COUNTERS, which aggregates an integer counter and a (reference to) a particular DEPOSIT_ITEM.

```
OBJECT-PROVISION SENSORS: SENSOR-SET [global]
OBJECT-PROVISION RECEIPT-PRINTER: PRINTER [global]
OBJECT-PROVISION GRAND-TOTAL: ITEM-SET [global]
OBJECT-PROVISION PANEL: CUSTOMER-PANEL [global]
OBJECT-PROVISION ACCEPTABLE-ITEMS: ITEM-COLLECTION [global]
MESSAGE PANEL.START()
    OBJECT-PROVISION CUSTOMER-TOTAL: ITEM-SET [global]
    MESSAGE SENSORS.ACTIVATE()
    RESULT: The sensors have been activated
RESULT: The systems is started and ready to receive items.
MESSAGE PANEL.DEPOSIT-ITEM(A description of deposited item)
    MESSAGE ACCEPTABLE-ITEMS.DOES-EXIST?(A description of deposited item)
    RESULT: OBJECT-PROVISION ITEM: BOTTLE [local]
    MESSAGE CUSTOMER-TOTAL.EXISTS?(ITEM )
    RESULT: Assume it does not.
    MESSAGE CUSTOMER-TOTAL.INSERT(ITEM )
        OBJECT-PROVISION IC: ITEM-COUNTER [part]
    RESULT: The kind of item has been registered in the customers record.
    MESSAGE CUSTOMER-TOTAL.INCREMENT(ITEM )
    RESULT: counter has been incremented
    MESSAGE GRAND-TOTAL.INCREMENT(ITEM )
    RESULT: counter has been incremented
RESULT: item has been deposited.
```

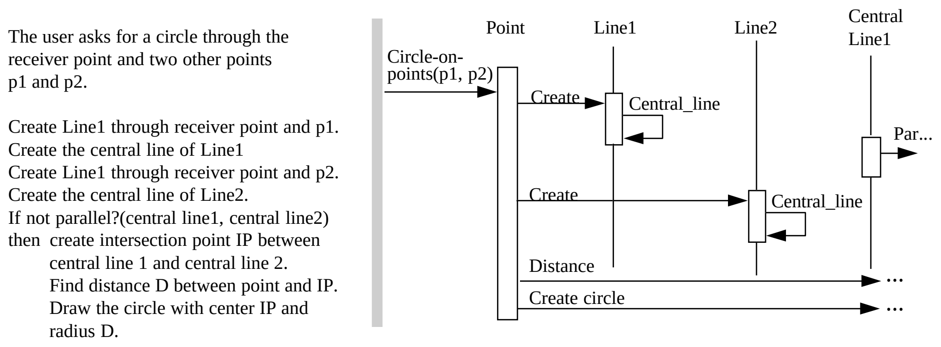
Figure 12: *A trace of the recycling scenario using the concepts from this report.*

Objects of type ITEM_SET can be used to keep track of the individual customer's record (CUSTOMER_TOTAL) as well as to administer the total number of items (GRAND_TOTAL) of each particular kind, which is returned during a day.[6]

In figure 12 we show a trace of our representation of the recycling example from above. Recall that such a trace shows an extract of the information in an underlying tree structure. As can be seen from the figure, five different global objects are provided initially. The interesting messages is 'deposit-item', which takes some description of the deposited item as a parameter. We first check whether the item is acceptable, by sending the message does-exist? to the object ACCEPTABLE_ITEMS. We assume that this operation returns the relevant item object, if the description of the item matches an item object from the acceptable items. In turn, this item is inserted into the CUSTOMER-TOTAL object. Next an item counter in the CUSTOMER-TOTAL object is incremented. Finally the GRAND-TOTAL object is incremented with respect to the item.

In section 5.1 we showed how the circle construction through three given points could be described in Nerson's and Waldén's BON notation. Although the circle construction example probably is far away from the typical problems supposed to be attacked in OOSE, we will also sketch an OOSE interaction diagram for the circle construction problem. The result is shown in figure 13.

[6]In the solution from [3] the daily total of each kind of item is an attribute of 'Deposit item' objects. In our opinion this is not a good modeling, because the number of items received hardly can be considered as a quality of a 'Deposit item'.



```
The user asks for a circle through the
receiver point and two other points
p1 and p2.

Create Line1 through receiver point and p1.
Create the central line of Line1
Create Line1 through receiver point and p2.
Create the central line of Line2.
If not parallel?(central line1, central line2)
then  create intersection point IP between
      central line 1 and central line 2.
      Find distance D between point and IP.
      Draw the circle with center IP and
      radius D.
```

Figure 13: *An interaction diagram for the circle construction problem.*

One of the main difficulties is to illustrate how objects are created dynamically during the course of the interaction. Neither of Line1, Line2 and the central lines exist when Circle-on-points operation is called; they are all created as temporary objects during the execution of the operation. Similarly it is awkward to illustrate how objects are passed as parameters to operations. As a consequence it is difficult to make a good correspondence between the pseudo-program in figure 13 and the interaction diagram proper. Notice that these weaknesses basicly are the same as those identified with the BON notation in section 5.1.

5.3 The work by Booch.

In Booch's book *Object-oriented Analysis and Design with Applications* (2. edition) [2] two different dynamic modeling notations are introduced. The first of these, which is called *object diagrams*, presents a dynamic model as a graph, where the nodes are objects and where the edges are object relations in a broad sense. Following Booch's notation it is possible to adorn both nodes and links with a variety of details. The second notation is *interaction diagrams*. As acknowledged by Booch, the interaction diagrams in the Booch notation are very similar to Jacobsons's interaction diagrams which we described in section 5.2 of this paper. In addition interaction diagram and object diagram carry the same semantics. Consequently, we will limit ourselves to a discussion of Booch's object diagrams in this paper.

Booch introduces Object Diagrams through a simple example, which we have reproduced in figure 14. We will not here explain the details of Booch's notation nor the intuition behind the example; for that purpose the reader should consult section 5.4 of [2]. From the figure it it appears that a PlanAnalyst object sends a message timeToHarvest to the PlanMetrics "utility class" (a collection of procedures) which in turn activates GardeningPlan object. The GardeningPlan object is seen to be a parameter

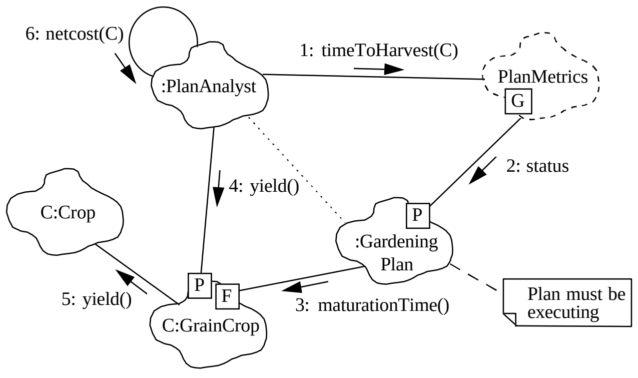


Figure 14: *An object diagram reproduced from Booch.*

of the PlanMetrics procedure (because of the P symbol in the GardeningPlan object). Further on, the GardeningPlan object sends the message maturationTime to the C object (of type GrainCrop). The latter is a part of the GardeningPlan object because of the F symbol in C. The temporal sequencing of the messages is described by the numbers in front of each message.

A number of additional graphical means associated with the nodes and edges make it possible to describe such aspects as (1) the returned object from an operation, (2) roles of the object associations, (3) data flow, (4) syncronization among objects, and (5) informal 'development notes'. It is not intended that all these aspects should be present on all objects or object relations in an object diagram. Still, it is our evaluation that one can easily end up with complicated and cluttered diagrams.

When dealing with object diagrams from Booch, the designer's attention is attracted towards the graphical means of expressions in the diagrams. The degree of formalization is relatively high, compared with Jacobson and the BON notation, not least because it is required that the object diagram should be consistent with a preexisting static model (a class diagram). In itself the diagrammatic style of presentation appeals to the reader's intuition: It is possible to see a number of objects that communicate via the established object relations. In addition, a number of qualities of the objects and the object communications can be read directly from the applied adornments.

As a weakness of the approach behind the Booch object diagram it can be noticed that there is a limit to the number of aspects of a dynamic model that can be put into a graph presentation, as adornments. The limit is probably already reached in the proposed notation. If it is attempted to add elements of explanations (along the lines of the so-called development note in figure 14), the diagram would be totally overloaded with information.

In figure 15 we show a trace of the example from figure 14 in the notation adopted for

```
OBJECT-PROVISION PA: PLANANALYST [global]
OBJECT-PROVISION DUMMY: PLANMETRICS [global]
OBJECT-PROVISION GP: GARDENINGPLAN [global]
OBJECT-PROVISION C: GRAINCROP [part of GP]
MESSAGE PA.START(C)
 MESSAGE DUMMY.TIMETOHARVEST(C)
  MESSAGE GP.STATUS()
   MESSAGE C.MATURATIONTIME()
   RESULT: Here the result is explained.
  RESULT: Here the result is explained.
 RESULT: Here the result is explained.
 MESSAGE C.YIELD()
  MESSAGE C.YIELD()
  RESULT: Here the result is explained.
 RESULT: Here the result is explained.
 MESSAGE PA.NETCOST(C)
 RESULT: Here the result is explained.
RESULT: Net result can be described here.
```

Figure 15: *A trace of the scenario from figure 14.*

this report. We are mainly interested to show the structure of a scenario trace of the graph in figure 14. (The details, especially the intuitive explanations, are only standard strings). We provide the four[7] different objects as being on an initial scene. (There is no need for objects that show up during the scenario). Notice that we are able to express that the C object is part of the GP object, see line four of the trace in figure 15. This corresponds to the small F symbol in the GrainCrop object in figure 14.

In our approach to making dynamic models, we are able to deal with the problem of 'information overloading', because we focus on an internal representation instead of just means of digrammatical presentation. A trace is only one out of many possible presentations, which may select information from the internal representation and somehow present that information for the designer.

Using Booch's object diagrams it is not possible to capture creation of new objects (nor destruction of existing objects). In that respect, object diagrams are similar to the other dynamic models we have found in the literature and described in this paper. It is, however, interesting to observe that Rumbaugh has introduced a notation called *object interaction diagrams* for the 2nd-generation OMT method (JOOP paper) which to some degree is able to capture the creation of new objects in the notation. In many respects, Rumbaugh's object interaction diagrams are similar in style to Booch's object diagrams, although different graphical conventions are used. What is important here is that Rumbaugh propose to use colors (or similar means) in order to distinguish existing objects, new objects, and disposed objects from each other. This may be useful in some limited scenarios, but it is difficult to imagine that colors can be used to give a full

[7]There seems to be five different objects in the graph of figure 14. However, the C object is shown twice, namely in terms of the Crop part of the object and in terms of the GrainCrop part of the object. The class Crop is a superclass of the class GrainCrop. In our opinion, this is an unfortunate duplication.

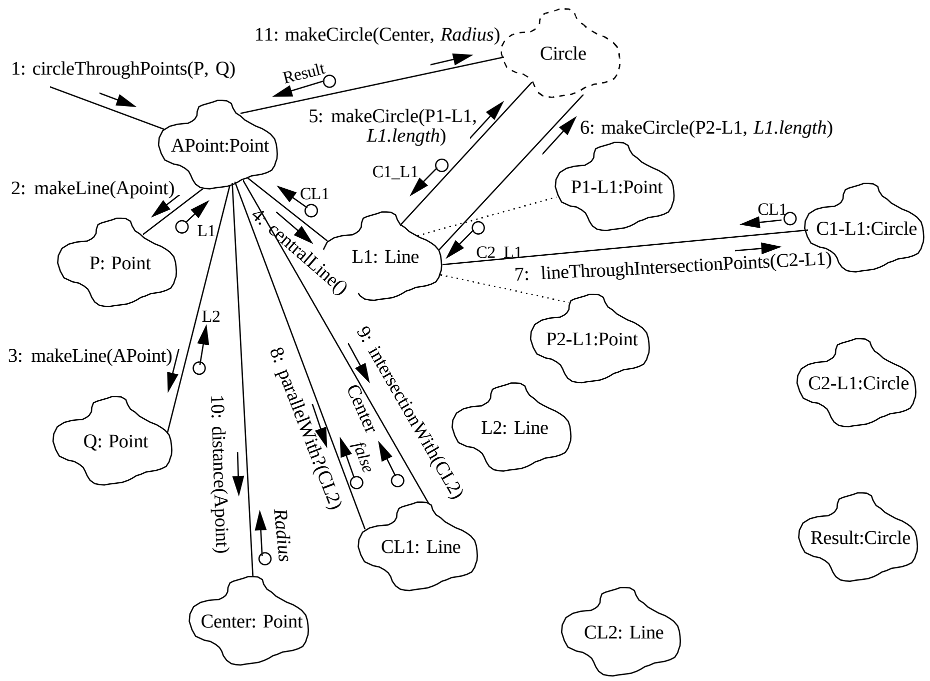


Figure 16: *Circle construction through three given points using a Booch object diagram.*

account on the temporal aspects of dynamic models.

As we have done for both the BON notation and for the Jacobsons's interaction diagrams, we will finally in this section show how the circle construction problem can be expressed in a Booch object diagram, see figure 16.

In step 1, the message circleThroughPoints is sent to APoint. In step 2 and 3 we construct two line segments from APoint to P, and from Apoint to Q. We see that these messages return the lines L1 and L2. In step 4 the central line of L1 is found. The resulting object is called CL1. Step 5 through 7 show the different step that are necessary for the construction of CL1. It might be attractive to modularize the scenario such that the details involved in the construction of a central line is shown in its own scenario. We also need the central line CL2 of L2, but the messages that are involved in constructing it are not shown on figure 16. In step 8 we test whether the two central lines are parallel. As it appears, we assume in the shown scenario that they are not. In step 9 we find the center point of the desired circle as the intersection of CL1 and CL2, and we find the distance from the center point to one of the original points (step 10). Finally we construct the circle in step 11 (by sending a message to the class Circle).

By using the dataflow symbol to indicate returned values and returned objects from messages, it is possible to indicate which objects are created by which message. However, all the necessary details make the object diagram quite difficult to read (and difficult to produce and maintain). And as it has been the case for all static presentations of dynamic models, which we have exercised in this paper, the actual dynamics of object creation cannot easily be captured. I.e., all the objects are present in the figure, although the majority of the objects actually appear during the construction process (and disappear again, because they are only temporary objects, local to some of the methods involved in the construction task).

6 Status and conclusions

In this paper we have discussed dynamic models in relation to static models, as they are used during an object-oriented design process. It is a central claim of the paper that the dynamic model should be constructed before the static model, especially in complicated design situations. Whether a design situation is complicated depends on the skills of the designer. The main purpose of our research is to find out to which degree the claim is true.

A design model can be characterized with respect to

1. the means of expressions
2. the degree of formalization, and
3. the level of abstraction.

Our proposal for a dynamic OOD model avoids a fixed stand on the first characteristics, because we chose an abstract, internal representation which can be presented in many different ways, using several different means of expressions. We go for a dynamic model with both formal and informal aspects. In several respects, the informal and formal aspects describe the same elements. Finally, the level of abstraction is modest in our proposal. The starting point of the model was the actual program execution model, on which we abstracted irrelevant details away. Consequently, our dynamic model has similarities with an actual execution of an object-oriented program.

We do not think it is possible to make satisfactory descriptions of dynamic OOD models as diagrams on sheets of paper. Therefore we have proposed to deal with dynamic models on a more dynamic medium. The medium we have described is a *dynamic exploration tool*. With the tool we have in mind, it is possible to build, explore, and analyze a dynamic model.

Currently we have implemented a very simple prototype of a dynamic exploration tool in CLOS. We are also in the process of implementing a graphical user interface to the tool. We plan to make an additional and improved prototype in order to gain more experience, and in order to test the hypothesis, which have been formulated in this paper.

References

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonay E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification X3J13 document 88-002R. *Sigplan Notices*, 23(Special Issue), September 1988.

[2] Grady Booch. *Object-oriented analysis and design with applications, second edition.* The Benjamin/Cummings Publishing Company Inc., 1994.

[3] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley Publishing Company and ACM Press, 1992.

[4] Kim Walden and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems.* Prentice Hall, 1995.