10

A user-centered method for the development of data-intensive dialogue systems: An object-oriented approach

B. Schewe and K.-D. Schewe
Kampengrund 20b, D-22359 Hamburg, Germany
Technical University of Clausthal, C. S. Instiitute,
Erzstr. 1, D-38678 Clausthal-Zellerfeld, Germany
E-mail: schewe@informatik.tu-clausthal.de

Abstract

Reactive information systems for highly qualified users require their participation in the development process in order to ensure optimal support of their work. In this paper we present a user-centered development method for data-intensive dialogue systems. In this case the system itself must offer a high grade of flexibility in the use of dialogues.

The core of the method is built by an integrated object oriented data and dialogue model. The datamodel follows the fundamental distinction between objects and values and allows an almost natural representation of application entities and their behaviour. At any time a database may be considered a collection of such objects.

The dialogue model follows the same principle in that system states correspond to collections of active dialogue objects. It turns out that the abstraction mechanisms of the datamodel such as classification, inheritance and referential aggregation can be adopted. The smooth integration with the datamodel is established by the means of views.

The models support the participation requirement in that incomplete data and dialogue descriptions can be handled and refined.

Keywords

Object-oriented datamodel, object-oriented dialogue-model, software development method, user-centered design

1 Different views of information systems and implied methodological requirements

During research, development and practical use of Software Engineering Concepts (SE-concepts) several different approaches arose depending on the specific goal of the software to be developped.

Viewing a SE-concept as an integrated concept consisting of a software development model, methods, tools and a project model, we state for example the following features:

- The development of software for clerical work with clients requires concepts different from those required for software for highly automated industrial work.
- SE-concepts for real-time software require a more complex type system and a more sophisticated dynamic model than those for business software.

E. D. Falkenberg et al. (eds.), *Information System Concepts*

- The development of applications with large amounts of complex persistent data requires a more powerful data model than the development of systems without or with very little persistent data.

In this paper we look at a specific kind of software. For this we present two highly integrated object oriented models, a datamodel [ST93, SST94] and a dialoguemodel [Sch]. Due to lack of space we are not able to present refinement steps which show the modeling of data-intensive dialogue systems in practice, see [Sch]. The methods fit well into evolutionary SE development strategies that understand a design rather as a continuous user-driven process than as a sequence of phases [FRS89].

1.1 Requirements from the work to be supported: The user's perspective

The work reported in this paper originates form a project which aimed at the design of software for client-centered clerical work in agencies of a health insurance company. The work is determined by the service for the clients, who

- behave one different from another;
- demand for optimal service and information without delay;
- address their demands to the agents either personally, by phone or by fax;
- appreciate not to be burdened with complicated terminology and forms;

just to mention a few characteristics.

Work in direct contact with such clients can not be pressed into a monotone scheme. However, even behind the direct client contact the work remains unregularly forcing the user to diverge from regular workflow by examining additional circumstances, writing specialized letters instead of forms, escaping automated processes or interrupting processes to gain advice of colleagues or external experts.

For the supporting software this implies, that it has to be composed of several independently usable elements and to leave to the user the decision which one to use in a concrete situation. The dialogue system has to offer many quickly reachable dialogue elements, but must not force the user to a specific way to reach them. Furthermore, it must offer possibilities to store incomplete and unstructured data and must not force the user to grasp all information to a process before recording its data. Finally, it must offer a good overview about a client's situation as context to the special data to be actually processed. For the development process this means:

- that the users of the system have to participate in the development because they are the only ones who can judge whether a solution is usable or not;
- that the software development process has to be an evolutionary one, because problems often become evident only by use of software and ideas for solutions arise when trying a first cut; and
- that the development process is a learning process. At the beginning of a project it is impossible to give a complete list of requirements. As requirements are unknown it is impossible to analyze them first and to design software in the next phase.

For a more detailed view to software development with users see [FRS89]. Of course, the dialogues must satisfy ergonomic quality criteria [IBM91]. Here we abstain from a detailed discussion of this issue.

1.2 The database perspective

A second perspective independent from the user's view of the software to be built are the general requirements concerning the quality of the database:

- It should be free of redundancies except for those which are introduced in physical design to achieve a highly increased performance;
- it should be flexible with respect to future extensions;
- it should not be limited to a specific application, but should be suitable for any.

These requirements make database design a central task in the software development process of data-intensive applications. As a consequence, the information on the data to be processed in the dialogue system has to be simultaneously used for conceptual database design which then has to follow a stepwise approach. In particular, there will never be a starting point with the complete conceptual information available, but the design process will be incremental. This requires partial incomplete data representations to be supported by the used conceptual datamodel. This view has also been emphasized in [ST94].

1.3 Consequences for the SE process model and the development methods

The user's perspective and the database perspective lead to a flexible software design model. There are several ways to proceed and the right way depends on the concrete situation of a project, see [Sch].

It is emphasized that this model works only for little units. One working process after the other should be investigated. It is quite normal that later investigated processes lead to changes in solutions for those already 'finished'.

The user's and the database perspective lead to the following requirements to the design methods:

- The used methods must support the design of small units. This means that dialogue objects can be designed independently from a dialogue flow and that dialogue-model and datamodel allow incomplete, highly parametrized schemata which are refined when the next unit is designed.
- Refinement steps of the datamodel are issued by already designed dialogues.
- It must offer a dialogue-model which allows standardization without enforcing this work to be done *before* the design of the individual dialogue classes.
- Both models must support the use of user's terminology.
- The method must not enforce any refinement step or any element of specification which does not arise from the subject of the application itself.
- The method must be simple in the sense that its application enables at its best a natural representation of the user's conceptual world.
- It must be integrated in the sense that no work is done twice and no system element is repeated in different models.

1.4 Overview of the integrated object oriented design method

As to dialogue systems we think that the object oriented approach, which means that the user first selects an object and then applies operations on it, is commonly accepted as the best support for the clerical work described in this paper. We believe that the requirements

of a user-centered software development method for data-intensive dialogue systems are best fitted by using an integrated object oriented datamodel and dialogue-model, where the selectable objects of the dialogue system are represented as objects in the database in a quite natural way. (This approach to object-orientation is quite different from those which focus on design methods for systems, which will be realized using object oriented programming concepts [RBP+91, CY91].)

In an object oriented datamodel classes describe the structure of objects to be stored in a database. Classes gather operations (often called methods, a notion that is avoided here because of the double sense of that word) to be performed on objects. The datamodel will be described in section 3.

An object oriented dialogue-model supports the object-action-principle by decomposing a dialogue system into dialogue classes with data and actions. The connection to the datamodel is realized by the means of views. The dialogue model will be described in section 4.

Both models use data types to describe the domains of the values of objects. A simple type system for that purpose is presented in section 2.

1.5 Related work

An integration of a dialogue-model and the ER-datamodel has been investigated in the cooperative project TASK [IZ93, papers of Weisbecker, Beck and Janssen], [JWZ93]. Starting with a view of an ER-schema (which contains not only elementary attributes but also derived ones) the authors generate a user interface using a knowledge base of ergonomic rules. From a practical point of view this approach is only useful if it is applied iteratively, because it cannot be assumed to have a schema before starting the dialogue design. It should be possible to start with the dialogue design and to extract the schema from the data used in a dialogue class.

The TASK project lays emphasis on the investigation of the workflow and consequently on the dialogue flow. This approach is not very useful in the case of data-intensive applications where the dialogue flow has to be flexible and cannot be standardized.

There are several other knowledge based approaches which derive a dialogue design from a given complete data schema [SLN93, Bal93]. Hence the criticism to TASK holds on for these tools.

2 A SIMPLE TYPE SYSTEM

Types are used to describe immutable sets of values with (type-)operations predefined on them. Type systems are prescriptions for the syntax and semantics of permitted type definitions. In our methodological framework types occur within the datamodel and the dialoguemodel.

We follow the classical view in [Mit90] using a type system that consists of some *basic types*, *type constructors* and a *subtyping* relation. Moreover, *recursive types*, i.e. types defined by equations, and *predicative types*, i.e. types defined by restricting formulae, are included.

The *base types* used here are $BOOL$, NAT, INT, $FLOAT$, $STRING$, ID or $\perp$, where ID is an abstract identifier type without any non-trivial supertype and $\perp$ is the trivial

type that is a supertype of every type.

The *type constructors* used here are $e_1 \mid \cdots \mid e_n$ (enumeration), $(a_1 : \alpha_1, \ldots, a_n : \alpha_n)$ (record), $\{\alpha\}$ (finite set), $[\alpha]$ (list), $\langle\alpha\rangle$ (bag) or $(a : \alpha) \cup (b : \beta)$ (union), where $\alpha_1, \ldots, \alpha_n, \alpha, \beta$ are already defined types, $e_1, \ldots, e_n$ are constant values and $a_1, \ldots, a_n, a, b$ are field selectors.

We may use base types and constructors to define new types by nesting. If there is no confusion, the field selectors in record or union types may be omitted.

The semantics of such types as sets of values is defined as usual. Moreover, we assume the standard operations on base types and on records, sets, bags, ... We omit the details here. A type T is called *proper* iff the number of its parameters is 0. T is called a *value type* iff there is no occurrence of ID in T. If T' is a proper type occurring in a type T, then there exists a corresponding *occurrence relation* $o : T \times T' \rightarrow BOOL$ with $o(v_1, v_2) = true$ iff v_2 occurs in v_1 at the position indicated by the position of T' in T.

A *subtype function* is a function $T' \rightarrow T$ from a subtype to its supertype ($T' \preceq T$) defined by the usual subtyping rules [Mit90].

Predicative types are used to restrict the set of values given by some type definition to a subset. Formally, a *predicative type* T consists of an underlying type T' and a formula $\mathcal{P}$ with exactly one free variable `self` of type T'. Clearly, the inclusion then gives a subtype function. In order to avoid inflationary use of quantifiers, other variables are also allowed to occur freely in such a formula. They are assumed to be universally quantified.

EXAMPLE 1. We define a type *PERIOD* and a predicative subtype *COURSE* of [*PERIOD*]:

Type *PERIOD* = (begin : *DATE*, end : *DATE* $\cup$ $\perp$)
  Where `self`.end $\neq \perp \Rightarrow$ `self`.begin $\leq$ `self`.end
End *PERIOD*

Type *COURSE* = [*PERIOD*]
  Where `self` = concat(L_1,[P_1,P_2 | L_2]) $\Rightarrow$ P_2.end $\neq \perp \wedge P_2$.end $\leq P_1$.begin
End *COURSE*

L_1 and L_2 are lists with elements of type *PERIOD*, P_1 and P_2 are values of type *PERIOD* and 'concat' is the concatenation of two lists. Informally, the formula requires for any two successive periods the begin date of the first one to be later than the end date of the second one. □

3 THE DATAMODEL

In the object-oriented datamodel (OODM) [ST93] we distinguish between objects and values. Whereas values are common abstractions identified by themselves, objects depend on the particular application context and have to be encoded by object identifiers. In the OODM each object consists of a unique, immutable *identifier*, a set of values of possibly different types, *references* to other objects and *operations* associated with the object.

Values can be grouped into *types* as we have shown in section 2. The *class* concept provides the grouping of objects having the same structure and behaviour. Structurally this uniformly combines aspects of object values and references. Behaviourally, this abstracts

from operations on single objects including their creation and deletion. In the OODM objects usually belong to more than one class.

References between classes give rise to implicit referential constraints. In addition, subclasses (IsA-relationships) require each database instance to satisfy inclusion constraints on object identifiers. As usual in object oriented approaches *class operations* are used to model the database dynamics. In the OODM these are associated with classes.

3.1 The central notion of a class

Each object in a class consists of an identifier, a collection of values, references to other objects and operations. Identifiers can be represented using ID. Values and references can be combined into a representation type, where each occurrence of ID denotes references to some other class. Therefore, we may define the structure of a class using parameterized types. Moreover, classes are arranged in IsA-hierarchies.

More formally, if T is a value type with parameters $\alpha_1, \ldots, \alpha_n$ and if some of the parameters are replaced by pairs $r_i : C_i$ with a reference name r_i and a class name C_i, the resulting expression is called a *structure expression*. Note that a structure expression may still contain parameters.

Then a *class* consists of a class name C, a structure expression S, a set of class names $D_1, \ldots, D_m$ (called *superclasses*) and a set of *operations*. We call r_i the *reference* named r_i from class C to class C_i. The type derived from S by replacing each reference $r_i : C_i$ by the type ID is called the *representation type* T_C of the class C. The type $U_C = (\text{ident} : ID, \text{value} : T_C)$ is called the *class type* of class C.

EXAMPLE 2. Let us consider a class INSURANT for an insurance application.

```
Class INSURANT =
   Structure (insurance_number: NAT, name: NAME, address: ADDRESS,
      course of insurance: [ ( kind : "self", begin : DATE,
            end : (date: DATE, reason: STRING) ∪ ⊥ ) ∪
         (kind : "fam", begin : DATE, end : DATE ∪ ⊥,
            self : SELF INSURANT, relation: "child" | "spouse") ])
   Operation ...
End INSURANT

Class SELF INSURANT =
   IsA INSURANT
   Structure ( employed by : COMPANY , account no : NAT )
   Operation ...
End SELF INSURANT
```

A period of insurance in this example is of one of two possible kinds: Either the insurant is employed by a company and therefore pays his/her own fee or (s)he is a family member of the insurant without own income. □

3.2 Operations

The OODM distinguishes between visible and hidden operations on classes to emphasize those that can be invoked by the user. However, all operations on a class including the

hidden ones can be accessed by other operations. The justification for such a weak hiding concept is due to two reasons:

- Visible operations serve as a means to specify (nested) transactions. In order to build sequences of database instances we only regard these transactions assuming a linear invocation order on them.
- Hidden operations can be used to handle identifiers. Since these identifiers do not have any meaning to the user, they must not occur within the input or output of a transaction.

Each operation on a class C consists of a *signature* and a *body*. The *signature* consists of an operation name O, a set of input-parameter/type pairs $\iota_i :: T_i$ and a set of output-parameter/type pairs $o_j :: T'_j$. The *body* is recursively built of the following constructs:

- *assignment* $x := E$, where x is the class variable C of type $\{U_C\}$ or a local variable (including the output-parameters), and E is an expression of the same type as x,
- *local variable declaration* `Let` $x :: T$,
- *skip* and *fail*,
- *sequencing* S_1 ; S_2 and *branching* `IF` $\mathcal{P}$ `THEN` S_1 `ELSE` S_2 `ENDIF` ,
- *operation call* $C' :\text{-} O'(\text{in} : E'_1, \ldots, E'_j, \text{out} : x'_1, \ldots, x'_i)$, where O' is an operation on class C' with compatible signature and
- non-deterministic *selection* of values $New.f(x)$, where f is a selector on the representation type of C; *New_Id* selects a new identifier.

An operation O on a class C is called *value-defined* iff all types occurring in its signature are proper value types. As already mentioned we require each visible operation to be value-defined. Subclasses inherit the operations of their superclasses, but overriding is allowed as long as the new operation is a *specialization* of all its corresponding operations in its superclasses, but we dispense with a formal discussion of operational specialization. An example of an operation is given in example 4 in [ST94].

3.3 Database states

A database *schema* $\mathcal{S}$ is given by a finite collection of type and class definitions. It is *closed* iff all types, classes and operations occurring within type definitions, structure definitions and operations are defined in $\mathcal{S}$.

In examples 1 and 2 some of the types in the schema such as *NAME*, *ADDRESS*, *REQUEST-DATA* are undefined, therefore it is not closed. This style of allowing partiality in OODM schemata allows to capture also incomplete information about an application area and is essential for the development method.

At any time, a class represents a finite set of objects. More precisely this is captured by the notion of an *instance* (or database state). For a closed schema $\mathcal{S}$ an *instance* $\mathcal{D}$ assigns to each class C a value $\mathcal{D}(C)$ of type $\{(ident : ID, value : T_C)\}$ such that the following conditions are satisfied:

- For each class C identifiers must be unique.
- The set of identifiers in a subclass C is a subset of the one in the superclass C'. Moreover, if $T_C \preceq T'_C$ with subtype function $f : T_C \to T'_C$, then $(i, v) \in \mathcal{D}(C) \Rightarrow (i, f(v)) \in \mathcal{D}(C')$ holds.

- For each reference r from C to D identifiers j occurring in a value v of an object in C with respect to the occurrence relation o_r, i.e.$(i, v) \in \mathcal{D}(C)$ and $o_r(v, j)$ hold, must occur in $\mathcal{D}(D)$.

Basic update operations, i.e. insertion, deletion and update of a single object into a class C, cannot always be derived in the object-oriented case, because the abstract identifiers have to be hidden from the user. However, in [ST93] it has been shown that for *value-representable* classes these operations are uniquely determined by the schema and consistent with respect to the implicit referential and inclusion constraints.

Value-representability of all classes in a closed schema is implied, if we have a (trivial) *uniqueness constraint* for each class. Such a constraint requires the values of type T_C in the class extension C to be unique.

Finally, the semantics of a closed schema is given by database histories, where a *database history* on a schema $\mathcal{S}$ is a sequence $\mathcal{D}_0, \mathcal{D}_1, \ldots$ of instances such that $\mathcal{D}_0$ is the empty database and each transition from $\mathcal{D}_{i-1}$ to $\mathcal{D}_i$ is due to some visible operation on some class $C \in \mathcal{S}$.

The semantics can be extended for *open* (i.e. not closed) schemata by the use of instantiations. An *instantiation* I is given by a closed schema $\mathcal{S}'$ that results from $\mathcal{S}$ by replacing each parameter T by a value type.

3.4 Views

Roughly spoken a *view* may be regarded as a stored query. In the relational datamodel queries can be expressed by terms in relational algebra. This can be generalized to the OODM using its type system. Then a *query* turns out to be represented by a term t over some type T such that the free variables of t represent the classes. This approach is in accordance with the algebraic approach in [Bee90].

However, things change when object identifiers come into play [Bee93], since now we have to distinguish between queries that result in values and those that result in (collections of) objects. Therefore we distinguish in the OODM between *value queries* and general access expressions. For a value query the type T of the defining term t must be a value type.

This allows terms t to be built which involve only identifiers already existing in the database. Thus, such queries are called *object preserving*. If we want the result of a query to represent 'new' objects, i.e. if we want to have *object generating queries*, we have to apply a mechanism to create new object identifiers. This can be achieved by object creating functions on the type ID with arity $ID \times \ldots \times ID \rightarrow ID$ [ST93].

The idea that a view is a stored query then carries over easily. Thus, a *view* on the schema $\mathcal{S}$ consists of a view name $v \in N_C$ such that there is no class C with this name, a structure expression $S(v)$ containing references to classes in $\mathcal{S}$ or to views on $\mathcal{S}$ and a defining access expression $t(v)$ of type $\{U_v\}$, where T_v is the representation type corresponding to $S(v)$.

The notion of a closed schema carries over to schemata that are extended by views.

EXAMPLE 3. Let us give a sample view on the schema of example 2:

`View` COURSE OF INSURANT =

```
Structure
    [ ( kind : "self", begin : DATE,
        end : (date: DATE, reason: STRING) ∪ ⊥ ,
        fams: { ( id: INSURANT, name: NAME, relation: "child" | "spouse",
            begin : DATE, end : DATE ∪ ⊥ ) } ) ∪
      (kind : "fam", begin : DATE, end : DATE ∪ ⊥,
        self : (id : INSURANT, name: NAME, begin : DATE, end : DATE ∪ ⊥ )) ]
Definition
    { (i,course) | ∃ cou . (i,cou) ∈ INSURANT ∧
        course = [ p ‖ ∃ c ∈ cou.course of insurance .
            p.kind = c.kind ∧ p.begin = c.begin ∧ p.end = c.end ∧
            ( c.kind = "self" ⇒ p.fams = { (j,fam) | ∃ cou' . (j,cou') ∈ INSURANT ∧
                fam.name = cou'.name ∧
                ("fam", fam.begin, fam.end, i, fam.relation) = cou'.course of insurance.first ∧
                p.begin ≤ fam.begin ∧ (p.end ≠ ⊥ ∧ fam.end ≠ ⊥ ⇒ fam.end ≤ p.end )) ∧
            ( c.kind = "fam" ⇒ ∃ (k,cou'') ∈ INSURANT . c.self = k ∧
                p.self.name = cou''.name ∧ p.self.begin ≤ p.begin ∧
                ("self", p.self.begin, p.self.end) ∈ cou''.course of insurance ∧
                (p.self.end ≠ ⊥ ∧ p.end ≠ ⊥ ⇒ p.end ≤ p.self.end )) ] }
End COURSE OF INSURANT
```

This view contains the course of insurance of one concrete insurant. Together with one period of kind 'self' in that course there are also the latest insurance periods of the family members. Together with one period of kind 'fam' there is also the period of the insurant to whose family the related insurant belongs. □

4 THE DIALOGUE-MODEL

The use of an object oriented dialogue system consists of two parts:

- The entry of data or the selection of values in fields on the screen and
- the invocation of actions.

When entering data or selecting values the dialogue systems react by offering other data or by activating and deactivating entries in selection lists or possible actions in the action bar [IBM91]. Thus data entries and selections have no effects on the database but serve to collect data and to change them. We call such a collection of data and possible actions a *dialogue object* (d-object). In graphical user interfaces d-objects are normally presented in a window.

Users invoke actions to change the database, to navigate to a new dialogue object or to another presentation of the same dialogue object. Depending on selections or entries made in a d-object only a part of the possible actions are valid. The processing of an action may require further preconditions depending on the state of the dialogue system especially on other user's d-objects.

The handling of both parts of a dialogue system is best performed using a User Interface Management System (UIMS). Such a system provides (among other features)

- windows and operations to open and close them, to move them on the screen, to scroll, to change their size etc.;
- several representations of data, such as selection lists or buttons, text entry fields etc.;
- a main menu where all dialogues start, often called the operation desk.

In this paper we assume the use of such a system and concentrate on the design of the dialogue system although there are also hints to the functionality a UIMS should offer.

A *dialogue system* consists of selection classes, of dialogue classes (d-classes) and of dialogue boxes (d-boxes) which are described in detail in the following sections.

4.1 Dialogue objects

A *dialog object* consists of:

- an abstract identifier,
- a set of values v_i in fields $F_1, \ldots, F_n$ provided for them,
- a set of actions (may be grouped together to menus) to change the data and to control the dialogue and
- a state with the values 'active' and 'inactive'.

This means, that dialogue objects are not persistent but exist as long as the dialogue object is visible on the screen. If a window is closed the corresponding dialogue object ist deleted.

The identifier serves the UIMS to administrate the dialogue objects. It is not known to the user, cannot be used by him and is not visible. Only the active d-object allows manipulations of the represented data and only its actions can be invoked.

EXAMPLE 4. To present the following d-object (figure below) the user selects the insurant named 'Luise Neumann' from a list of insurants and invokes the action 'Course of the Insurance'.
Explanation of values and fields:

- The 'insurant information part (IIP)' is part of most d-objects and gives an overview about the insurant. It is presented in example 7.
- Besides the IIP the d-object contains a list of insurance periods. Each period is represented by a group of lines of which the first line contains the kind (self or as family member of another insurant), the begin and the end of the period. For periods of kind 'self' several lines (maybe 0) follow with names of family members, the relation of the family member to the insurant and begin and end of the latest insurance period of the family member. For periods of kind 'fam' one line follows with the name and the insurance period of the insurant whose family the member belongs to.
- The last line is used for messages.

Explanation of some actions:

- 'History' shows earlier states of the course of the insurance.
- 'System' and 'Options' are pull-down-menus (not shown in the example). 'System' contains e. g. the following actions: New Insurant, Save, Cancel (Esc), Save and Quit (F3), Scroll Forward (Bild↓), Scroll Back (Bild↑), Desk (Strg + F4).

System History Options Windows

Course of the Insurance

1133557 Neumann, Luise 10.11.1948 273
+ more information about the insurant +

Kind	Begin	End	Reason of End / Name	Relation	Begin	End
self	01.04.1979					
			Neumann, Marga	child	13.02.1984	
			Neumann, Horst	child	27.04.1986	
fam	10.11.1976	31.03.1979				
			Meier-Neumann, Fritz		01.01.1975	
self	01.10.1967	09.11.1976	Too old as student			
fam	10.11.1948	30.09.1967				
			Neumann, Wilhelm		01.01.1919	16.08.1990

- 'New insurant' saves the data on the screen and shows the course of insurance of another insurant which can be selected in the list of periods. If no insurant is selected a dialogue box with entry fields for the search for a new insurant is activated.
- 'Save' saves the changes of the data on the screen and shows the same dialogue object again.
- 'Cancel' deletes the dialogue object and returns to the one which was active before respectively to the desk. Changes made to the data are forgotten.
- 'Windows' is a pull-down-menu, containing the list of all existing dialogue objects. It is offered by the UIMS and not described here. □

4.2 Dialogue classes

Dialogue classes serve to group dialogue objects which have the same data and on which the same actions are possible. A *Dialogue class* (d-class) consists of:

- a unique name and title DC,
- one or more selection classes describing the identifying 'types' of the objects to be selected when invoking the d-class and containing an operation for the selection of an adequate object,
- an invoke operation, which generates a new dialogue object,
- a visual value type DT_{DC} which describes the data shown on the screen,
- a content type DT'_{DC}, which is a subtype of the visual type DT_{DC} and consists of the types of all data provided by the invoke operation,
- a set of names $\mathrm{DC}_1, \ldots, \mathrm{DC}_n$ of super-d-classes,
- a set of navigation operations with which the invocation of other d-classes is prepared,
- a set of processing operations with which the state of a d-class can be changed and
- a set of actions, with which operations are called.

EXAMPLE 5. We give a part of the formal definition of the d-class of example 4:

Dialogue class COURSE OF INSURANCE
  IsA IIP
  Selection INSURANT
  Invoke operation *invoke*
  Visual type $DT_{\text{COURSE OF INSURANCE}}$ =
    [(kind: "self", begin: *DATE*, end: (date: *DATE*, reason: *STRING*) $\cup \perp$,
    fams: {(name: *NAME*, relation: "child" | "spouse", begin: *DATE*,
      end: *DATE* $\cup \perp$)}) $\cup$
    (kind: "fam", begin: *DATE*, end: *DATE* $\cup \perp$,
      self: (name: *NAME*, begin: *DATE*, end: *DATE* $\cup \perp$))]
  End $DT_{\text{COURSE OF INSURANCE}}$
  Content type $DT'_{\text{COURSE OF INSURANCE}}$ =
    (id : *ID*, [(kind: "self", begin: *DATE*, end: (date: *DATE*, reason: *STRING*) $\cup \perp$,
    fams: {(id : *ID*, name: *NAME*, relation: "child" | "spouse", begin: *DATE*,
      end: *DATE* $\cup \perp$)}) $\cup$
    (kind: "fam", begin: *DATE*, end: *DATE* $\cup \perp$,
      self: (id : *ID*, name: *NAME*, begin: *DATE*, end: *DATE* $\cup \perp$))]
  End $DT'_{\text{COURSE OF INSURANCE}}$
  Actions History, System.New Insurant, System.Save, System.Cancel, ...
End COURSE OF INSURANCE □

For each d-class there is at least one representation on the screen. Normally there are actions with which the representation of the d-class on the screen can be modified without changing the state of the d-class. The representation of the d-class is given by the UIMS. The concrete description is therefore depending on its functionality. For the layout of d-class representations ergonomic criteria are taken into account.
A *field* consists of:

- a relation to a component of the content type of a d-class;
- field attributes like 'protected' / 'unprotected', 'normal' / 'emphasized', ...;
- the type of the field (text entry field, selection field, ...);
- a selection state with the values 'selected' and 'unselected';
- the information whether data have been entered in a field or not;
- the information where the cursor is placed; and
- an optional name of the field.

Fields may be grouped to other fields. Further properties of fields are possible depending on the features of the UIMS. For each field there is at least one representation on the screen comprising a declaration of its length, its style of emphasis and its style of representation of protection. For each representation there is also a representation of the selection state of the field.

Besides of d-classes which are invoked by the user there are *dialogue boxes*, in which data can be entered and processed. Dialogue boxes are called by operations of d-classes, if further data are needed to finish a running operation. Dialogue boxes have no selection class. As they are not invoked by the user he/she need not select an object before invoking

them. Dialogue boxes have no processing operations that means they do not change the data in the database. The data are changed only when finishing the operation called by the user.

4.3 Selection classes

Objects of which the data are given by a d-class and which are manipulated with its actions have to be selected by the user before invoking the d-class. For this purpose *selection classes* are defined. A selection class consists of a name, a structure expression and a selection operation.

The structure expression defines the types of values with which the selectable objects can be uniquely identified. In an easy case this is a number attribute to which the user is accustomed to, but it also might be a composed type. Selection operations are operations which select objects of a database, i. e. give back at least the abstract identifier of an object in the database.

EXAMPLE 6.

`Selection class` INSURANT
  `Structure`
    (Isn: *NAT*) ∪ (name: *NAME*, date of birth: *DATE*, address: *ADDRESS*)
  `Selection operation` *select* (in: a:: *Ins-selection criteria*, out: i:: *ID*)
`End` INSURANT

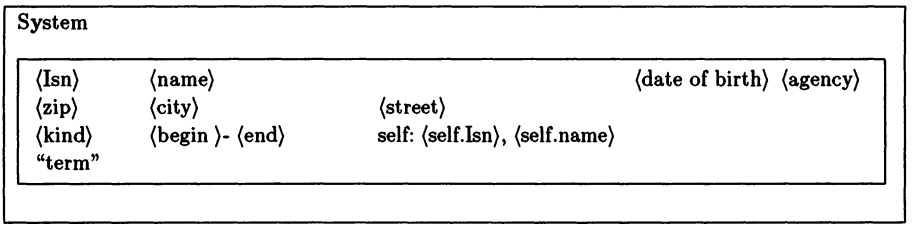
In this example an insurant is identified by his/her insurance number or by his/her name, his/her date of birth and his/her address. This means these data have to be presented to the user to enable him to choose an insurant in a selection list. □

Selection classes are introduced because there normally are several d-classes using the same selection class. There also might be d-classes without a selection class e. g. lists of all objects in a database. Selection classes should be presented by a symbol on the desk of the UIMS. So should be a selected object which the user can use in further dialogue steps.

4.4 Super dialogue classes

Super dialogue classes are used to model reusable parts of d-classes.

EXAMPLE 7. The following figure describes the dialogue class IIP.



Here 'term' means that there are terms of the insurant to be watched out by the employee. □

4.5 Actions and operations

Actions are invoked by the user and call operations on d-classes, on d-boxes or on classes of the schema. Furthermore **delete** is an operation of the UIMS which deletes the acutal dialogue object.
We distinguish four types of operations:

An *invoke operation* assures that the invocation of a d-class is valid and retrieves the data of a selected object from the database.

A *selection operation* is the operation on the selection class, which retrieves an object from the database.

A *navigation operation* provides parameters for the invocation of a new d-class or another presentation of the actual dialogue object.

A *processing operation* keeps the actual dialogue object and acts upon its data. This implies that a processing operation usually changes the database.

Each operation may call operations on classes of the schema. The syntax of operations on the dialogue model corresponds to that of the datamodel. Actions and operations on super-d-classes transmit their operations to their sub-d-classes.

The user uses *actions* to change the data on the screen and to control the dialogue. Actions consist of:

- a name used in the action bar; if necessary names of menus are added to the actions;
- a symbol with which the action is invoked;
- a name or a type of a field which may or must be selected before invoking the action;
- the body of the action, in which operations $o_1, \ldots, o_n$ are called which will run when the action is invoked.

EXAMPLE 8. We give some actions on the d-class COURSE OF INSURANCE of example 5:

Dialogue class COURSE OF INSURANCE
...
Actions
System.New Insurant, YN, self.name $\vee$ fam.name $\vee \perp$
Call *save* in: [(kind, begin, end)], out: -
Call *New Insurant* in: v:: $DT'_{\text{COURSE OF INSURANCE}}$, out: ins :: $ID \cup \perp$
Delete
Invoke COURSE OF INSURANCE **using** ins
System.save, YS, *
Call *save* in: [(kind, begin, end)], out: -
Invoke COURSE OF INSURANCE **using** DT'_{IIP}.id
System.Save and Quit, YQ, *
Call *save* in: [(kind, begin, end)], out: -
Delete
...
...
End COURSE OF INSURANCE

Here '*' means that nothing needs to be selected but anything may be selected.

The operation *save* stores the actual data in the database, but does not affect the dialogue object. The operation *New Insurant* retrieves the data on the screen and determines the identifier of the selected insurant (if so). As the names on the screen are unique and the abstract identifier is part of the content type, this operation is possible. Note that the identifier itself cannot be selected by the user, as it is invisible to him/her. □

An *invoke-operation* consists of:

- the test of an optional precondition;
- a query to the database to get the actual values of the content type of the d-class;
- a preparation of the queried data (sequence, computation of derived values, setting of field attributes, selection of fields, emphasize a standard action ...);
- the positioning of the cursor, the mouse, etc.

EXAMPLE 9. The invoke operation of example 4 is as follows:

COURSE OF INSURANCE-*invoke*
  in: id:: ID, out: course :: $DT'_{\text{COURSE OF INSURANCE}}$
  Invoke IIP in: id, out: iip
  If $\exists$ (id, c) $\in$ COURSE OF INSURANCE **Then** course := c **Endif**
  Put Cursor On first (course [**self**.begin])
End COURSE OF INSURANCE-*invoke*

This operation uses the view COURSE OF INSURANCE of example 3. There is no precondition in this example. □

The example shows the integration of datamodel and dialoguemodel. The invoke operation uses a view on the database. The objects of the d-class with values of the content type are given by a view on the datamodel.

4.6 Dialogue systems

As in the datamodel a *dialogue system* consisting of d-classes, selection classes and dialogue boxes may be incomplete in the design phase. A usable version of a dialogue system is called *closed*. Then we analogously define an *instance* of a closed dialogue system assigning to each d-class a set of d-objects. As we have shown in example 3 the actual set of d-objects at a given time may be defined by a view on the datamodel. For the generalization to open dialogue systems we have to use instantiations.

REFERENCES

[Bal93] Helmut Balzert. Der JANUS-Dialogexperte: Vom Fachkonzept zur Dialogstruktur. *Softwaretechnik-Trends, hrsg. von der Fachgruppe 'Software-Engineering' der GI*, 13(3), August 1993.

[Bee90] Catriel Beeri. A formal approach to object-oriented databases. In *Data and Knowledge Engineering, Vol. 5*, pages 353 – 382. North Holland, 1990.

[Bee93] Catriel Beeri. Some thoughts on the future evolution of object-oriented database concepts. In W. Stucky and A. Oberweis, editors, *Datenbanksysteme in Büro, Technik und Wissenschaft*, Berlin, Heidelberg, Newyork, 1993. Springer Verlag.

[CY91] Peter Coad and Edward Yourdan. *Object-oriented analysis.* Prentice Hall, Englewood-Cliffs, N.J., 1991.

[FRS89] Christiane Floyd, Fanny-Michaela Reisin, and Gerhard Schmidt. STEPS to Software Development with Users. In Carlo Ghezzi and John A. McDermid, editors, *ESEC 89, LNCS 387*, pages 48 – 64, Berlin, Heidelberg, New York, 1989. Springer Verlag.

[IBM91] IBM (International Business Machines Corp.). *Systems Application Architecture Common User Access / Advanced Interface Design Guide*, 1991. Nr. SC34-4290.

[IZ93] Rolf Ilg and Jürgen Ziegler, editors. *Benutzergerechte Softwaregestaltung.* Oldenbourg Verlag, München, Wien, 1993.

[JWZ93] Christian Janssen, Anette Weisbecker, and Jürgen Ziegler. Generating user interfaces from data models and dialogue net specifications. In *Human Factors in Computing Systems (INTERCHI)*, pages 418 – 423, Amsterdam, 1993. ACM.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. von Leeuwen, editor, *The Handbook of Theoretical Computer Science, Vol. B*, pages 365 – 458. Elsevier, 1990.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlane, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Sch] Bettina Schewe. Eine objektorientierte Methode für den benutzergerechten Entwurf datenintensiver Dialoganwendungen. to appear.

[SLN93] Peter Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *Human Factors in Computing Systems (INTERCHI)*, Amsterdam, 1993. ACM.

[SST94] Bettina Schewe, Klaus-Dieter Schewe, and Bernhard Thalheim. Objektorientierter Datenbankentwurf in der Entwicklung datenintensiver Informationssysteme. *Informatik - Forschung und Entwicklung*, 1994.

[ST93] Klaus-Dieter Schewe and Bernhard Thalheim. Fundamental concepts of object oriented databases. *Acta Cybernetica, Szeged*, 11(1/2), 1993.

[ST94] Klaus-Dieter Schewe and Bernhard Thalheim. Principles of object oriented database design. In H. Jaakkola, H. Kangassalo, T. Kitahashi, and A. Márkus, editors, *Information Modelling and Knowledge Bases V*, pages 227 – 242. IOS Press, Amsterdam, 1994.

5 BIOGRAPHY

Bettina Schewe
1974-1982 Student of mathematics und economy at the university of Bonn;
1982-1994 consultant for software engineering and information management, later leader of projects and working groups at an insurance company;
today head of the product development department of a software delopment company.
Dr. Klaus-Dieter Schewe
1976 - 1982 Student of mathematics and computer science at the university of Bonn, 1985 ph. d. at Bonn;
1985-1990 development and research activities in consulting and industrial companies;
since 1990 scientific worker at the (technical) universities of Hamburg, Cottbus and now Clausthal.