Activities: Abstractions for Collective Behavior *

Bent Bruun Kristensen[1] and Daniel C. M. May[2]

[1] Institute for Electronic Systems, Aalborg University
Fredrik Bajers Vej 7, DK-9220 Aalborg Ø, Denmark
e-mail: bbkristensen@iesd.auc.dk

[2] Department of Information Systems, Monash University
Caulfield East, Victoria 3145, Australia
e-mail: dmay@ponderosa.is.monash.edu.au

Abstract. Conventional object-oriented modeling lacks support for representing the interaction between objects in a conceptually intuitive way – often dispersing the logic/control of interplay throughout the objects. We introduce the concept of an *activity* as an abstraction mechanism to model the interplay between objects.
Activities model how our human cognition organizes interaction into units of collective behavior. They are described as classes, allowing interaction to be modeled by such abstraction processes as generalization and aggregation.
At the analysis and design level activities are presented as a general modeling tool for describing the collective behavior of systems of objects. We also discuss how activities can be supported at the implementation level by extending existing language constructs in relation to object-oriented programming languages.

1 Introduction

Objects are a powerful means of modeling entities that exist. But what of the interaction between them? We need effective modeling approaches that will allow us to describe the way objects work together – this need is especially acute in systems where the number and organization of objects becomes increasingly complicated. Where interaction is widely dispersed, discerning the purpose achieved by such interaction becomes difficult.

An abstraction may be used as a tool to help us understand the nature of a problem, as well as describing a possible design solution. Moreover, an abstraction should aid in understanding the purpose which a system accomplishes through the interaction of its sum parts.

Most of all, we need abstractions that are intuitive to our comprehension of problems – offering a modeling approach that is closer to our understanding of how things are organized in the real world.

This paper presents the *activity abstraction mechanism*, that seeks to capture the interplay between groups of objects throughout at different points in

* This research was supported in part by the Danish Natural Science Research Council, No. 9400911.

time. The activity is a modeling and language mechanism that may be used to create *abstractions* relating other objects together – describing not merely their participation in the relationship, but their interplay.

Modeling with Activities. As an illustrating example, we examine the activity of reviewing papers for an upcoming conference – this can be referred to as a `paper_review`. This activity requires a certain degree of interaction/interplay between those who are involved in it. For instance, an `author` will submit a paper for review, while the `chairman` will distribute papers to each `reviewer` who must report back.

There will usually be some sort of specification that describes how the activity should be carried out. For instance, with the `paper_review`, the specification might be carried out in three distinct portions:

```
(a) author submits paper to chairman
    chairman distributes papers to reviewers
    reviewers submit referee reports to chairman
(b) paper_selection
(c) chairman informs authors about result
```

Figure 1 illustrates `paper_review`, its participants and specification (directive).

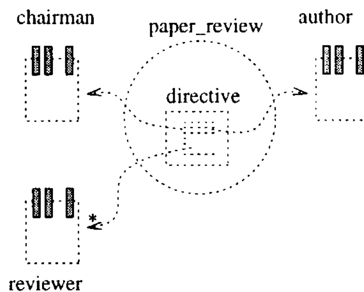

Fig. 1. Graphical illustration of `paper_review`

We should first note that `paper_review` is only one type of `review` that can take place. For example, a `periodical_review` is the review of a submitted article that takes place for a periodical; it is somewhat similar but involves an `editor` rather than a `chairman` and its selection process is different. Both `paper_review` and `periodical_review` are specialized types of `review`.

The directive that specifies how `paper_review` should be carried out may also be seen as a specialization of a more general `review` directive, namely the following:

```
prepare_review_process
carry_out_review_process
complete_review_process
```

Each of these portions correspond to (a), (b) and (c) above (which are more specialized). The participants of these activities may also be similarly classified. For instance, all `review` activities involve a `coordinator` and an `author`. Thus, in a `paper_review`, we can refine a `coordinator` to be a `chairman` – in a `periodical_review`, we can specialize a `coordinator` to become an `editor`.

These different types of `review` activity might have similar methods. For example, producing a `status_report` (produce a listing of the current status of the ongoing reviewing process) is something that each `review` activity must do – a `paper_review` will produce a specialized type of `status_report`, as will a `periodical_review`.

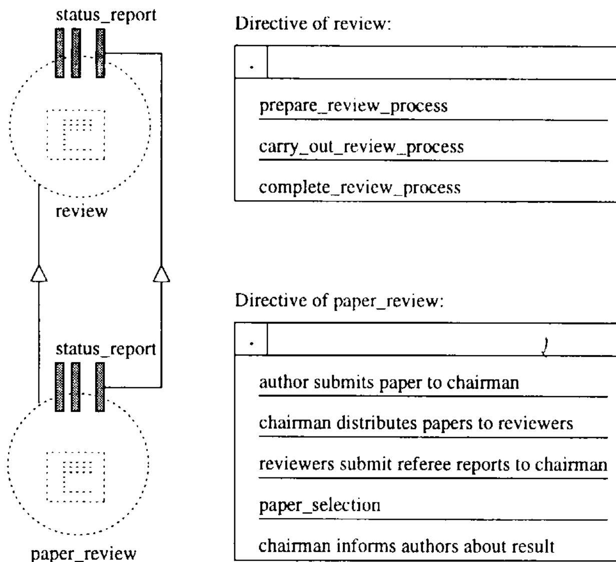

Fig. 2. `paper_review` is a specialization of `review`

Figure 2 illustrates how `paper_review` is specialized from `review`. The directive of `review` is refined in `paper_review`. The figure also notes that the method `status_report` of `paper_review` is a refinement of the same method of `review`.

Finally, it is important to realise that an activity may be constituted from smaller sets of activities (part-activities). For example, within the `paper_review` activity, there is a `paper_selection` activity to choose acceptable papers.

Thus, there are important characteristics of activities that we can note.

Their behavior is *collective*; that is, they can be contained into some logical unit (such as paper_review). Activities may be *composed* of smaller activities (e.g. paper_selection). Principles of generalization/specialization may apply to activities (review) and their participants (coordinator). An activity may also have methods (behaviour) that may be specialized (e.g. produce status_report) – such methods may access and control the state of the ongoing process modelled by the activity. It is worthwhile noting that an activity can have a set of properties or state (e.g. which reviews have been returned?, is the review process complete?).

And there are the entities that will participate in an activity (e.g. author, chairman, reviewers), while a set of directions specifies how the activity will be carried out.

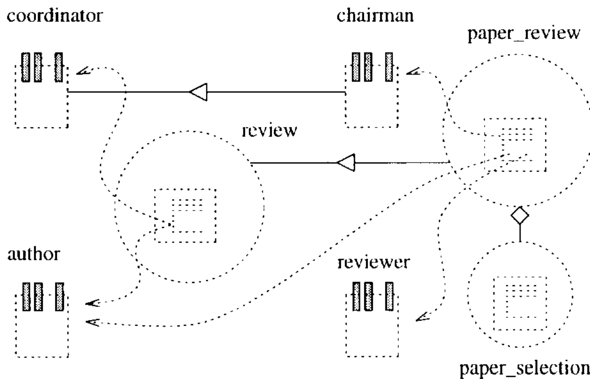

Fig. 3. Relationships in paper_review

Figure 3 shows the specialization and aggregation relationships between activities and participants in the paper_review example.

Collective Behavior is Objectified. An activity can be defined as an interplay between entities over a given time – intuitively, this corresponds to a general everyday understanding of an 'activity'. It is normal to describe our participation in an activity as a coherent unit; we engage in an activity as a single module and describe it to others in the same way (e.g. "They reviewed the papers, then they ate lunch"). The significant feature of viewing activities as units of collective operation is a reduction in complexity.

The activity abstraction mechanism depicts the relationships that links interacting objects. It is more than a mere gathering objects – applying the object-oriented paradigm, this parcel of collective behavior is objectified, resulting in an object that represents a contiguous unit of process. Activities will then possess

properties similar to those of regular objects: they may be aggregated, specialized, recursively defined, etc. Thereby, it is possible to describe the functionality of a system by combining the behavior of different types of activities – in the same way, these activities may be described in terms of the interplay between their participants.

The interplay between the participants, which is described collectively, is quite different to an object- or participant-centric view (refer Fig. 4). Rather than specifying "what is done to whom", a collective description states "who is doing what to whom". In an object-centric description, the 'who' is implicit, whereas the collective description makes it explicit. Therefore, the atomic elements of an activity's directive will usually comprise three things: subject (who), object (whom) and verb (what is done). There is more clarity in such a description – participants are more clearly identified, and the nature of their interplay can be more quickly discerned.

The notion of *time* has been implicit thus far in our discussion of the activity abstraction. It is a key characteristic of activities that they are *temporal* in nature; an activity such as `paper_review` has inception, execution and termination phases. We emphasize that this is similar to the everyday activities that we encounter which possess a finite lifetime.

The primary benefit of characterizing the behavior of a software system as comprising activities is that it models our human approach to reducing complexity in how we handle everyday tasks – in the same way that our cognition clusters information to enhance comprehension, the activity abstraction seeks to resolve complexity by clustering the interplay between objects.

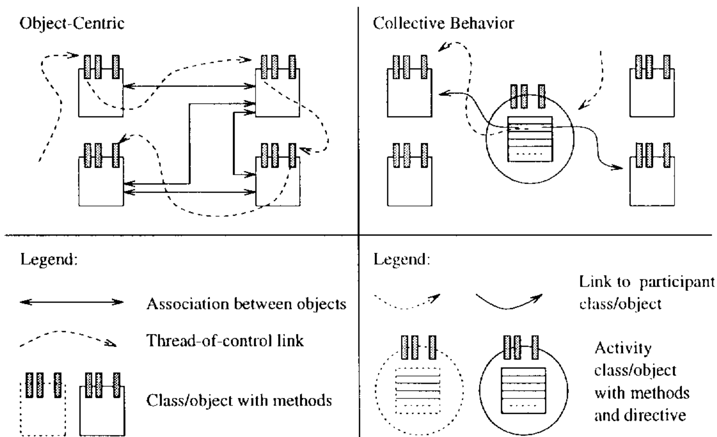

Fig. 4. Object-Centric and Collective Behavior

In Fig. 4, we illustrate two alternative forms of execution sequence – the

usual object-centric method invocation where a method of one object invokes a method of another, etc (a recursive sequence of method invocations); and an alternative scheme employing collective behavior, where the activity prescribes a sequence of interplays each including a caller and a method call for the (called) object.

Of course, there is a functional flavour to activities. It is critical to note that while purely functional modelling is insufficient to cope with describing systems, its style of examining process and functionality appeals to us on an intuitive level. Instead of advocating a binary 'either/or' approach to modelling, it is possible to combine the functional perspective with object-oriented concepts. This may result in a way of looking at the world that is more natural and powerful than either individual perspective affords.

Paper Organization. In Sect. 2 we discuss the use of activities for modeling in the analysis design processes. We use "card games" as a concrete example, and we discuss the fundamental characteristics of activities in general terms. In Sect. 3 we discuss the implementation of activities. We discuss various proposals for special language constructs for both direct support and simulation of activities in object-oriented programming languages. In Sect. 4 we review theoretical and practical experiments with activities. In Sect. 5 we summarize the proposals and the results of the paper, compare them with related work (frameworks/design patterns in software architecture and notation/language mechanisms in object-oriented analysis, design, and implementation), and discuss further challenges.

2 Designing with Activities

We shall use a card game as a concrete example to illustrate the use of activities in analysis and design. Through this example, we demonstrate and discuss the fundamental characteristics of activities. Activities as abstractions in object-oriented analysis and design is originally introduced in (Kristensen 93a).

Card Game Example. Our intuitive understanding of a card game is that it is a human activity – it involves a specific kind of interplay between people that exists over a duration of time. More importantly, like other activities we engage in, a card game comprises recurring patterns of interplay that form its totality. As such, the card game is an intuitive example that allows us to identify a commonly understood activity, and explicitly abstract and model its aspects.

Figure 5 illustrates the structure for a bidding process of a game, consisting of an activity `the_bidding` and the participating objects `Peter`, `Mary`, `John`, and `Jane`. `the_bidding` models an actual bidding activity at some specific point in time. Activities are abstractions over the interplay between entities, – here exemplified by the card game example. We have abstracted and classified that specific activity as a `bidding` activity (to be described later) and each of the persons as a `player` in the card game.

The particular example that serves as our model is the card game of Five Hundred. The object of the game is to score 500 points before the other players.

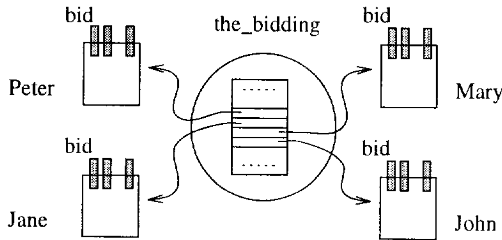

Fig. 5. The Card Game Example

Each game comprises one or more rounds; players are dealt cards and play against one another in each round. A player wins (or loses) points at the end of each round depending on how well he/she plays.

We distinguish between part-activities and sub-activities. Descendant activities – activities specialized from another (super)activity – are called *sub-activities* or just activities. Activities that can be aggregated to form larger activities (whole-activities) are called *part-activities.*

Part-Activities. At the highest level of organization, we can create a `cardGame` activity that represents the totality of interplay in the game. This activity actually comprises several subordinate phases: a `gameOpening` phase (where initialization and set-up take place), a `gameRounds` phase (in which one or more rounds are played), and a `gameClosing` phase (where clean-up procedures take place). We consider the activity `cardGame` to be composed of the part-activities `gameOpening`, `gameRounds` and `gameClosing` executed in sequence.

As most of the significant interplay takes place during each round, we shall further decompose the `gameRounds` part-activity. This phase of the game comprises one or more rounds – each of which is a part-activity. Like the `cardGame`, a `gameRounds` part-activity consists of an opening phase (`roundOpening`), a central execution phase (`roundPlay`) and a closing phase (`roundClosing`).

Figure 6 illustrates the hierarchical class organization of the `cardGame`: a part-whole hierarchy (where `gameOpening`, `gameRound` and `gameClosing` are part-activities of `cardGame`).

Activity Directive. In our normal understanding, the `gameRound` part-activity is where most of the card playing takes place. Each round has three distinct stages:

1. `dealing`: Cards are dealt to each player.
2. `bidding`: Each player is successively asked to make a bid – the players bid against each other, until the highest bidder is found. The player with the winning bid starts the game.
3. `trickTaking`: After bidding, the players engage in taking tricks. A trick involves each player putting down a card; the player whose card beats the

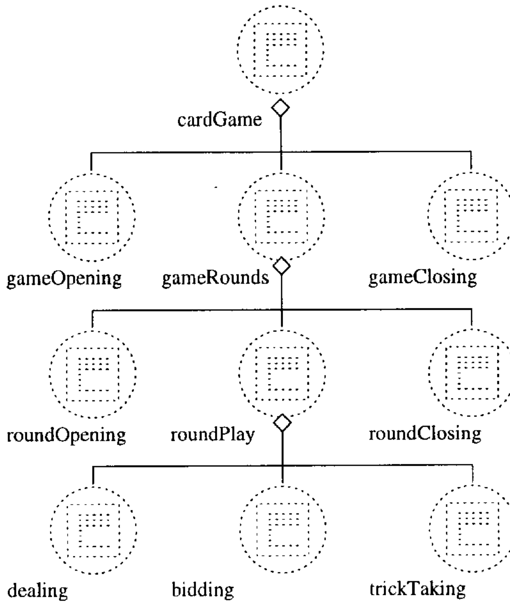

Fig. 6. Card Game: Part-Activities

others is said to have taken the trick. For 4 players, the trick-taking phase of each round involves the playing of 10 tricks.

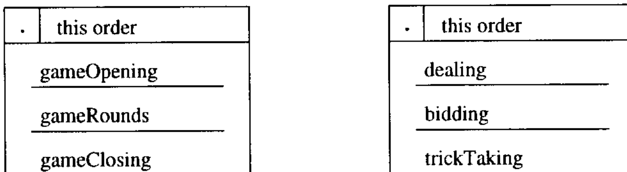

Fig. 7. Directives for `cardGame` and `roundPlay`

In Fig. 7, we illustrate the use of special diagrams for describing the structure of the directives of the `cardGame` and `roundPlay` activities. The notation '.' represents sequential action according to a given specification (e.g. deal cards according to rules).

Each activity/part-activity is responsible for managing its associated interplay. For instance, the `bidding` part-activity has to control the sequence of bids

performed by the players – as each player makes a bid, the part-activity will ensure that certain constraints are in force: Is the present bid legal? Who is the next bidder? When is the bidding process over and who is the winner?

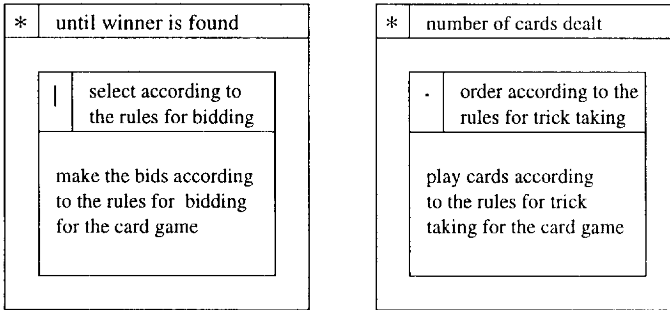

Fig. 8. Directives for `bidding` and `trickTaking`

In Fig. 8, we describe the structure of the directives of the `bidding` and `trickTaking` activities. The notation '*' indicates iterative execution – for example, continue execution performing these actions "until a winner is found". The notation '|' indicates that a selection action will take place, matching a given condition (e.g. perform bidding actions, then select the highest bidder).

Sub-Activities. The game of Five Hundred is only one example of a card game; we may consider Bridge as another example. Therefore, activities may be specialized – `cardGame` may be specialized to the sub-activities `fiveHundred` and `bridge`. Most of the part-activities of `cardGame` may be specialized similarly.

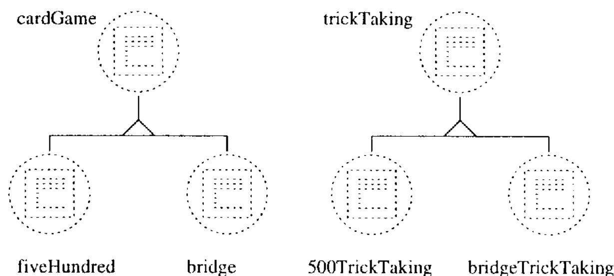

Fig. 9. Card Game: Sub-Activities

Figure 9 illustrates the hierarchical organizations of the `cardGame`: two generalization hierarchies where `fiveHundred` and `bridge` are sub-activities of `cardGame` and `500TrickTaking` and `bridgeTrickTaking` are sub-activities of `trickTaking`.

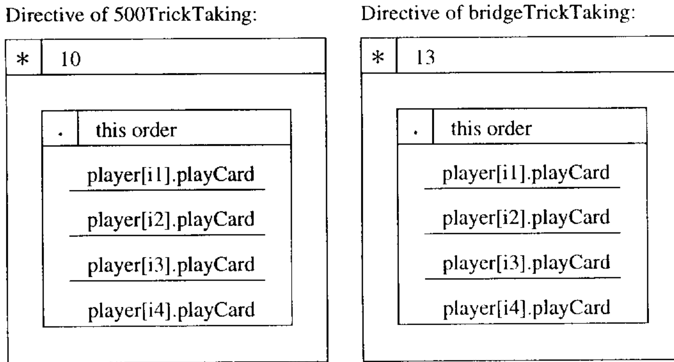

Fig. 10. `500TrickTaking` and `bridgeTrickTaking`

In Fig. 10, we describe the structure of the directives of the `500TrickTaking` and `bridgeTrickTaking` activities as specializations of the directive of the `trickTaking` directive (Fig. 8).

Properties of Activities. We have briefly mentioned how activities may have properties. The `roundPlay` activity has information about which cards have been played (`cards_played`) during the round and which tricks have so far been taken by the players (`tricks_taken`). The properties `cards_played` and `tricks_taken` of `roundPlay` are examples of *emergent properties* because they only emerge through the interplay of the part-activities of `roundPlay`. In other words, these properties do not exist independently of the part-activities.

The `bidding` activity has a property `bidding_steps` that details how each bidding step takes place, while the property `legal_bid` ensures that the next bid made by some player is a legal step (in relation to `bidding_steps`). The properties `bidding_steps` and `legal_bid` are examples of sub-activity properties for `500bidding` and `bridgeBidding`, that are refined (inherited) from the super-activity `bidding`. Thus, in the sub-activities, the checking of a bid's legality is specialized according to the rules of the particular card game.

In Fig. 11, the properties `cards_played` and `tricks_taken` for `roundPlay` and `bidding_steps` and `legal_bid` for `bidding` are shown in relation to the part-whole and generalization hierarchies.

Summary: Designing with Collective Activities. Activities are abstraction mechanisms over the interplay between entities, as exemplified by the card game ex-

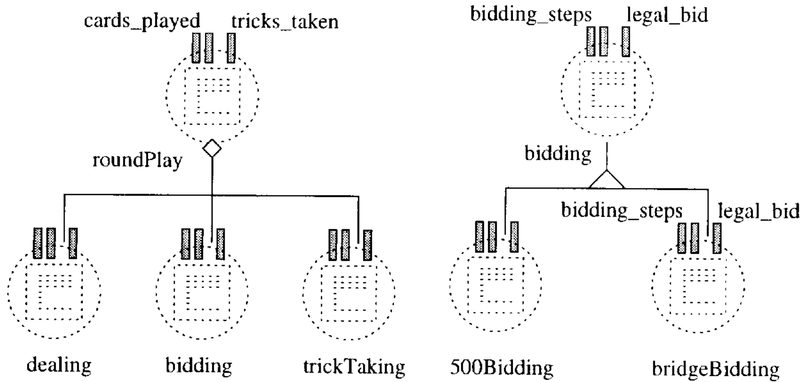

Fig. 11. Properties for `roundPlay` and `bidding`

ample. In the card game, the players are the entities; they are seen to *participate* in the game. For simplicity, we assume that the number and roles of players is fixed. During the game, the players will take turn either bidding, playing a card, etc – according to the rules of the card game (be it Five Hundred or Bridge). This sequence of behavior forms the card game activity. The legal sequence (and control) of possible behavior taken by the participating players is an abstraction over the possible games to be played in the specific type of card game.

The term *collective activity* denotes the abstraction of the actual sequence of behavior (in terms of the activity phenomenon) taken by the *participants* – on instantiation, the collective activity is *in progress*. The *directive* is the behavior description part of the collective activity. The term *collective structure* denotes the totality of the activity phenomenon and the participating phenomena – on instantiation, a collective structure is said to be *performing*.

Figure 12 illustrates the fundamental components of activities: the collective structure consisting of a collective activity (with directive) and a number of participants.

As the card game example indicates, the principles of aggregation and specialization may be applied to collective activities. In relation to aggregation, we have seen that it is possible to form whole-activities using subordinate part-activities. Elaborating on this, we can say that participants involved in these part-activities are seen as participants of the whole-activity. An activity's directive is also subject to aggregation. The directives of part-activities form the whole-activity's functionality – specifying the whole-activity's directive.

With respect to specialization, general activities and their participants (`review` and `coordinator`) may be refined (`paper_review` and `chairman`), and additional participants may be included. It is also possible to apply specialization to the directive of a general activity. This is possible by specializing the directives of an activity's part-activities (for example using virtual part-activities) or by explicitly adding to an activity's directive (for example via an `inner`-like mechanism).

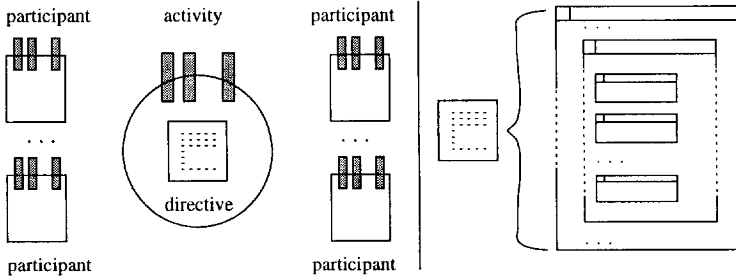

Fig. 12. Collective Structure, and Directive

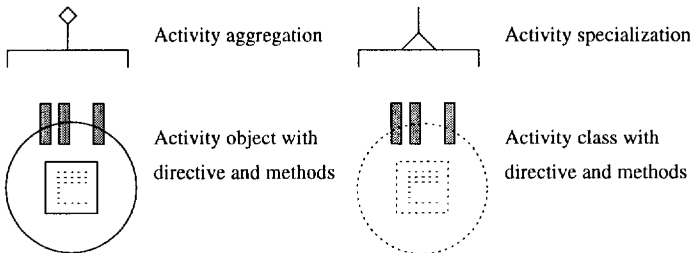

Fig. 13. Abstraction Diagrams

Figure 13 summarizes the class/object notation proposed for the activity as an abstraction mechanism. A general description of abstraction for activities in the form of classification, specialization and aggregation is given in (Kristensen 93b).

We have proposed a special notation for expressing the sequencing of an activity's directive. In (Kristensen & May 96) a more complete description of the diagrams is given. The diagrams employ the fundamental imperative sequencing forms (*sequence*, *selection*, *iteration*) , and are inspired from other diagrammatic notations (Nassi & Shneiderman 73).

3 Implementation of Activities

The following sections focus on the mapping from design with activities onto object-oriented language constructs. We discuss language constructs which, when added to existing object-oriented programming languages will directly support

the activity as a language mechanism. Such constructs are seen as an extension to these languages rather than a radical shift in programming perspective. In the uni-sequential execution the implementation of activities may be supported by a set of abstract classes.

3.1 Integrating Activity and Participant

Fundamentally, we see activities as relations between entities. At the same time, participants will belong to an activity. Our language constructs to support such properties of collective behavior.

Activities as Relations. The activity is described by a *relation-class.* The participants are temporarily related through their participation in the activity.

Activities can be seen as a type of relation between domains (which represent the participants). Therefore, we may define an activity class `A` acting as a relation between participant classes `B`, `C` and `D`:

```
CLASS B ( ... )
CLASS C ( ... )
CLASS D ( ... )

CLASS A [B, C, D] ( ... )
```

Such a declaration may be further extended. In the following example, objects of class `B`, `C`, `D` may be accessed from the names `rB`, `rC`, `rD`. Given "`rB : B`", `rB` is a reference to an object of class `B` – while "`rC :* C`" means that `rC` may refer to an arbitrary number of `C` objects (one-to-many cardinality).

```
CLASS A [rB : B, rC :* C, rD : D] ( ... )
```

Figure 14 illustrates the relation `A` with domains `B`, `C` and `D`.

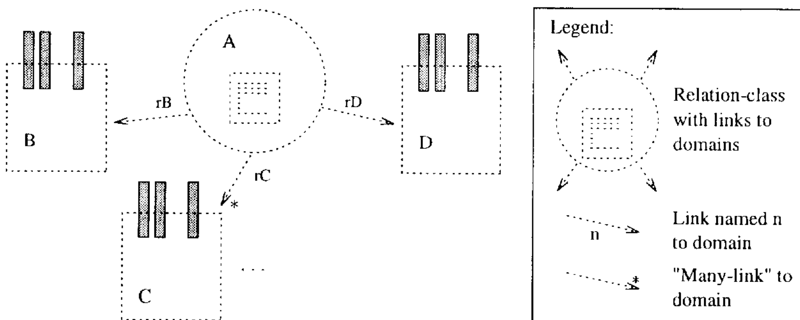

Fig. 14. Activity as a Relation

Figure 15 describes the `paper_review` activity in relation to its participants `chairman`, `reviewer`, and `author`, named respectively `the_chairman`, `reviewers`, and `an_author`:

```
CLASS chairman ( ... )
CLASS reviewer ( ... )
CLASS author ( ... )

CLASS paper_review
  [ the_chairman : chairman,
    reviewers :* reviewer,
    an_author : author ]
  ( ... )
```

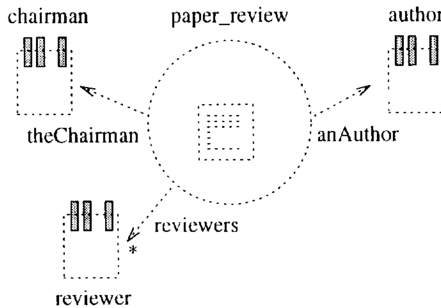

Fig. 15. `paper_review` as a Relation

The relations-class may be extended to include optional domains in order to describe the dynamically varying situation, where participant are coming and going during the activity. We shall not describe the use of activity objects for access of the participants etc, but refer to general descriptions of relation classes and objects in (Rumbaugh 87).

Participation using Roles. Often, participants in an activity have no logical existence when the collective structure is not performing. A person may take part in a `paper_review` activity several times during a year: the `reviewer` aspect of this person is only existent when he/she participates in the `paper_review`. Additionally, a participant may take part in several activities during some period of time, either in the same or different kinds of activities. Hence, different aspects of the participants may be relevant for different kinds of activities.

For example, a person taking part in a `paper_review` activity may be simultaneously writing a paper with some colleagues. In the `paper_writing` activity the person is an `author` whereas in the `paper_review` activity, he is a `reviewer`.

He/she may still take part in several `paper_review` and `paper_writing` activities during the same period. Language and modelling constructs should be able to give force to such design models.

Throughout the course of his/her lifetime, a `person` will take on roles *in addition* to his/her own functionality – at some point, a `person` will also act in the added capacity of a `reviewer` and as an `author`. To support these dynamic role changes, we introduce the concept of *role* (Kristensen & Østerbye 96) and *subject* (Harrison & Ossher 93). The concepts of role and subject allow different perspectives to be dynamically added to an object (for a given period of time), augmenting its integral properties. It is should be noted that this is distinct from dynamic class mutation as well as view of some existing object with its properties given.

In general, an entity may assume a number of roles at a given time. These roles may be allocated and deallocated dynamically. Roles are described as *role-classes.* A role-class is defined to be role for some class or other role-class. In the use of roles here, we define activities as relation-classes with role-classes as their domains. An object will play the roles given as a domain of the relation-class when the object takes part in an activity of the relation-class. This is then a means of describing the dynamic participation of entities in activities, as well as participation in several activities simultaneously.

`R1` and `R2` are role-classes for class `C`. The relation-classes `A1` and `A2` have the role-classes respectively `R1` and `R2` as one of their domain classes.

```
CLASS C ( ... )

CLASS R1 ROLE C ( ... )
CLASS R2 ROLE C ( ... )

CLASS A1 [... , R1, ...] ( ... )
CLASS A2 [... , R2, ...] ( ... )
```

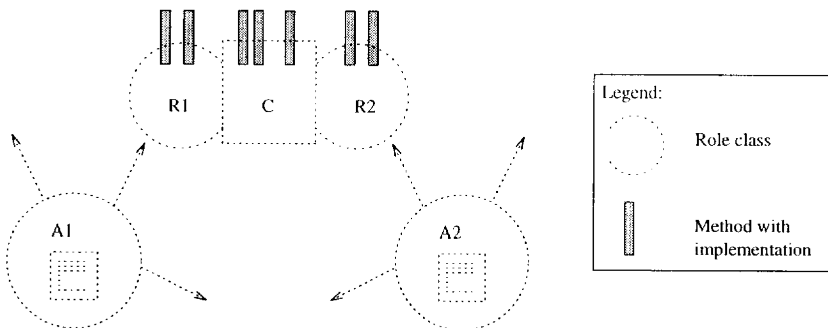

Fig. 16. `participants` as Role-Classes

Figure 16 illustrates a class C with role-classes R1 and R2 associated. An object of class C can acquire role-objects form R1 and R2 during its life cycle. The diagram shows that a role of role-class R1 (R2) is acquired for the activity A1 (A2).

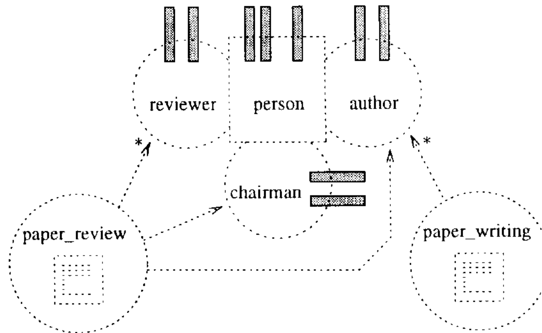

Fig. 17. Roles in paper_review and paper_writing

In Fig. 17, we illustrate how roles may be organized in the conference example. The diagram describes how reviewer is a role-class to person when a person is engaged in a paper_review activity. The same person may may engaged in a paper writing activity, paper_writing, where he/she plays the role of an author. A person object can have role-objects from these role-classes allocated and deallocated during its life-cycle.

```
CLASS person ( ... )

CLASS reviewer ROLE person ( ... )
CLASS author ROLE person ( ... )
CLASS chairman ROLE person ( ... )

CLASS paper_review
  [ the_chairman : chairman,
    the_reviewers :* reviewer,
    an_author : author ]
  ( ... )

CLASS paper_writing
  [ the_authors :* author ]
  ( ... )
```

3.2 Simulation of Activities

In uni-sequential execution (a single thread at one time), we need to integrate the participant's behavior with the progress of the corresponding activity. Either the activity or one of its participants is executing. To model how the participants are taking part in the activity, execution control has to switch between a participant and the activity to secure coordination. In the support of activities in the uni-sequential case we distinguish between *initiating* activities, where the activity is in charge and the participants are invoked from the activity in order to contribute to the process, and *reactive* activities, where the participants take initiative and the activity is 'awakened' to control and guide the process.

Initiating Activities. Participants may be seen as passive objects controlled and activated by the collective activity – the initiating activity. In this scenario, the activity will have the initiative towards the participants – invoking methods of the participants, in order to make them contribute to the progress of the activity. A participant may then (acting on its internal logic or by asking the user for direction) invoke one of the activity's methods. The activity is seen to be continuously guiding the participants, controlling their behavior through its method invocations.

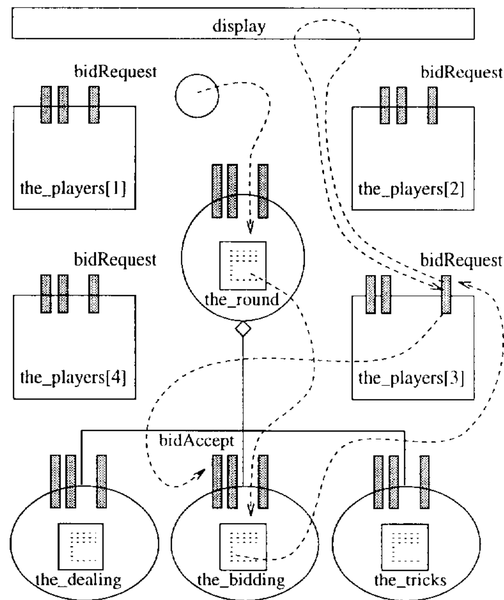

Fig. 18. Card Game: Initiating round

The following simplified version of the card game illustrates initiating activ-

ities – a single round of a Five Hundred game is modeled. An activity object `the_round`, will sequentially invoke three part-activities: `the_dealing`, `the_bidding` and `the_tricks`. Each of these part-activities respectively model the dealing, bidding and trick-taking phases of a round. The four players (`the_players[4]`) are the participants in `the_round`. Each player has a number of operations which model the various requests he/she can make during the game. The life-cycle of `bidding` is to repeatedly find and ask the next player to bid until the bidding is complete (according to the rules of the game). In the example, we have the operation `bidRequest` which asks a player to make a bid. The actual bid is registered by invoking the `bidAccept` method of the part-activity `the_bidding`. The `bidding` class has an operation `bidAccept` available to the players for placing their bids. In Fig. 18, we illustrate activities, part-activities, participants and a calling sequence in the example.

Reactive Activities. Alternatively, participants may be seen as active – actively executing or having some interface to 'active' users who will invoke the participants' methods. The main point to note is that participants will have the initiative towards the collective activity – the reactive activity, calling methods of the activity. In turn, these calls may provide feedback to the participants to direct what kind of 'input' is required next. The role of the collective activity is to police the behavior of the participants (rather than actively controlling), in some cases prohibiting a participant from performing certain actions.

In the simplified version of the card game, the life-cycle of `the_bidding` object is lie dormant until activated. When 'awakened' by a player, `the_bidding` object verifies the player's bid. This life-cycle continues until bidding is complete. For example, we have a player who makes a bid by invoking the operation `bidRequest`. The actual bid is registered by invoking the `bidAccept` method of the part-activity `the_bidding`. Once activated, the directive of `the_bidding` checks the legality of this bid. Also note that in this design, the whole-activity `the_round` lies dormant until the part-activity `the_bidding` wakes it from its waiting state. In Fig. 19, we illustrate activities, part-activities, participants and a calling sequence in the example.

Simulation by Abstract Classes. In the case of uni-sequential execution an alternative to extensions to existing language constructs is a set of abstract classes to support the use of activities. In (Kristensen & May 94) the implementation of both initiating and reactive activities in C++ (Stroustrup 91) is presented. In (Kristensen & May 96) the classes are expressed in an abstract, general object-oriented programming language – which may be translated into an existing language. Only initiating activities are simulated in detail. In the simulation it is assumed that classes can have action clauses, virtual references/methods, and inheritance for methods (action clauses and methods are extended by means of the `inner` mechanism originally introduced in SIMULA (Dahl et al. 84).

In these approaches we *simulate* activities in some existing object-oriented programming language, and we rely on the mechanisms available in the language.

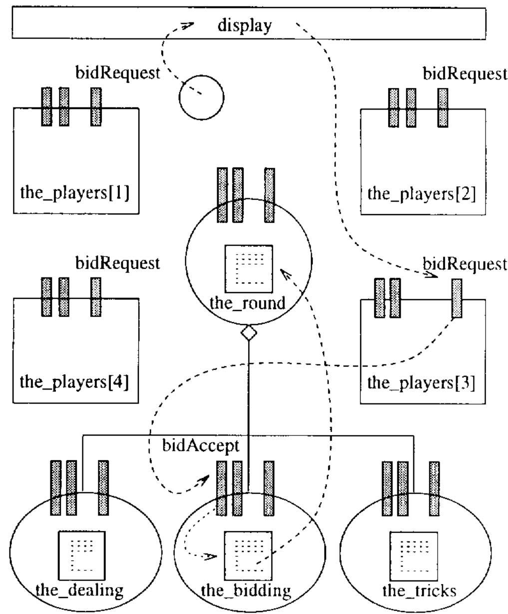

Fig. 19. Card Game: Reacting `round`

These mechanisms support – and restrict – our expressive freedom in describing the aggregation and specialization of participants and activity directives.

3.3 Concurrent Execution in Activities

In general, objects may execute concurrently (or multi-sequentially with interleaved execution, with only one active object at a time). We need special communication and synchronization constructs to properly describe the interplay between active (concurrently executing) objects, – in the form of activities and participants.

Active Objects. We introduce briefly a simple model with active objects. The `ACTION` clause illustrates that objects like `oC` may have an individual action part – on instantiation, an object will immediately execute its action part and is inherently active.

```
CLASS C
(
  METHOD M   ( ... )
  METHOD M'' ( ... )
  OBJECT R : C'
```

```
ACTION:
   ...
   M''
   ...
   R.MM
)

OBJECT oC : C
```

The description of the action part may involve the activation of methods in class `C` and methods for other objects than `oC`. Because the objects are active, the interaction between objects is usually coordinated by means of various forms of language mechanisms[3] available for the synchronization of the execution of the life cycle of the object and method activation requests from other objects. In the model we assume that when the object `oC` attempts a method request to `R` by `R.MM`, then `oC` must wait until `R` explicitly accepts this invocation. When the invocation is accepted the objects are synchronized and the invocation can take place.

Multi-Sequential Execution. The action part of an activity object describes the interaction of the activity with other entities (activities/participants). The collective activity is seen as a supplementary part of the life cycle of its participating entities – the life cycle of such an entity is described both in its own action part *and* in the various collective activities in which it is participating.

Given activity `A` and participant `rC`, some form of communication will take place between these two objects during execution. If `rC` is executing and `A` attempts a method execution `rC.M`, then `A` must wait until `rC` explicitly accepts this invocation. To avoid the additional synchronization arising from such a scheme, we specify that a participating entity may execute its action part *interleaved* with that of the method being invoked from an activity in which the entity takes part. For example, if `rC` is executing its action part and `A` calls `rC.M`, both method `M` and `rC`'s action part may execute in interleaved fashion.

The following schematic example illustrates the mechanisms introduced:

```
CLASS A [rB : B, rC : C]
( ...

ACTION:
   ...   rC::M   ...   rB::rC.M   ...
)

OBJECT oA : A
```

The activity object `oA` is of activity class `A`. The object `rC` is a participant of class `C` – we assume that `rC` denotes the object `oC`. The construct `rC::M` means

[3] According to (Chin & Chanson 91) this is an *active object model*; the model is *static* with exactly one thread per object and the thread is controlled by the description in the action part.

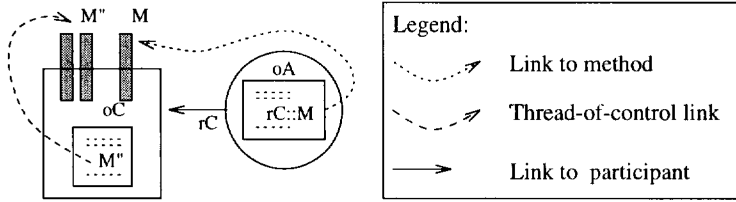

Fig. 20. Execution of `rC::M`

that the object `rC` is requested to execute its method `M` (Fig. 20) – because it is requested by an activity `oA` it is as if object `rC` itself requested the execution of `M`. This corresponds to the object `rC` invoking one of its own methods – the only difference is that the description is given outside `oC`. At the time of this request, the object `oC` may be executing its action part (e.g. executing `M''`). Here, the method `M` and the action part of `oC` are executed interleaved: at certain (language defined) locations, `oC` will switch between the execution of `M` and its action part.

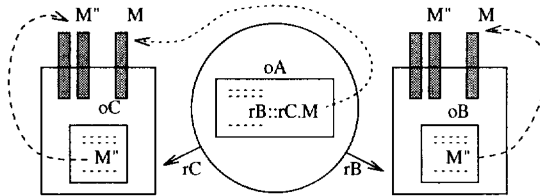

Fig. 21. Execution of `rB::rC.M`

The construct `rC::M` is only one example of the mechanisms that may be employed in collective activities and participants. The other example is `rB::rC.M`, where the participant `rB` requests `rC` to execute its method `M` (Fig. 21). In this case the activity specifies both the invoking participant `rB`, the invoked participant `rC` and the method `M`. Because this situation is as if the object `rB` itself requested the method execution `rC.M` a synchronization may be involved between `rC` and `rB` (as a case of ordinary communication between the active objects `rB` and `rC`).

In the `paper_review` example, `the_chairman` and `the_reviewers` may be actively performing other actions than those in connection with the `paper_review` (such as research, teaching, writing). It would be natural to describe such tasks in the life cycle of these objects (in the action part). Furthermore, `the_chairman` and `the_reviewers` may be engaged in other collective activities concurrently (e.g. participating in a selection committee, writing a paper) – possibly in different roles.

```
CLASS reviewer
( ...
  METHOD remind : ( ... )
  METHOD submit : ( ... )
  METHOD is_busy : ( ... )
)

CLASS author
( ...
  METHOD submit_paper : ( ... )
)

CLASS chairman
( ...
  METHOD distribute : ( ... )
)

CLASS paper_review
  [ the_chairman : chairman,
    the_reviewers :* reviewer,
    an_Author : author ]
( ...

ACTION:
  ...
  the_author::submit_paper
  the_chairman::distribute
  ...
  if the_reviewers[i]::is_busy then
     the_chairman::the_reviewers[i].remind
  ...
)

OBJECT a_paper_review : paper_review
```

The situation could arise where one of `the_reviewers` did not supply a referee report within the specified time. Here, it is possible for the `paper_review` activity to monitor the overall progress of the activity, checking for such an occurrence – the activity may then send a reminder to one of these `the_reviewers` on behalf of `the_chairman`.

Figure 22 demonstrates how a `paper_review` activity may request participants (here `the_reviewers[i]`) to invoke methods while they are performing their own sequence of execution. The directive shows how the activity may communicate on behalf of the participants (here `the_chairman`).

In general, activities may be performed simultaneously – interleaved or concurrent (e.g. `paper_review` and `paper_writing` activities). Nested execution is supported by the sequential execution of part-activities. Additionally, overlapping execution of part-activities (either interleaved or concurrent) may also be supported (the `prepare_review_process` and the `carry_out_review_process` may over-

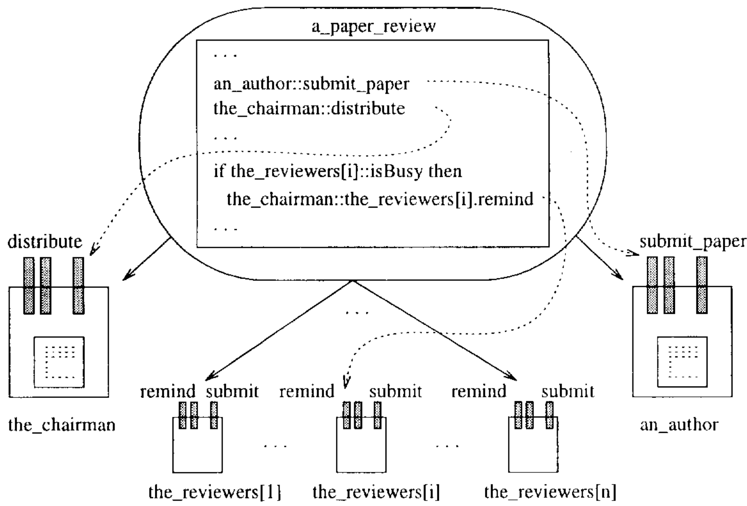

Fig. 22. Requests sent by `paper_review`

lap without problems to speed up the process).

The refinement of the directive of an activity in the multi-sequential case – by means of a multiple `inner` mechanism (Kristensen 93b) – is motivated by two principles. Firstly, it should be possible to describe additional sequences of action which must be executed simultaneously (interleaved or concurrent) with sequences of the existing directive. Secondly, it should be possible to ensure that parts of the existing directive of an activity are refined, allowing the part of the existing directive *and* the refined part to execute together as a unit. These two principles may be applied in a mixed sequence in the refinement of the directive.

4 Experiments

In this section we review some practical and theoretical experiments based on the notion of activities; we classify activities in the dichotomy of software abstraction; we summarize the experience from a project focused on the construction of a framework for card games; and we compare implementations based on design patterns and the activity abstraction.

Where should activities be classified in the dichotomy of software abstraction? Activities are distinct from frameworks and design patterns. A framework (Johnson & Foote 88) is a software architecture, including an abstract program. In essence, it provides a reusable design solution for a specific class of software – the design decisions that are common to the framework's domain are captured. Frameworks are specialized to become applications in the domain.

A design pattern can be seen as an organizational idiom, comprising an

abstract code component. A pattern represents the core of a solution to similar recurring problems, comprising a general arrangement of classes/objects (Gamma et al. 94). Design patterns are more abstract design elements than frameworks (they may be applied in the construction and design of a framework) and their architectural granularity is finer.

The activity is a modelling and programming mechanism that may be used to describe a wide variety of programs and program fragments, abstract or concrete. It is an abstraction mechanism, able to be used to create abstractions of varying definition – from the more abstract/general (design patterns) to the more concrete (frameworks). This abstraction mechanism enables us to have another basic component in the design vocabulary we use when creating such abstractions.

Frameworks, design patterns and abstraction mechanisms are also used differently in the program development process. The universe of frameworks and design patterns is not, by nature, finite. Therefore, it will be necessary to search for and identify appropriate frameworks/design patterns that may be applicable to the problem. In the case of a framework, you will need to understand its functionality and how to customize it. With design patterns, it is necessary to recognize the context in which a pattern could be applied and how it should be realized in a concrete design.

An abstraction mechanism forms part of the language. It is therefore *fundamental* in its influence on how we conceive the world around us, how we initially form our understanding, and then later in expressing it. Such mechanisms give us a basic lexicon with which we can describe higher-level, structured abstractions – like frameworks and design patterns.

Card Games and C++. The activity abstraction was investigated in a project described in (May 94). The objective of the project was to explore issues related to the design and construction of object-oriented frameworks – the C++ language was used to build software artifacts through the course of the project.

The problem domain on which the study concentrated was that of card games, namely, designing a framework for writing card game applications. Several pieces of software were produced: a Blackjack game (to gain experience in the problem area), a card game framework, and a Five Hundred game.

We explored the activity abstraction to address the issue of representing more complicated sequences of interplay yet simplifying their complexity. A framework for activities was created, which later became the basis for a card game framework. This was eventually used to create the Five Hundred game.

Limitations were encountered using C++. In its 'standard' form, the language lacks multi-sequential and concurrent mechanisms – it is not possible to properly represent multiple executing active objects or reactive activities. Further, there is no support for locality.

Overall, the project indicated that most mainstream object-oriented languages had little support for the facilities required to accurately simulate activities. While activities could be implemented in such languages, the absence of such support will result in an implementation solution that does not properly

match the design solution – and is less intuitive. Languages should provide more flexible constructs (e.g. sub-method inheritance, locality, roles) to enable the design model to be implemented *and communicated* in a far more natural way.

Conference Organizing and the Mediator. The activity abstraction was compared to design patterns, especially the Mediator (Gamma et al. 94), in an implementation project. Descriptions of a subset of a conference organizing system was developed and compared.

The intent of the Mediator pattern is to "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently".

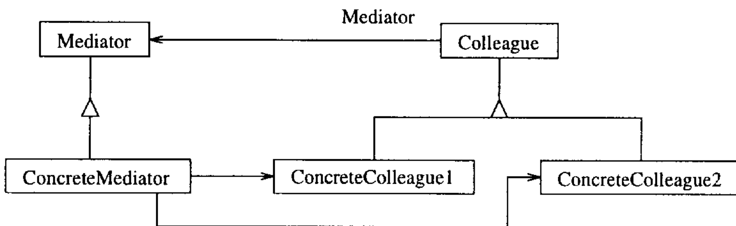

Fig. 23. Structure of the Mediator Pattern

In Fig. 23 the structure of the Mediator pattern is reproduced. The `Mediator` class corresponds to an activity and the `Colleague` classes to participants. The `ConcreteMediator` class implements cooperative behaviour by coordinating `Colleague` objects, and knows and maintains its colleagues. Each `Colleague` class knows its `Mediator` object, and each `Colleague` class communicates with its mediator whenever it would have otherwise communicated with another colleague.

From our description in the experiment, the Mediator pattern appears to be similar to how we characterise the collective behavior of activities. Indeed, there is centralisation of control into an object – which manages the interaction between other objects. But while the overall architectural techniques are similar, motivation and focus for collective behavior is different.

At its heart, the design pattern aims to provide an abstract, general design solution to a set of problems. It is structured to be less well-defined and more informal. In short, it is an abstraction that is not supposed to be set in concrete: its malleability and generality is its strength.

On the other hand, our approach has been at a more elemental and atomic level. The purpose of our research was to inquire into the nature of interaction – first, our intuitive understanding, then mapping into an object-oriented domain. We have therefore characterised and defined how we picture activities: the ways

in which object-oriented concepts may be applied to them, and how they relate to each other and their participants.

The activity is an abstraction mechanism – a type of brick that can be used to build abstractions. In the same way that a *class* is used to create abstractions (like design patterns), the *activity* is used as well-defined component in constructing abstractions. In short, the activity is a mechanism that is supposed to be capable of extension, but well-defined and less vague.

Similar to the class, it is used 'as is' rather than requiring a mapping into specific domains (like design patterns). Therefore, it is possible to envisage building libraries of components that have been constructed using activities (and other mechanisms). The nature of this abstraction mechanism covers the ambit of modelling and programming languages. Thus, we can use activities as building mechanisms in creating modelling abstractions then isomorphically map into a concrete language.

Furthermore, the emphasis of the Mediator pattern is to promote loose coupling and encapsulate object interaction. The activity abstraction mechanism also seeks to give force to these goals; of themselves, they are useful and worthwhile objectives. But the emphasis of our inquiry was to represent and give force to the properties of interaction – collective and otherwise.

We also investigated beyond the generalised object arrangement scheme embodied by the Mediator. The present paper looks at a dichotomy of how activities can be classified into initiating and reactive forms, how they may exploit the power of object-oriented properties in a natural fashion, and supporting the characteristics of activities using concepts such as *relations* and *roles*. This clearly exceeds the scope and definition of the Mediator pattern.

In essence, activities are used to build abstractions – solutions.

5 Summary

The underlying assumption in this paper is that in existing object-oriented methodologies and languages, objects appear as isolated elements with an implicit and poor description of the interplay structure between them. However, as human beings we identify such interplay structures as another kind of phenomena – usually known as activities – and inspired of this kind of phenomena we introduce an abstraction mechanism, which may be used to model the interplay structures. This language mechanism – the activity – is expressive and powerful for the modeling of organization and interplay of usual objects, – the collective behavior of objects.

Related Work. While not discussed explicitly, collective activities are within the ambit of Booch's description of an object (Booch 94): "tangible and/or visible thing; something capable of intellectual apprehension; and something toward which thought or action is directed". Most pointedly, "an object models some part of reality and is therefore something that exists in time and space".

In OMT (Rumbaugh et al. 91) the dynamic model is a collection of state diagrams that interact via shared events. A *state diagram* describes the life cycle of a class of objects – but only from a local perspective. State diagrams are also related to the object structure: the aggregation of objects implies the aggregation of the state diagrams of these objects, resulting in composite states; the specialization of classes implies that a subclass inherits the state diagram of its superclass and together with the state diagram added it works as a composite of concurrent state diagrams.

Responsibilities, *collaborations*, and *contracts* from (Wirfs-Brock et al. 90) define the dependencies between objects and subsystems. However, these are only concerned with the static dependencies, and there is no support for the description of the dynamic interplay between them.

Use cases from (Jacobson et al. 92) models system functionality. Use cases – with actors of various kinds – are abstractions of the user's interaction with a system. The actor and system in dialogue is a sequence of transactions each initiated by some stimulus from the actor. An actor may be involved in a number of use cases. A user is seen as an instantiation of an actor, and an actual execution of an interaction session with a user is seen as an instantiation of a use case. This instantiation yields an object. A use case is described by a state transition graph, and user stimuli imply state changes. The description of a use case is organized as a basic course and several alternative courses. A use case may be seen as a special case of a collective activity, which is restricted to the system functionality; it is an abstraction, but no generalization and aggregation hierarchies are discussed for use cases, only the distinction between abstract/concrete use cases, as well as the insertion of a "part" use case into another by means of *extends*.

BETA (Madsen et al. 93) has active objects in the form described here but no mechanisms for supporting the combined execution of the directive of an activity-object and the life cycles of its participants. The aggregation and specialization as presented here can (with a few exceptions) be seen as an adaptation of corresponding mechanisms of BETA. Relations and roles – as described here – are not supported.

Contracts (Helm et al. 90) are specifications of behavioral dependencies between cooperating objects. Contracts are object-external abstractions and include invariants to be maintained by the cooperating objects. The inter-object dependencies are made explicit by means of supporting language mechanisms. The result is that the actions – i.e. the reactions of an object to changes – are removed from the object and described explicitly in the contracts: the objects are turned into reactive objects, whereas the reaction-patterns for an object in its various relations with other objects are described in the corresponding contracts. The intention of the contract mechanism is not the modeling of real world phenomena and their interdependencies – rather, it is to have a mathematical, centralized description that supports provable properties. In (Holland 92) a further development of contracts is presented. Contracts are used for representing and reusing algorithmic programming cliches.

According to (Gamma et al. 94) the *Mediator design pattern* has the fol-

lowing benefits and drawbacks: It limits subclassing. It decouples colleagues. It simplifies object protocols. It abstracts how objects cooperate. It centralizes control. These qualities are also valid for the activity abstraction. In contrast, the description of the Mediator pattern does not include any considerations about multi-sequential execution of participants and activities. A straightforward use of the Mediator pattern the multi-sequential case will certainly lead to a lot of unnecessary synchronization. Furthermore, we have presented a notation/language constructs for activities for use both in the design phase and in the implementation phase. Most design patterns, including the Mediator pattern, can be used in both these phases too. The activity is defined differently for these phases, in order to support the informal, intuitive description during design, as well as the formal, though still high level description in terms of relations, roles etc at the implementation level.

There are well-known, existing abstraction mechanisms that are commonly used to construct *control abstractions* (Tennent 81). Such examples include the procedure, coroutine, and various process mechanisms. Similarly, the activity mechanism supports abstraction of control. However, in the same way that the class is not merely equivalent to a record or structure, the activity mechanism integrates concepts of functional behavior with object-oriented modelling capabilities. Activities can be used to coordinate/control objects – not merely stating an order or sequence of action, but explicitly stating the nature of the interplay: the participants, their relationships, and the actions that take place between activities and participants.

In (Aksit et al. 94) *Abstract Communication Types* (ACTs) are classes/objects in the object-oriented language Sina. The purpose of ACTs is to structure, abstract and reuse object interactions. The *composition filters* model is applied to abstract communications among objects by introducing *input* and *output composition filters* that affect the received and sent messages. Using an input filter a message can be accepted or rejected and, for example, initiate the execution of a method. Inheritance is not directly expressed by a language construct but is simulated by the input filter by delegating a message to the methods supported by *internal* objects. Several primitive filters are available in Sina, e.g. *Dispatch*, *Meta*, *Error*, *Wait*, and *RealTime*. In ACTs the Meta filter is used to accept a received message and to reify this as an object of class *Message*. The requirements for ACTs include large scale synchronization and reflection upon messages; the ACT concept is used as an object-oriented modeling technique in analysis and design. ACTs appear to be a technical, very comprehensive concept that – according to (Aksit et al. 94) – supports a wide variety of object interaction kinds including action abstraction, distributed algorithms, coordinated behavior, inter-object constraints, etc.

Results. The main results are summarized as:

- Intuitive and general understanding of the fundamentals of activities as abstractions for collective behavior. Activities support the modeling of the organization of and interplay between objects in object-oriented analysis, design, and implementation.

- Activities support modeling that is more similar and intuitive to our human understanding – in our clustering of information and abstracting of detail (particularly of processes): A notation to support the modeling with activities in analysis and design.
- In the implementation activities offer an orthogonal solution to expressing and manipulating collective behavior of objects:
  (1) Abstract classes for the support of implementation of activities.
  (2) Language features for direct support of activities as an abstraction mechanism in object-oriented programming languages.

Challenges. There exist numerous issues with activities that remain to be investigated and/or resolved:

Dynamic participation: An activity relates the interplay between various participating entities. The actual entities participating may change during the activity – entities may join or leave the activity. This has been achieved to a certain degree by allowing participants to assume additional roles during the life cycle of an activity. However, this is a participant-centric view of dynamic participation. From the activity's point of view, it is not possible to tell which participants are involved in the activity. (This is a general problem in programming languages – to be able to "know" which objects are associated in a given relationship.)

Part-whole activity state access: It is an open question as to the degree of state access enjoyed between part-activities and their whole-activities. Should a part-activity have state access to its enclosing whole-activity? Part-activities will not generally execute in isolation or ignorance of their whole-activities' properties – however, a relatively high degree of encapsulation should be enforced between activity classes. Conversely, it is a question as to whether a whole-activity has automatic access to the state of its part-activities.

References

Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting Object Interactions Using Composition Filters. Proceedings of the ECOOP '93 Workshop on Object-based Distributed Processing, Guerraoui, R., Nierstrasz, O., Riveill, M. (Eds.), LNCS 791, Springer-Verlag, 1994.

Booch, G.: Object Oriented Analysis and Design with Applications. Benjamin/Cummings, 1994.

Chin, R. S., Chanson, S. T.: Distributed Object-Based Programming Systems. ACM Computing Surveys, Vol. 23, No. 1, 1991.

Dahl, O. J., Myhrhaug, B., Nygaard, K.: SIMULA 67 Common Base Language. Norwegian Computing Center, edition February 1984.

Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

Harrison, W., Ossher, H.: Subject-Oriented Programming (A Critique of Pure Objects). Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, 1993.

Helm, R., Holland, I. M., Gangopadhyay, D.: Contracts: Specifying Behavioral Compositions in Object-oriented Systems. Proceedings of the European Conference on Object-Oriented Programming / Object-Oriented Programming Systems, Languages and Applications Conference, 1990.

Holland, I. M.: Specifying Reusable Components Using Contracts. Proceedings of the European Conference on Object-Oriented Programming, 1992.

Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: Object-Oriented Software Engineering, A Use Case Driven Approach. Addison Wesley, 1992.

Johnson, R. E., Foote, B.: Designing Reusable Classes. Journal of Object-Oriented Programming, 1988.

Kristensen, B. B.: Transverse Classes & Objects in Object-Oriented Analysis, Design, and Implementation. Journal of Object-Oriented Programming, 1993.

Kristensen, B. B.: Transverse Activities: Abstractions in Object-Oriented Programming. Proceedings of International Symposium on Object Technologies for Advanced Software (ISOTAS'93), 1993.

Kristensen, B. B., May, D. C. M.: Modeling Activities in C++. Proceedings of International Conference on Technology of Object-Oriented Languages and Systems, 1994.

Kristensen, B. B., May, D. C. M.: Modeling with Activities: Abstractions for Collective Behavior. R 96-2001, IES, Aalborg University, 1996.

Kristensen, B. B., K. Østerbye: Roles: Conceptual Abstraction Theory & Practical Language Issues. Accepted for publication in a Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity in Object-Oriented Systems, 1996.

Madsen, O. L., Møller-Pedersen, B., Nygaard, K.: Object Oriented Programming in the Beta Programming Language. Addison Wesley 1993.

May, D. C. M.: Frameworks: An Excursion into Metalevel Design and Other Discourses. Department of Computer Science, Monash University, 1994.

Nassi, I., Shneiderman, B.: Flowchart Techniques for Structured Programming. Sigplan Notices, 8 (8), 1973.

Rumbaugh, J.: Relations as Semantic Constructs in an Object–Oriented Language. Proceedings of the Object–Oriented Programming Systems, Languages and Applications Conference, 1987.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall 1991.

Stroustrup, B.: The C++ Programming Language. 2/E, Addison-Wesley 1991.

Tennent, R. D.: Principles of Programming Languages. Prentice Hall, 1981.

Wirfs-Brock, R., Wilkerson, B., Wiener, L.: Designing Object-Oriented Software. Prentice Hall, 1990.