Modeling Activities in C++

Bent Bruun Kristensen *
Institute for Electronic Systems, Aalborg University
Fredrik Bajers Vej 7, DK-9220 Aalborg Ø, Denmark
e-mail: bbkristensen@iesd.auc.dk

Daniel C.M. May
Department of Computer Science, Monash University
Clayton, Victoria 3168, Australia
e-mail: dcmay1@aurora.cc.monash.edu.au

Abstract

The object-oriented paradigm seeks to model phenomena as entities possessing state and behaviour. Usually, the interaction between such objects in a system is modeled as a sequence of method invocations – a method of one object sending a message to the method of another – such that the logic and control of interaction is distributed throughout the entities.

As the number of participating entities increases, the ability to discern the locus and logic of interaction rapidly becomes obscured. In an attempt to manage such complexity, we introduce the notion of an *activity* entity, as an abstraction to model the interaction between objects. Thus, the description of the logic of interaction becomes explicitly and partially centralized.

Activities are describes as classes, possessing such standard properties as methods and attributes – thus allowing interaction to be modeled by such abstractions as aggregation/composition and inheritance. As such, activities may be employed in diverse and subtle ways.

We have implemented activities using a set of abstract classes, specifying two types of activity: *initiating* (where the participant objects are directed and controlled by the activity) and *reacting* (where the participants act of their own accord, and the activity acts only in a controlling role). The framework is implemented in C++.

*This research was supported in part by the Danish Natural Science Research Council, No. 11-0542-1, while this author was on leave at Department of Computer Science, Monash University, Australia.

0

1 Introduction

The object-oriented paradigm involves the modeling of phenomena as entities (objects) that encapsulate behaviour and state. More complex structures such as systems can be constructed by combining sets of objects; relationships are specified between these entities, defining the nature in which they interact.

Object-oriented modeling facilitates the construction of such systems by providing mechanisms such as encapsulation, information hiding and message passing. An individual object's internal structure can be self-contained and isolated, while its interface to its surrounding environment is explicit and enforces a standard access protocol that other components must observe. We can refer to such characteristics of an object as its *intra-object integrity*.

As the number of participant objects within a software system increases, so too does the complexity of interaction between them. The resultant effect is that it becomes more difficult to discern the interrelation and logic of interaction between component objects – a crucial factor in the maintenance and expression of a system's design.

Just as intra-object integrity ensures the creation of sturdy components, it is necessary for *inter-object integrity* to exist at a higher level of construction/design; components should be bound together by abstractions that express the nature of their relationships. Further, such abstractions should serve to unify and centralize the locus of interaction between these objects.

This paper presents the *activity* abstraction, that seeks to capture the interaction between groups of objects at different points in time – this interaction is modeled as a distinct entity, such that it is capable of being treated in the same way as an object. It is emphasized that this object relates other objects to each other, describing not merely their participation in the relationship, but their interaction.

Why Activities? An activity can be defined as an interaction between entities over a given time – intuitively, this correlates to a general everyday understanding of an 'activity'. We may engage in the activity of ...

We use the example of teaching at university on a given day. We say "I taught at university yesterday". You subsume all the activities that you carried out in that day with that single statement.

Your actual activities included:

- Teaching activity: 9.00-10.00, in which you engaged with other participants of the activity (students). At the conclusion of this activity, you proceeded to another teaching activity.

  Within each teaching activity, you engaged in individual interaction with some students - this could be an activity in itself too.

- Lunch activity with other lecturers. This involved an eating activity, followed by an informal seminar activity.

And it is normal to describe our participation in an activity as a coherent unit; we engage in the activity as a single module and describe it to others in the same way. The significant feature of viewing activities as units of collective operation is a reduction in complexity – we similarly achieve a reduction in complexity in our real-world interactions when we describe our actions.

This abstraction depicts the relationships that link interacting objects; it is more than a mere gathering together of objects. Applying the object-oriented paradigm, this parcel of collective behaviour can be objectified, resulting in an entity/object that represents a contiguous unit of process. Activities will then possess similar capabilities to normal objects; they may be aggregated, composed, and recursively defined.

Thus, it is possible to view the behaviour of a system (such as a framework) as a collective set of behaviours of such units; similarly, these units of behaviour (activities) can be described in terms of the interaction between their participants.

By the language used thus far to describe the concept of an activity, it is possible to detect the temporal nature of this abstraction. An activity such as *day-at-university* has an inception, an execution, and a termination phase; the many everyday activities that we perceive also possess a lifetime and a context (e.g. after 5:15 p.m., we no longer engage in the activity of teaching at university. Our *day-at-university* activity is over. At the end of each class, the *class-teaching* activity is over, but the *day-at-university* activity is not over yet).

The potential benefit of characterising the behaviour 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 interaction between objects.

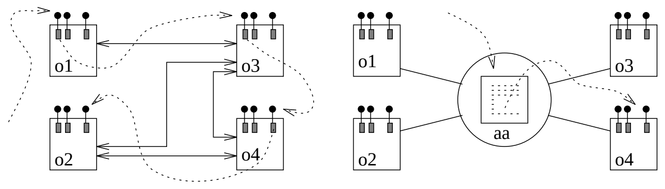


Figure 1: Alternative Execution Sequences

In Figure 1, we illustrate two alternative forms of execution sequence. Firstly, we present the usual object-centric method invocation, where method calls are scattered throughout the object organisation. Alternatively, an activity may be used to organise method invocation, where the execution sequence is centralised, abstracting the interaction between objects.

The use of activities as abstractions in object-oriented analysis, design, and programming is introduced in [Kristensen 93a].

The Results of this Paper. The main results in this paper are summarized as:

- Focus on the use on activities in the modeling process.

- Intuitive and general understanding of the fundamentals of activities.

- Abstract classes for the support of activities in C++ [Stroustrup 91].

- The distinction between two important types of implementations of activities: *initiating* activities and *reacting* activities.

Paper organization. In section 2 we discuss the use of activities as they apply to the modeling process. We use card games as a concrete example, and we discuss generally the fundamental characteristics of activities. In section 3 we present an implementation of activities in C++. A general activity class may be used for describing application specific activity classes. The general activity class forms the basis for a framework. In section 3.1 the framework supports *initiating* activities: the activity has the initiative and will activate – and control – the participants whenever their participation is needed. In section 3.2 the framework supports *reacting* activities: the participants have the initiative and the activity will only react upon – and control – their requests. In section 4 we review an experimental project that focused on the construction of a framework for card games based on the notion of activities, and summarize the experience from the project. In section 5 we summarize the proposals and the results of the paper.

2 Modeling with Activities

We shall use a card game as a concrete example to illustrate the use of activities in the modeling process. Through this example, we hope to demonstrate and discuss the fundamental characteristics of activities.

Card Game Example Our intuitive understanding of a card game is that it is a human activity – it involves a specific kind of interaction between people that exists over a duration of time. More importantly, like other activities we engage in, a card game comprises recurring patterns of interaction (part-activities) [1] 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 2 illustrates the structure for a game, consisting of an activity `theGame` and the participating objects `player1-player4`.

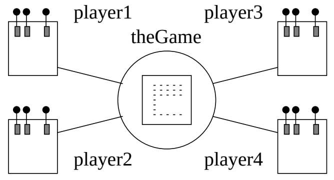


Figure 2: The Card Game Example

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. 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.

At the highest level of organisation, we can create a `cardGame` activity that represents the totality of interaction in the game. This activity actually comprises several subordinate phases: a `gameOpening` phase (where initialisation 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 interaction 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 comprises an opening phase (`roundOpening`), a central execution phase (`roundPlay`) and a closing phase (`roundClosing`).

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 in a special sequence 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.

[1] We distinguish between part-activities and sub-activities: Descendant activities – activities specialized from another activity – are called *sub-activities* or just activities. Activities aggregated to form larger activities are called *part-activities*.

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

Each activity/part-activity is responsible for managing its associated interaction. For instance, the `Bidding` part-activity has to control the sequence of bid actions 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?

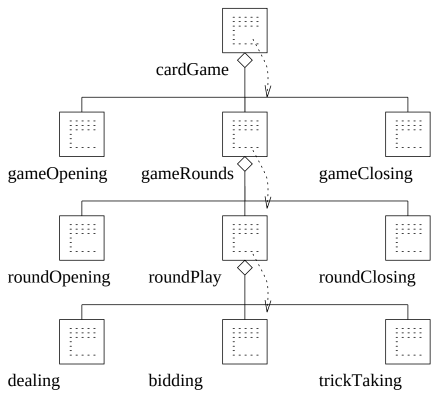


Figure 3: Card Game Activities and Part-Activities

Figure 3 illustrates the hierarchical organisation of the activity of a Five Hundred game: `cardGame` invokes `gameRound` which, in turn, invokes `roundPlay`, etc.

Fundamentals of Activities. Activities are abstractions over interactions between components, as exemplified by the card game example. Such activities are identified and then classified as `cardGames`. We have seen that an activity may be composed of part-activities: `cardGame` is seen as an aggregation of `gameOpening`, etc. The game of Five Hundred is only one example of a card game; another example is the Blackjack (21) card game. Therefore, activities may be specialized: for example, `cardGame` may be specialized into `fiveHundred` or `blackjack` game activities. A general description of abstraction for activities in the form of classification, specialization and aggregation is given in [Kristensen 93b].

In the card game example the players are the components; they *participate* in the game. For simplicity, we assume that the number of players is fixed and there is no exchange of players. In the card game the players take turn, for example, either bidding, playing a card, etc., – always according to the rules of the specific card game. This sequence of actions forms the card game activity. The legal sequence (and control) of possible actions taken by the participating players is an abstraction over the possible games to be played in this specific type of card game.

The term *transverse structure* denotes the totality of the activity phenomenon and the participating phenomena. A transverse structure *is performing.* The term *transverse activity* denotes the abstraction of the actual sequence of actions (the abstraction of these in terms of the activity phenomenon) taken by the participants. The activity *is in progress.* The *directive* is the action description part of the transverse activity. In Figure 4 we illustrate the fundamental elements of activities: The transverse structure consists of a transverse activity (with a directive) and a number of participants.

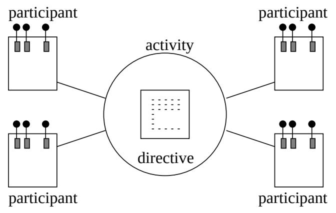


Figure 4: Transverse Structure

The participants may be seen as active, – either explicitly as active objects – or more likely in this example implicitly by means of some interface to the "active" users, who will invoke the methods of the participants. The point is that the participants have the initiative towards the transverse activity by calling methods of the activity (these calls may provide feedback to the participants (and the users) to direct what kind of "input" is needed afterwards). The transverse activity is guiding the actions taken by the participants and may in some cases prohibit an action from being executed.

Alternatively the participants may be seen as passive objects, controlled and activated by the transverse activity. In this situation, the activity will have the initiative towards the participants by calling methods of the participants in order to make the participants contribute to the progress of the activity. A participant will then possibly call a method of the activity – either acting on its own internal logic, or by prompting the user for directions on what to do. In this case, the activity continuously guides the participants and also controls the actions taken by the participants with respect to the activity's method calls.

3 Implementation

In C++ an activity is modeled by an object. The directive of an activity is implemented by a method. The associated participants are denoted by references between the activity object and each participant, and visa versa.

An activity object has a *state*. The state is composed of some *data* and a *point of execution*, both of which can be simple or complex. The state registers the current situation of the activity. Whenever an action is performed by a participant in the activity – either on its own initiative or requested by the activity object itself – the activity object checks its state to control the legality of the action and – if the action is approved – updates its state according to the effect of the action.

Activities can be *initiating* or *reacting*, reflecting different approaches to a system's overall design. Initiating activities may be used where there is a central thread of execution dominating the system. Here, the activity will usually drive program execution, activating the participants as needed (e.g. programs in single-tasking environments such as DOS).

Alternatively, reacting activities will passively await activation, to be carried out by the participants. These participants may be executing concurrently or be controlled by another portion of the system that is acting as the main program. Such activities may be employed in multi- tasking environments and event-driven systems.

General Activity Class. Activities and participants are described as subclasses of general classes, `activity` and `participant` respectively. The class `activity` is described as follows:

```
class Activity {
  public:
    Activity( Activity* p = NULL );
    virtual void activate() = 0;

    boolean inProgress() const
      { return ( h_begun && ! i_complete ); }

    boolean hasBegun() const
      { return h_begun; }

    boolean isComplete() const
      { return i_complete; }

    virtual ~Activity();

  protected:
    virtual void beginActivity()
      { h_begun = TRUE; }
    virtual void endActivity()
      { i_complete = TRUE; }

    boolean h_begun, i_complete;
    Activity* parent;
};


Activity::Activity( Activity* p = NULL ) {
  parent = p;
  h_begun = i_complete = FALSE;
}
```

The methods `beginActivity` and `endActivity` are used for setting the state of the activity, while the methods `inProgress`, `hasBegun` and `isComplete` are used for measuring the overall state of the activity.

The directive of the activity is modeled by the abstract method `activate`, with different implementations for initiating and reacting activities. The general form that specific `activate` methods should take is:

```
void activate() {
  // mark activity as begun,
  // perform actions, and
  // mark activity as complete
}
```

Subclasses of activity may have methods that accept message requests from participants. Any methods that are added in a specific subclass of activity must conform to the following skeleton:

```
void method_Accept() {
  //  if activity in progress
  //  then perform actions
}
```

A method of any activity class can only be invoked if the activity is still performing – otherwise, the invocation will be flagged as an error.

The class `participant` is described as follows:

```
class Participant {
  public:
    Participant( Activity* a = NULL );

    void setOwner( Activity* a )
      { owner = a; }

    virtual ~Participant();

  protected:
    Activity* owner;
};
```

Concrete subclasses of participant will have methods that call the *accept*-methods of an activity. These are termed as *request*-methods, as they query an activity to ask it to perform a special action. The general form for request-methods is:

```
void method_Request( SomeActivity& a ) {
  a.method_Accept();
}
```

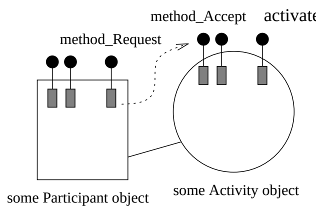


Figure 5: General Activity & Participant Object

In Figure 5, we illustrate a general activity object with the method method_Request, and a general participant object with the methods method_Accept and activate.

General Schematic Example. The activity object theActivity is described by SomeActivity, a subclass of Activity. In the method activate, we describe (as an example) the sequential execution of two part-activities pa1 and pa2. Note how theActivity is composed of the two part-activities.

```
class SomeActivity : public Activity {
  public:
    SomeActivity( Activity* a );

    virtual void activate();
    void F_Accept();

    virtual ~SomeActivity();

  private:
    PartActivity1* pa1;
    PartActivity2* pa2;
};

void SomeActivity::activate() {
  // mark activity as begun,
  // activate PartActivity pa1,
  // activate PartActivity pa2, and
  // mark activity as complete
}

void SomeActivity::F_Accept() {
  //  if activity in progress
  //  then perform actions
}

SomeActivity    theActivity;
```

The method activate models the lifecycle of theActivity. F_Accept models an operation available for theActivity to be invoked by a participant. An operation such as F_Accept will only execute if activate is under execution, i.e. the activity is inProgress.

In the activity classes PartActivity1 and PartActivity2, we describe various operations available for the activities, here exemplified by F1_Accept and F2_Accept. We also describe the activate method:

```
class PartActivity1 : public Activity {
  public:
    PartActivity1( Activity* a);

    virtual void activate() {
      // mark activity as begun,
      // call to a participant:
      aParticipant.F1_Request( this );
      // and mark activity as complete
    }

    void F1_Accept() {
      //  if activity in progress
      //  then perform actions
    }

    virtual ~PartActivity1();
};


class PartActivity2 : public Activity {
  public:
    PartActivity2( Activity* a);

    virtual void activate() { ... }

    void F2_Accept() { ... }

    virtual ~PartActivity2();
};
```

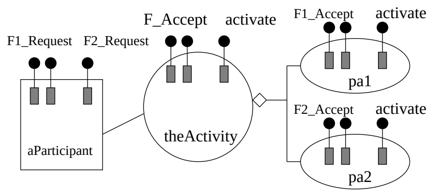


Figure 6: General Activity Example

In Figure 6, we illustrate a general activity `theActivity`, a participant `aParticipant`, and part-activities `pa1` and `pa2`.

The participant object `aParticipant` is described by `SomeParticipant`, which is a subclass of `Participant`. In `SomeParticipant`, we describe request-methods (`F1_Request` and `F2_Request`) which will be used to ask `aParticipant` to contribute to the execution of `pa1` and `pa2`. This is achieved by calling `pa1`'s accept-methods – `F1_Accept` and `F2_Accept`, which receive `aParticipant`'s requests to contribute.

```
  class SomeParticipant : public Participant {
    public:
      SomeParticipant( Activity* a );

      void F1_Request( PartActivity1& p ) {
        p.F1_Accept();    // call to an activity
      }

      void F2_Request( PartActivity2& p ) {
        p.F2_Accept();    // call to an activity
      }
  };

SomeParticipant aParticipant;

void main()
{
  // ...
  theActivity.activate();
}
```

Example: Card Game. The following simplified version of the card game example illustrates the use of the general classes `Activity` and `Participant` – a single round of a Five Hundred game will be modeled. An activity object `theRound`, comprises three part-activities: `theDealing`, `theBidding` and `theTricks`. Each of these part-activities respectively model the dealing, bidding and trick-taking phases of a round. The `theRound` activity is the parent of these three part-activities.

```
  class RoundActivity : public Activity {
    public:
      RoundActivity( Activity* a = NULL );

      virtual void activate();
      // ...
    private:
      DealingActivity* theDealing;
      BiddingActivity* theBidding;
      TrickTakingActivity* theTricks;
  };


  RoundActivity::RoundActivity
              ( Activity* a = NULL ) : Activity(a) {
    theDealing = new DealingActivity( this );
    theBidding = new BiddingActivity( this );
    theTricks = new TrickTakingActivity( this );
  }

RoundActivity theRound;
```

The four players (`thePlayers[4]`) are the participants in `theRound`. Each player has a number of operations which model the various contributions he/she can make to the game. In the example, we have the operation `bid_Request` which asks a player to make a bid. The bid is registered by invoking the `bid_Accept` method of the part-activity `theBidding`.

```
  class Player : public Participant {
    public:
      // ...
      void bid_Request( BiddingActivity& ba );

    private:
      Bid players_bid;
  };

  void Player::bid_Request( BiddingActivity& ba ) {
    // get bid from player and store in player_bid

    // Bidding activity accepts player's bid
    // and processes it
    ba.bid_Accept( this, player_bid );
  }

  Player thePlayers[4];
```

In Figure 7, we illustrate the activities, part-activities, and participants of this simplified example. In `theRound`, the `activate` method will sequentially invoke the part-activities `theDealing`, `theBidding` and `theTricks`. `theRound` is itself invoked in the main-program.

```
  virtual void RoundActivity::activate() {
    // mark activity as begun,
    // activate theDealing,
    // activate theBidding,
```

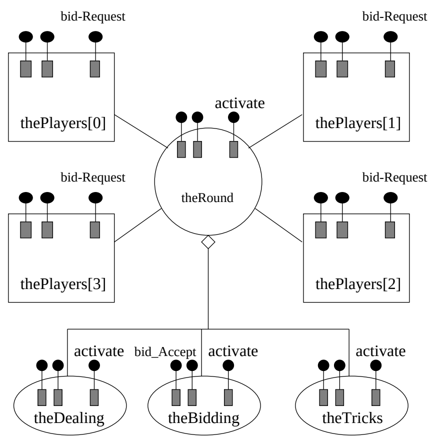


Figure 7: Example: Card Game

```
  // activate theTricks, and
  // mark activity as complete
}

void main()
{
  // ...
  theRound.activate();
}
```

3.1 Initiating Activities

An initiating activity will execute its `activate` method throughout its lifetime. This activity is set into progress by invoking this method – this method will determine when the activity must end and will then terminate itself.

Initiating Activity Class. The `activate` method of this class is structured thus:

```
class InitiatingActivity : public Activity {

  // ...
  virtual void activate() {
    beginActivity();
    // perform actions
    endActivity();
  }
};
```

The general form of an `accept` method is:

```
void method_Accept() {
  if ( inProgress() )
    // perform actions
  else
    // handle error!
}
```

We assume the existence of a `display` object that allows user interaction with the objects in our example. `display` is initialized and will then be passive – in contrast to the `activate` method of `theActivity`, which will actively execute:

```
void main()
{
  // ...
  display.initialize();
  theActivity.activate();
}
```

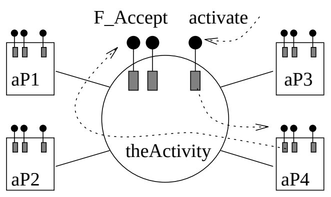


Figure 8: Initiating Activity

Initiating Schematic Example. In Figure 8 we illustrate the initiating activity `theActivity` and the participants `aP1` to `aP4`. `theActivity` executes `activate`, which at some point calls a method of `aP4`, which contributes to the activity by calling the `F_Accept` method of `theActivity`.

In `SomeActivity` we extend the description of `activate`. `activate` may start other part-activities such as `pa1` by executing `pa1.activate`:

```
class SomeActivity : public InitiatingActivity {
  // ...
};

void SomeActivity::activate() {
  beginActivity();
  pa1->activate();
  pa2->activate();
  endActivity();
}
```

In Figure 9, we illustrate an initiating activity `theActivity` and two part-activities `pa1` and `pa2`. When the `activate` method of `theActivity` is invoked, it will call the `activate` method of `pa1` at some point.

In `PartActivity1` we describe `F1_Request` and `activate`. The activity will check that the execution of an operation `F1_Request` is legal, and will register the execution of `F1_Request`:

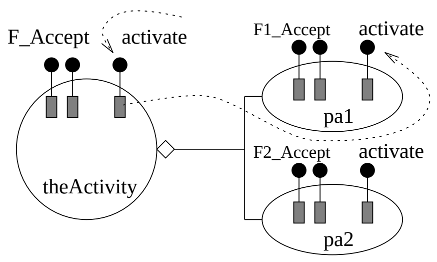


Figure 9: Initiating Activity and Part-Activities

```
class PartActivity1 : public InitiatingActivity {
  // ...
}

void PartActivity1::F1_Accept() {
  if ( inProgress() )
    // perform actions
  else
    // handle error!
}

void PartActivity1::activate() {
  beginActivity();
  // call to a participant
  aParticipant.F1_Request( this );
  endActivity();
}
```

`activate` may ask a participant to be active (by `aParticipant.F1_Request`) and to call one of the methods (either a specific method or just anyone), for example `F1_Accept`, of the activity.

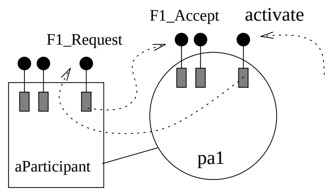


Figure 10: Initiating Activity and Participant

In Figure 10 we illustrate an initiating part-activity `pa1` and a participant `aParticipant`. The activity `pa1` executes its `activate`, which calls `aParticipant`'s `F1_Request` method, which contributes by calling the `F1_Accept` method of `pa1`.

Example: Initiating Card Game. In a `RoundActivity`, the `activate` method will sequentially activate the part-activities `theDealing`, `theBidding`, and `theTricks`:

```
virtual void RoundActivity::activate() {
  beginActivity();

  theDealing.activate();
  theBidding.activate();
  theTricks.activate();

  endActivity();
}
```

The `BiddingActivity` has an operation `bid_Accept` available to the players for placing their bids. The lifecycle of `BiddingActivity` is repeatedly to find and ask the next player to bid until the bidding is over according to the rules of the card game.

```
class BiddingActivity : public InitiatingActivity {
    // ...
    virtual void activate();
    void bid_Accept( Player& p, Bid& b );
  };

  void BiddingActivity::
       bid_Accept( Player& p, Bid& b) {
    if ( inProgress() ) {
      // validate bid "b" made by player "p"
      // if this is the highest bid,
      // record it & the player who made it
      // set "winnerFound" to TRUE
    } else
      // handle error!
  }

  void BiddingActivity::activate() {
    Player& next_player;

    beginActivity();
    do {
      // get next player and store in "next_player"
      // then call the "next_player" participant

      next_player.bid_Request( this );
    } while ( ! winner_found );
    endActivity();
  }
```

The passive `display` object is activated by the `Player` participant to prompt the user to enter a bid.

```
class Player : public Participant {
  // ...
  void bid_Request( BiddingActivity& ba );
};

void Player::bid_Request( BiddingActivity& ba ) {
  Bid new_bid;
```

```
    display << "Give me some bid";
    display >> new_bid;

    // call to activity
    ba.bid_Accept( new_bid );
  }
```

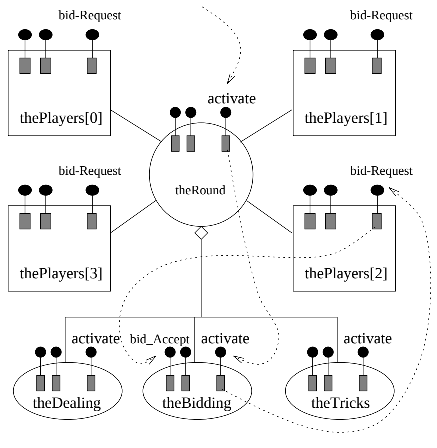


Figure 11: Example: Card Game `RoundActivity`

In Figure 11, we illustrate a calling sequence in the `RoundActivity` example. `theRound` activity has called the part-activity `theBidding`, which in turn calls `bid_Request` of `thePlayers[2]`. This participant contributes to the bidding activity by invoking the `bid_Accept` method of `theBidding`. After each bid is made, `theBidding` may call the next player to contribute to the bidding process.

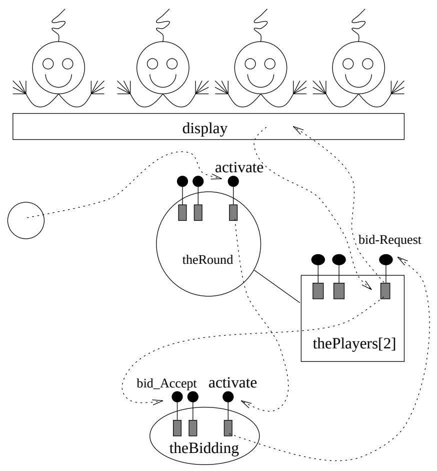


Figure 12: Example: Display and Users

In Figure 12, we illustrate how the `activate` method of `theRound` is invoked from the main program. The `bid_Request` method will get a `new_bid` from the user via the `display` before invoking the `bid_Accept` method of `theBidding`.

3.2 Reacting Activities

A reacting activity will have its `activate` method executed whenever an action in relation to the activity has taken place. It is executed initially and then successively as a side-effect of the execution of an accept-method. Finally it will determine the completion of the activity and terminate itself.

Reacting Activity Class. The `activate` method of the `ReactingActivity` class is specified as:

```
class ReactingActivity : public Activity {
  // ...
};

void ReactingActivity::activate() {
  if ( parent != NULL)
    parent->activate();
}
```

Methods of reacting activities will take the following form (notice the call to `activate` after a method has completed its actions):

```
void method_Accept() {
  if ( inProgress() ) {
    // perform actions
    activate();
  } else
    // handle error!
}
```

In subclasses of `ReactingActivity`, the `activate` method should be structured as follows:

```
void activate() {
  if ( first_time ) {
    beginActivity();
    first_time = FALSE;
  }
  if ( inProgress() ) {
    // perform actions
    // when finished
      endActivity();

    // perform default action
    ReactingActivity::activate();
  } else
    // handle error!
}
```

The initial test for `first_time` ensures execution of the `beginActivity` method only once. After the activity has completed its actions, `endActivity` is invoked. Following this, the activity invokes its parent's `activate` method, propagating the activation up the inheritance hierarchy.

The activity is passive; `activate` is only executed when a participant has called one of its methods (e.g. `F_Accept`) or there has been some activity in a part-activity.

The `activate` method of `theActivity` is invoked as an initialization. `theActivity` will then be passive whereas `display` will be actively executing:

```
void main()
{
  // ...
  display.initialize();
  theActivity.activate();
  display.execute();
}
```

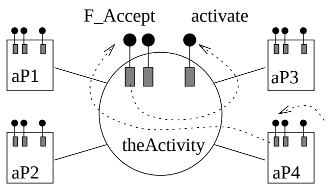


Figure 13: Reacting Activity

Reacting Schematic Example. In Figure 13 we illustrate the reacting activity `theActivity` and the participants `aP1` to `aP4`. The execution of a call a method of `aP4`, contributes by a call of the `F_Accept` method of `theActivity`, which then executes `activate` as a side-effect.

In `SomeActivity` we extend the description of `activate`. `activate` assumes that `beginActivity` has already been executed to set `hasBegun` to `True`. When the activity is over, `activate` will end and set `isComplete` to `True`. `activate` may start other part-activities such as `pa1` by executing `pa1.beginActivity`:

```
class SomeActivity : public ReactingActivity {
  // ...
};

void SomeActivity::activate() {
  if ( first_time ) {
    beginActivity();
    first_time = FALSE;
  }
  if ( inProgress() ) {
    sequence = PART_ACTIVITY1;
    switch( sequence ) {
      case PART_ACTIVITY1 :
        if ( pa1.hasBegun() == FALSE )
          pa1.activate();
        if ( pa1.isComplete() == FALSE ) {
          sequence = PART_ACTIVITY2;
          break;
        }
      case PART_ACTIVITY2 :
        if ( pa2.hasBegun() == FALSE )
          pa2.activate();
        if ( pa2.isComplete() == FALSE ) {
          sequence = END_ACTIVITY;
          break;
        }
      case END_ACTIVITY   : endActivity();
        sequence = NO_ACTIVITIES;
        break;
      case NO_ACTIVITIES  : break;
    }
    // default behaviour
    ReactingActivity::activate();
  } else
    // handle error!
}
```

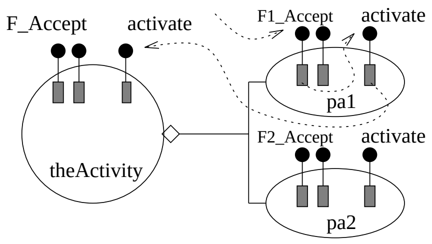


Figure 14: Reacting Activity and Part-Activities

In Figure 14 we illustrate a reacting activity `theActivity` and two part-activities `pa1` and `pa2`. The execution of the `activate` method of the part-activity `pa1` as a side-effect includes the execution of the `activate` method of `theActivity`.

In `PartActivity1` we describe `F1_Request` and `activate`. The activity will check that the execution of an operation `F1_Request` is legal, and will register the execution of `F1_Request`:

```
class PartActivity1 : public ReactingActivity {
  // ...
}

void PartActivity1::F1_Accept() {
  if ( inProgress() ) {
    // perform actions
    activate();
  } else
    // handle error!
}

void PartActivity1::activate() {
  if ( first_time ) {
    beginActivity();
    first_time = FALSE;
  }
  if ( inProgress() ) {
    // perform actions
    // when finished
      endActivity();

    // perform default action
    ReactingActivity::activate();
  } else
    // handle error!
}
```

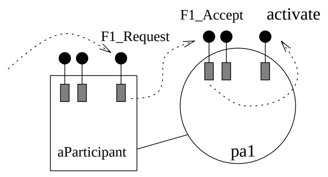


Figure 15: Reacting Activity and Participant

In Figure 15 we illustrate a reacting part-activity `pa1` and a participant `aParticipant`. The execution of `aParticipant`'s `F1_Request` method contributes by calling the `F1_Accept` method of `pa1` and as a side-effect `pa1` executes its `activate` method.

Once again, we assume the existence of a `display` object – in this scenario, `display` receives input from the user and calls the `request` method of `aParticipant`.

Example: Reacting Card Game. The object `theRound` is a round of a card game. Its `activate` method will be invoked each time any of its methods – or the part-activities `theDealing`, `theBidding`, `theTricks` – are invoked by a player. `activate` will check the state of `theRound`, especially the state of the part-activity in progress, and control the sequential execution of these part-activities. Finally, it will mark the end of its own execution.

```
virtual void RoundActivity::activate() {
  if ( first_time ) {
    beginActivity();
    first_time = FALSE;
  }
  if ( inProgress() ) {
    sequence = DEAL_ACTIVITY;
    switch( sequence ) {
      case DEAL_ACTIVITY :
        if ( theDealing.hasBegun() == FALSE )
          theDealing.activate();
        if ( theDealing.isComplete() == FALSE ) {
          sequence = BID_ACTIVITY;
          break;
        }
      case BID_ACTIVITY :
        if ( theBidding.hasBegun() == FALSE )
          theBidding.activate();
        if ( theBidding.isComplete() == FALSE ) {
          sequence = TRICK_ACTIVITY;
          break;
        }
      case TRICK_ACTIVITY :
        if ( theTricks.hasBegun() == FALSE )
          theTricks.activate();
        if ( theTricks.isComplete() == FALSE ) {
          sequence = END_ACTIVITY;
          break;
        }
      case END_ACTIVITY   : endActivity();
        sequence = NO_ACTIVITIES;
        break;
      case NO_ACTIVITIES  : break;
    }
    // default behaviour
    ReactingActivity::activate();
  } else
    // handle error!
}
```

The `BiddingActivity` has an operation `bid_Accept` available to the players for placing their bids. The lifecycle of `BiddingActivity` is only to check if the bidding is over according to the rules of the card game, each time a `set Bid` operation has been invoked by a player.

```
void BiddingActivity::
     bid_Accept( Player& p, Bid& b) {
  if ( inProgress() ) {
    // validate bid "b" made by player "p"
    // if this is the highest bid,
    // record it & the player who made it
    // set "winnerFound" to TRUE
    activate();
  } else
    // handle error!
}

void BiddingActivity::activate() {
  // ...
}
```

A `display` object exists – it actively solicits user interaction, repeatedly invoking the appropriate operations of the player objects. For example: `thePlayer[2].bid_Request( TheBA, the_bid );`.

```
  void Player::
       bid_Request( BiddingActivity& ba, Bid& b ) {
    // call to activity
    ba.bid_Accept( b );
  }
```

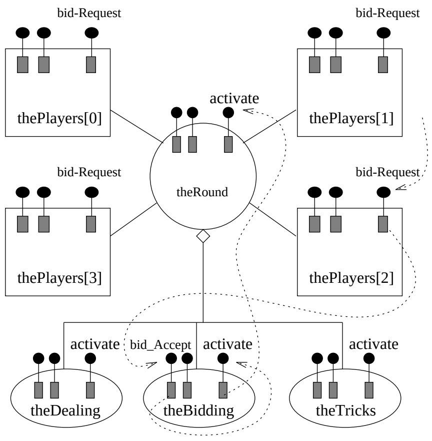


Figure 16: Example: Card Game `RoundActivity`

In Figure 16 we illustrate a calling sequence in the card game example. The call of `bid_Request` of `thePlayers[2]` implies the call of the `bid_Accept` method of `theBidding`, and as a side-effect, the `activate` methods of `theBidding` and `theRound` are executed in turn. The `bid_Request` of another player may be called to contribute to the bidding process.

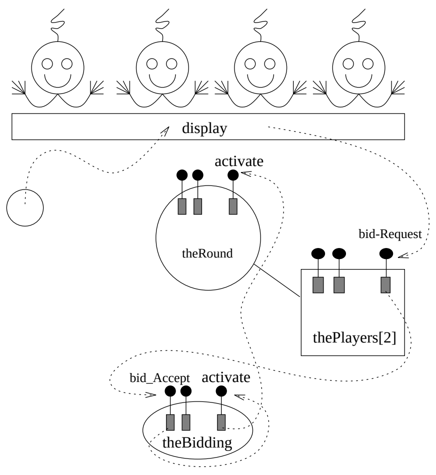


Figure 17: Example: Display and Users

In Figure 17, we illustrate how the `display` is invoked from the main program. As the result of some interaction with a user the display will invoke the `bid_Request` and supply some `new_bid`.

4 The Experimental Project

The activity abstraction was invested in a project described in [May 94]. The project's objective was to explore issues related to the design and construction of object-oriented frameworks – the C++ language was used to build software artefacts through the course of the project.

The problem domain that the study concentrated on 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 (to exercise the framework).

More importantly, the project highlighted important issues of software development. Specifically:

- Framework architecture: How abstract/concrete should a framework be? More abstract frameworks offer increased flexibility at the cost of additional time required for specialisation. Conversely, more concrete frameworks can save time by making assumptions that could also restrict application flexibility.

- Integration of Frameworks with Different Platforms: How should a framework be constructed so as to work seamlessly as possible on different platforms? The challenge is to design an architecture that allows rich interaction with the user without embedding platform-dependent user interface details.

- Activities: How can more complicated sequences of interaction/process be represented in such a way as to simplify their complexity? The present paper has dealt with an approach to abstracting interaction between entities. A framework for activities was created, which later became the basis for a card game framework – it was eventually used to create the Five Hundred game.

The study furnished an opportunity to experience framework construction, allowing scrutiny of insights and hurdles that were eventually encountered. The time limitation of the project did not allow sufficient refactoring of the framework – redundant features could be eliminated, while installing further functionality. This illustrated the need for additional time to iterate a framework through successive versions.

Limitations were encountered using C++. In its "standard" form, the language lacks concurrency and sequencing mechanisms that could have been useful in implementing activities. Additionally, the lack of standard, portable class libraries impinged on the framework construction process.

Overall, the project demonstrated that there are substantive aspects of software development that still require attention. There remains much more to understand about devising software structures – techniques to enhance our design expressivity can be further improved, as well as the tools/languages that realize our models.

5 Summary

The underlying assumption in this paper is that in existing object-oriented methodologies and languages, classes and objects appear as isolated elements with an implicit and poor description of the interaction structure between them. Activities present a different type of abstraction which may be used to model such interaction structures. As such, they are important for the modeling of organization and cooperation of classes and objects. We have described abstract classes in C++ to support the use of activities in this language.

Given the similarity of C++ and other mainstream object-oriented languages with respect to the restricted set of object-oriented programming language constructs used in our implementation of activities in C++, translation to other languages is a straightforward process.

Results and Restrictions. The main results may be summarized as follows:

- Activities support the modeling of the organization and cooperation of objects in object-oriented programming.
- 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).
- If implemented successfully, activities offer an orthogonal solution to expressing and manipulating interaction.

The present paper does not discuss the sequencing of multiple activities, which concurrently overlap in execution. We also have not discussed possible approaches for the dynamic inclusion/exclusion of participants in an activity – our examples show participants whose membership in an activity is static.

These restrictions are primarily due to lack of mainstream language support for such capabilities.

Related Work. *Contracts* [Helm et al. 90] are specifications of behavioral dependencies amongst cooperating objects. Contracts are object–external abstractions and include invariants to be maintained by the cooperating objects. The focus is on inter–object dependencies to make this 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 modeling of real world phenomena and their inter-dependencies. Instead the intention is to have a mathematical, centralized description, that supports provable properties.

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

- **Overlapping activities**: Often, the activities we wish model will not follow neat, consecutive parcels of execution. They may overlap and merge into each other at different times – we wish to seek methods of expressing such scenarios in our designs.
- **Contexts**: In non-trivial systems, activities will generally refer to a central context that represents common information from which these activities will draw. Part-activities will not generally execute in isolation or ignorance of their

super-activities; they will usually serve the purposes of their parent activity and/or draw information from the context which they share.

Fundamentally, contexts pose a problem for encapsulation of activities – the challenge is to devise techniques for modeling such relationships between activities; perhaps formalising some standard form of protocol for inter-activity communication

- **Existence of activities**: It should be possible to represent the circumstances in which an activity can be said to exist. Some activities may have no reasonable existence outside certain settings – for example, given the Five Hundred game, there is a phase of activity called bidding at the commencement of the game. We explicitly created a class to represent this activity and instantiated such an object at the appropriate point in the game.

  However, a more accurate view of the activity would be to say that the activity of bidding is "not defined" after a given point in the game. Any attempt to instantiate such an activity would result in failure – this is a more faithful model of the card game; there is no such concept as bidding at later stages of the game. Our present implementation allows a bidding activity to be created outside of its logical scope.

- **Concurrency issues**: If activities are to attain a powerfully expressive level of use, concurrency mechanisms that can facilitate the interactive communication required of activities need to be devised.

  Activities should possess the capability to block execution and continue, contingent on the action of other activities; a permission protocol should also be specified in order to impose a hierarchy of activation procedures – it should be possible to define different sets of rules regulating the control/invocation of activities by other activities.

Acknowledgments. Daniel May would like to thank Deeran Peethamparam for proofreading portions of the paper and discussion of its issues. Special thanks should also go to Penny Rush, for making it possible for him to complete the paper.

References

[Helm et al. 90] R.Helm, I.M.Holland, D.Gangopadhyay: Contracts: Specifying Behavioral Compositions in Object–oriented Systems. Proceedings of the European Conference on Object–Oriented Programming / Object-Oriented Systems, Languages and Applications Conference, 1990.

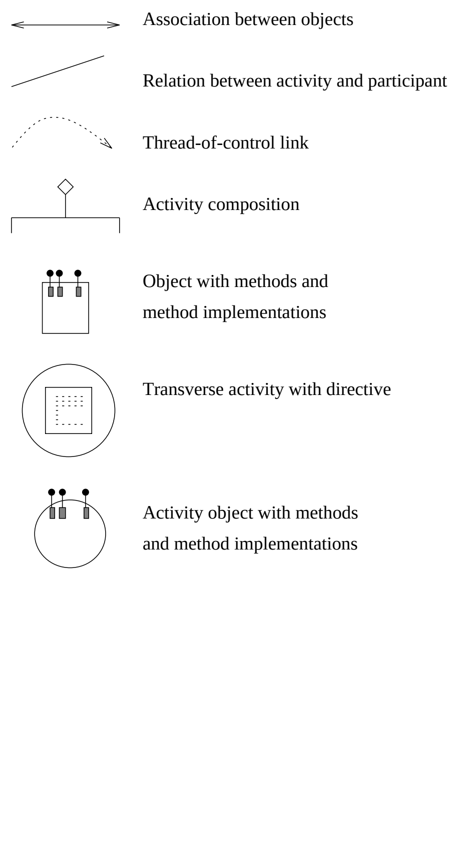
[Kristensen 93a] B.B.Kristensen: Transverse Classes & Objects in Object-Oriented Analysis, Design, and Implementation. Journal of Object-Oriented Programming, 1993.

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

[May 94] D.C.M.May: Frameworks: An Excursion into Metalevel Design and Other Discourses. Department of Computer Science, Monash University, 1994.

[Stroustrup 91] B.Stroustrup: The C++ Programming Language. 2/E, Addison-Wesley 1991.

Notation Summary


Association between objects
Relation between activity and participant
Thread-of-control link
Activity composition
Object with methods and method implementations
Transverse activity with directive
Activity object with methods and method implementations