



Norwegian University of
Science and Technology

Towards Modeling of Data in UML Activities with the SPACE Method

An Example-Driven Discussion

Nina Heitmann

Master of Science in Communication Technology

Submission date: June 2008

Supervisor: Peter Herrmann, ITEM

Co-supervisor: Frank Alexander Kraemer, ITEM

Problem Description

The SPACE Method is dedicated to model reactive systems with an emphasis on collaborative building blocks that are expressed as UML activities. Currently, focus lies on the description of control flows that do not carry any data. For the development of real systems, however, it is necessary to also describe how data is transferred between components. While UML provides several elements for activities to express data, it is not clear how these elements should be used within the SPACE method and which trade-offs have to be made when introducing detailed data handling.

In this thesis, the requirements for data handling should be studied based on a specification of the card game UNO. It should be described how typical situations with regards to data should be solved and how Arctis, the tool supporting SPACE, can be extended to handle these situations.

Assignment given: 14. January 2008
Supervisor: Peter Herrmann, ITEM

Abstract

The focus of this work is the rapid engineering method SPACE, developed at NTNU. In this method, services are modeled using UML 2.0 collaborations and activities, and from these executable code can be generated. Services can be composed from other services and building blocks. Until recently, SPACE has only focused on control flow. We have extended SPACE by introducing data flow modeling into SPACE activities. This raises some important questions, for example, how data between building blocks may be shared. We discuss a number of possible solutions. Our work is driven by the UNO card game as an example application. The structure and behavior of the UNO card game is analyzed and discussed, highlighting and exemplifying the aspects discussed in this work.

Preface

This master thesis is the final part of a Master of Science degree from the Department of Telematics (ITEM) at the Norwegian University of Science and Technology (NTNU).

I would like to use this opportunity to thank my supervisor Frank Alexander Kraemer for always being available for assistance. His help and guidance has been central in this work. I would also like to thank Professor Peter Herrmann, who is academically responsible, for valuable input.

Lastly, I would like to thank Stein Magnus Jodal for proofreading my work.

Trondheim, June 26, 2008

Nina Heitmann

Contents

1	Introduction	1
1.1	Contribution	2
1.2	Outline	2
2	Background	5
2.1	SPACE	5
2.1.1	Arctis and Ramses	6
2.2	UML	7
2.2.1	Collaborations	7
2.2.2	Activities	9
2.3	Extensions of UML by SPACE	14
2.3.1	Multi-Session Collaborations	14
2.3.2	Waiting Decision Nodes	15
2.4	Current integration of Data in SPACE and Arctis	16
2.5	UNO	17
2.5.1	Object of the game	17
2.5.2	Starting the game	17
2.5.3	Playing the game	18
2.5.4	Action cards	20
2.5.5	Jump-in Rule	21
2.5.6	Special properties of the mobile version	22
3	Specification of UNO — Part 1: Object and Collaboration Structure	23
3.1	Requirements Capture	23
3.1.1	Starting the Game	23
3.1.2	Playing the Game	24
3.1.3	Winning the Game	25
3.2	Object-Oriented Analysis	25
3.3	Object Attributes	27

3.4	Collaboration analysis	30
4	Introducing Data into SPACE	35
4.1	Input and Output Parameters to Actions	35
4.2	Setting and reading variables	36
4.3	Several Objects in Same Flow	37
4.4	Transforming Types between Object Nodes	38
4.5	Output Pins for Accept Signal Action	39
4.6	Input Pins for Send Signal Actions	40
4.7	Fork Node with both Object and Control Flow	40
4.8	Decision Nodes	42
5	Specification Guidelines	45
5.1	Active and Passive Objects	45
5.2	Modeling Issues in Regard to Variable Access and Clearness	46
5.2.1	Alternative 1: Shared data between Building Blocks .	49
5.2.2	Alternative 2: A Flat Specification	51
5.2.3	Alternative 3: More Comprehensive Call Operation Actions	52
5.2.4	Alternative 4: Providing Variables as Input Paramete- rs to Building Blocks	55
5.2.5	Alternative 5: Typed Collaboration Roles	56
5.2.6	Discussion	61
5.2.7	Conclusion	62
6	Specification of UNO — Part 2: Behavior	65
6.1	System View	65
6.2	Collaboration <i>Setup</i>	67
6.3	Collaboration <i>Playing</i>	70
6.3.1	Collaboration <i>Make Move</i>	72
6.3.2	Collaboration <i>Game Updates</i>	74
6.3.3	Collaboration <i>Draw Card</i>	75
6.3.4	Collaboration <i>Color Dialog</i>	76
6.3.5	Collaboration <i>Turn Pile</i>	77
6.3.6	Building Block <i>Player Input</i>	77
6.3.7	Building Block <i>Validate Move</i>	80
6.4	Collaboration <i>End</i>	81
6.5	Simplifications	82
7	Discussion	85
8	Conclusions and Future Work	87
8.1	Conclusions	87
8.2	Future work	87

CONTENTS

vii

Bibliography

89

List of Figures

2.1	SPACE	6
2.2	Sketch of SPACE and its tool support	7
2.3	Collaboration draw card	8
2.4	Collaboration Playing with collaboration use draw card	8
2.5	Activity <i>Draw Card</i>	9
2.6	Action <i>Draw random card</i>	10
2.7	Special action notations	10
2.8	Collaboration with activity parameter nodes	12
2.9	Example of action with an output pin	12
2.10	Control flow	13
2.11	Object flow	13
2.12	Example use of select	15
2.13	EBFN for select and exists	15
2.14	Example use of a waiting decision node	16
2.15	UNO game	18
2.16	Action cards	20
3.1	System view	26
3.2	System view with controller unit	27
3.3	System view with the controller included in the discard pile	27
3.4	System collaboration	30
3.5	Setup collaboration	31
3.6	The playing collaboration	32
3.7	The <i>End</i> collaboration	32
4.1	Sum action and corresponding Java code	36
4.2	AddVariableValueAction notation	36
4.3	ReadVariableAction notation	36
4.4	Counter	37
4.5	Merge action	38

4.6	Move class	38
4.7	Transformation notation in UML	39
4.8	Transformation notation in Arctis	39
4.9	Alternative methods for extracting info from received signal	40
4.10	Send signal action with input pin	40
4.11	Screenshot of the <i>DrawCard</i> service designed in Arctis	41
4.12	Examples of valid and invalid fork nodes	42
4.13	Example of decision node with object flow	43
4.14	Simple counter	43
5.1	Simplified Playing activity in Arctis	47
5.2	Building block Validate Move in Arctis	49
5.3	Screenshot of playing activity with building blocks	50
5.4	Solution with flat specification	51
5.5	Screenshot of call operations actions	53
5.6	Building block with variables provided as input parameters	55
5.7	Building block <i>Validate Move</i>	56
5.8	Collaboration Make Move	56
5.9	Activity diagram for collaboration Make Move	57
5.10	Interface implementation	58
5.11	Activity diagram for collaboration Playing	58
5.12	Multiple interface implementation	59
5.13	System activity using alternative 5	60
5.14	Inheritance of building block	61
6.1	System activity	66
6.2	Activity Setup	68
6.3	Activity <i>Distribute Players</i>	69
6.4	Activity <i>Select Top Card</i>	69
6.5	Activity <i>Select Turn</i>	69
6.6	Activity <i>Deal</i>	70
6.7	Playing activity	71
6.8	Collaboration <i>Make Move</i>	72
6.9	ESM <i>Make Move, Player</i> partition	73
6.10	ESM <i>Make Move, Discard Pile</i> partition	73
6.11	Activity Game Updates	74
6.12	Activity diagram draw card	75
6.13	Building block <i>Counter</i>	76
6.14	Activity <i>Color Dialog</i>	76
6.15	Activity <i>Turn Pile</i>	77
6.16	Activity <i>Player Input</i>	78
6.17	Input blocks used in the <i>Player Input</i> building block	79
6.18	Building block Validate Move	80
6.19	Activity <i>End</i>	81

LIST OF FIGURES

xi

6.20 Sub-collaborations of the *End* collaboration 82

Chapter 1

Introduction

Developing advanced telecommunication services can be a difficult and time consuming task. At the same time, rapid provisioning of services is important to meet customer demands. We want it to be easier for a designer to develop advanced and high quality services in shorter time.

The quality of a telecommunication system is to a large extent determined by its behavior. But behavior is difficult to describe due to its dynamic and transient nature. A reason for this may be a problem in service engineering known as the cross-cutting nature of services. This means that a service involves several objects, but the behavior is described object for object, for example using state machines. Getting a grasp of the complete behavior of the service is difficult as the specification focuses only on behavior in objects, not the interaction between objects. This leads to the collaboration-oriented approach, which can describe the complete behavior of services. Collaborations makes it possible to describe a complete behavior between a set of objects or components in isolation.

Another important factor in rapid service creation is reuse. Collaborations works very well for this purpose, as services can be composed from sub-services, instead of from components.

In addition to the collaborations that describe the complete behavior between objects, we also need models of the components in isolation, as it is the components that will have to be created and deployed to realize the system. However, keeping the diagrams consistent is a challenge and it takes unnecessary time. A better solution is to let developers create only one set of diagrams, and generate the other diagrams automatically from these. By describing the behavior using collaborations and actions, the state diagrams may be found by analyzing the actions. From the state diagrams it is possible with automatic generation of code that can be deployed on the different devices realizing the system.

SPACE is an engineering method for rapid creating of services, devel-

oped at NTNU. The method is based on three principles: Reusable collaborative building blocks, model transformation and code generation, and formal analysis of models. SPACE is supported by tools that help in the engineering process [17].

Telecommunication systems are examples of reactive systems, which are systems whose role is to maintain an ongoing interaction with their environment, rather than produce some final value upon termination [1]. It is a well known fact that the behavior of such systems, even small ones, may be very hard to analyze and understand. With the introduction of state machines for modeling and analyzing such systems, the situation was improved, but due to the fact that state machines model the behavior of one component in separation, the overall behavior was still difficult to understand. The collaboration-oriented approach, which is an important part of SPACE, provides a solution to this problem, as the overall behavior between components can be described a single diagram.

1.1 Contribution

Until now, SPACE has focused on modeling of control flows. We have introduced data into SPACE, which makes it possible to specify data-intensive systems in a complete way in SPACE. In almost any telecommunication system data is central. Without data, the system can not fulfill its purpose. With data, we mean spatial information that exists over time, like color of a card or name of a player, in opposition to control data, or only control, which is information about behavior progress, like states [9].

When modeling system with data, more complexity are introduced, as data representation requires extra UML elements. Structuring this UML elements in a clear, elegant way is challenging. We will show that this requires data sharing between collaborations and building blocks. Doing this in a clear, intuitive way that maintain the principle of reuse and conform to the UML standard is not easy. We provide several possible solutions to this problem.

The focus of this work is data in SPACE activities. The interface descriptions of collaborative building blocks, so-called External state machines (ESMs), are not discussed, as they are not concerned with data. Also note that the UNO game is not the primary focus, it is just used as a basis for our discussion to identify requirements on data modeling motivated by a rather complex example.

1.2 Outline

The rest of this report is organized as follows.

Chapter 2 introduces the UNO game and the game rules. Further it gives a background on the SPACE method and the SPACE tools. We then presents UML and the UML diagrams central in SPACE.

Chapter 3 describes the object and collaboration structure of UNO, and the important data attributes are discussed.

Chapter 4 introduces data into SPACE activities, which is necessary to specify the UNO behavior.

Chapter 5 discuss guidelines for making good choices when specifying a system, and discuss solutions for sharing data between building blocks and collaborations.

Chapter 6 describe the UNO behavior using UML activities.

Chapter 7 discuss and evaluates the work.

Chapter 8 concludes the work and suggests future work.

Background

In this chapter the engineering method SPACE and its tools Arctis and Ramses are presented. A description of the UML Collaboration and Activity diagrams is given, together with a description of how SPACE has extended UML. This chapter ends with a description of the UNO game and the game rules.

2.1 SPACE

SPACE is an engineering method for reactive systems that support rapid creation of services [17]. In this approach the specifications of services is done in terms of UML 2 collaborations, activities and external state machines. Collaborations express the structural properties of the system at high level of abstraction, like participants and their multiplicity. Activities express the complete behavior of collaborations, both local behavior and interactions between system participants [14]. External state machines define the externally visible behavior of building blocks [17].

The approach is outlined in Fig. 2.1. During service design, an engineer may consult the library for reusable building blocks, as a new service often can be composed from existing building blocks that may be adapted to the new service [15]. Services may also be designed from scratch, and the services may be composed with each other to form new services. The service specifications are the only manual work in the engineering process, the rest of the process is automated. The service specifications are transformed automatically into executable state machines and composite structures using model transformation. The state machines and structures are further transformed into executable code [16].

A crucial point when automatically transforming models into code is to ensure correctness of both the models and the transformation. Thus, a formal way to define the semantics of activities and state machines is needed.

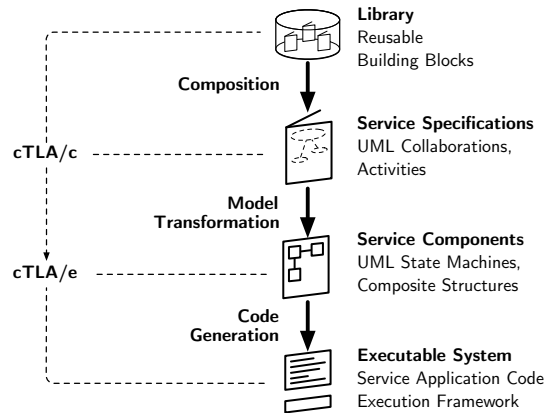


Figure 2.1: SPACE (from [16])

For this purpose, Herrmann’s framework compositional Temporal Logic of Actions (cTLA) is used, as discussed in [16]. The engineering approach shown in Fig. 2.1 is complemented by two variants of cTLA; cTLA/c and cTLA/e, shown on the left side of the figure. cTLA/c formalizes the collaborative service specifications given by UML 2.0 activities. As shown in Fig. 2.1, cTLA/c is used in the process of transforming collaborations and activities to state machines. The other variant, cTLA/e, is used to formalize the behavior of the state machines, and to transform state machines into executable code.

Two SPACE tool sets exist, Arctis and Ramses, both offered as plug-ins to Eclipse [11]. Eclipse is a Java-based open source development platform, with a large number of extension or plug-ins.

2.1.1 Arctis and Ramses

Fig. 2.2 shows how Arctis and Ramses support SPACE. Arctis provides functionality for manual editing and specification of collaborations and their behavior. Syntactic inspectors and a TLC model checker ensures a consistent specification. A model transformer translates a consistent specification into UML state machines and components [17], which can be used by Ramses to generate code. Ramses can also be used for modeling of services using state machines [2], but since these state machines are generated by Arctis, only the code generation of Ramses is used.

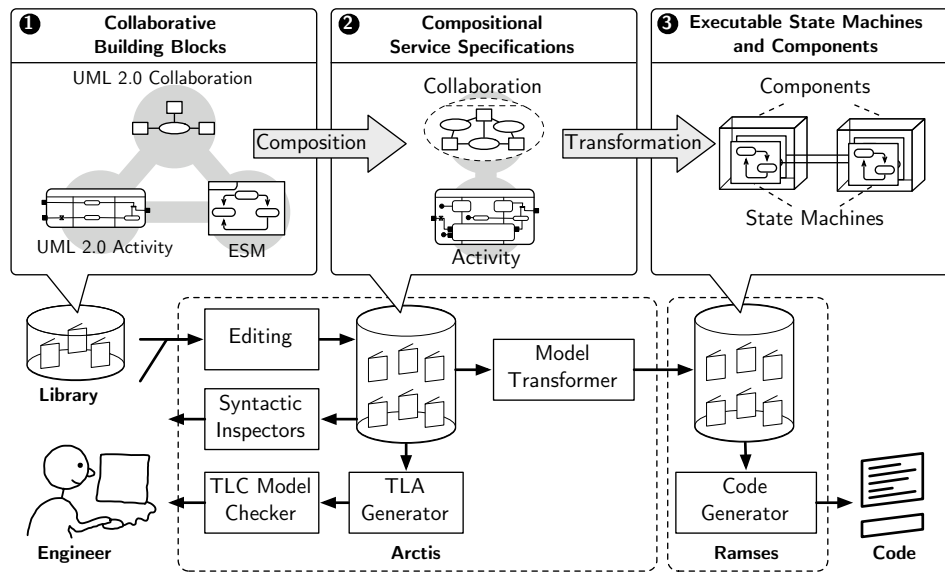


Figure 2.2: A coarse sketch of the SPACE engineering approach and its tool support (from [17])

2.2 UML

UML is a general-purpose visual modeling language defined by the Object Management Group [13]. It is a standardized specification language mainly used in object oriented software design. UML is used for all stages in the development process. Three different views may be expressed using UML: functional behavior view, static structural view, and dynamic behavior view [21].

In SPACE we focus on dynamic behavior view. The next subsections gives an introduction to collaboration and activity diagrams.

2.2.1 Collaborations

A collaboration is a specification of a contextual relationship among instances that interact within a context to implement a desired functionality [21]. A collaboration explains how a set of objects work together to carry out a particular purpose in ways that are unique to the particular situation. A collaboration consists of roles, which are descriptions of a participant in an interaction. A connector is a relationship between two roles within a particular collaboration [21].

UNO consists of several sub-services. One of these are the service *Draw Card*, in which a player can draw a number of UML cards from a draw pile. Fig. 2.3 shows this service as a UML 2.0 collaboration. Participants in

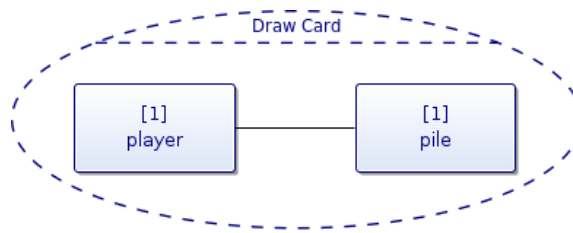


Figure 2.3: Collaboration draw card

the service are represented by collaboration roles *player* and *pile*. Both the *player* and the *pile* has a default multiplicity of 1, which means that this collaboration models the interaction between one *player* and the *pile*.

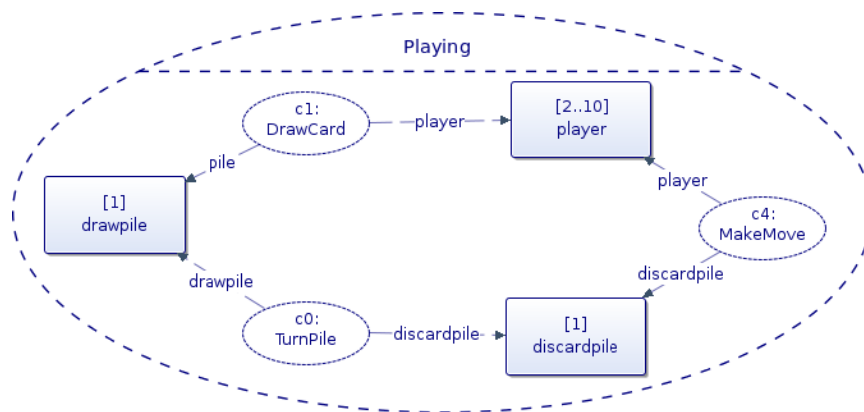


Figure 2.4: Collaboration Playing with collaboration use draw card

Fig. 2.4 shows a collaboration use of the *Draw Card* collaboration used in a composite collaboration *Playing*. The instance *c1* of the *Draw Card* collaboration is in UML called a collaboration use. In this example, the *player* also has two other collaboration uses, *c0* and *c4*. Each collaboration use is notated by a dashed ellipse containing the name of the collaboration use and the collaboration type, separated by a colon. The *player* role in the *Draw Card* collaboration is bound to the *player* element in the *Playing* collaboration, notated by a dashed line labeled by the name of the role in the collaboration use. Similarly for the *pile* role. Note that the *player* role in the *Playing* collaboration has a multiplicity of 2-8, which means that several *players* may interact with the *draw pile* with a *Draw Card* collaboration at a time.

2.2.2 Activities

A collaboration may have behavior attached, for example state machines, sequence diagrams or activities. In SPACE, activities are used. Activities focus on the sequence, conditions, and inputs and outputs for invoking other behaviors [4]. Activities use an intuitive token flow semantics inspired by Petri nets, where “token” is just a general term for control and data values. UML 2.0 activities define a virtual machine based on routing of control and data through a graph of nodes connected by edges. UML 2.0 activities contain nodes connected by edges to form a complete flow graph. Control and data values flow along the edges and are operated on by the nodes, routed to other nodes, or stored temporarily [3]. The activity in Fig. 2.5 shows the behavior of the collaboration shown in Fig. 2.3. The behavior details will be explained later in this work.

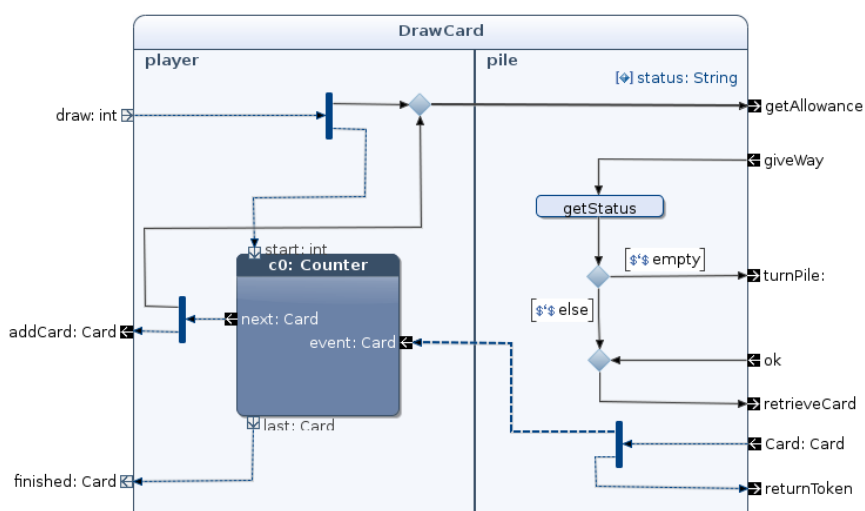


Figure 2.5: Activity *Draw Card*

Activity Nodes

There are three types of activity nodes: action nodes, control nodes, and object nodes. Action nodes operate on control and data tokens they receive via edges of the graph, and provide control and data tokens to other actions. Control nodes route control and data tokens through the graph. Object nodes hold data temporarily as they wait to move through the graph [5].

Action Nodes An action is the smallest unit of computation that can be expressed in UML. An action is an activity node that does something to the

state of the system or extracts information from it [21]. Fig. 2.6 shows an example of such an action. The action draws a random card from the draw pile.

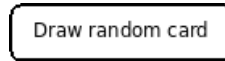


Figure 2.6: Action *Draw random card*

Actions are predefined in UML, whereas behaviors and operations are user-defined [4]. This means that actions are not behavior themselves. For example, in Fig. 2.6 the round-cornered rectangle is an action that invoke the user-defined behavior *draw random card*. Actions are the only objects that can query objects, make changes to objects, invoke operations owned by objects and invoke behaviors. This means all behaviors must contain actions to have any effect on objects.

There are a number of different types of actions. Most actions are drawn as a rectangle with rounded corners, as shown in Fig. 2.6. Some communication actions have special notations. These include the accept signal action, send signal action, and accept time event action [21]. These actions are shown in Fig. 2.7.

Accept event actions handle processing of events during the execution of a behavior. Such includes receiving signals from the environment and accept events when a timer expires. Send signal actions are used for sending signals to the environment.

Control Nodes Control nodes route both control and data/object flow.¹ There are seven kinds of control nodes, all listed and explained in table 2.1 [5].

Object Nodes There are four kinds of object nodes: *activity parameter nodes*, *pins*, *central buffer nodes*, and *data store nodes*. Data store and central buffer nodes are not in use in Arctis and will not be discussed further.

¹UML does not differ between object and data. They are used interchangeably under the notion of classifier [4].

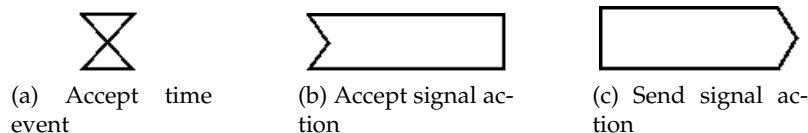


Figure 2.7: Special action notations



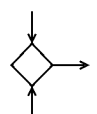
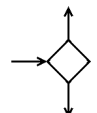



Control nodes	
	<i>Fork Node.</i> The incoming flow is split into several flows.
	<i>Join Node.</i> The outgoing flow starts after all incoming flows have arrived
	<i>Merge Node.</i> A merge node brings multiple flows together. The outgoing flow starts when one incoming flow arrives.
	<i>Decision Node.</i> A decision node chooses between outgoing flows.
	<i>Initial Node.</i> When an activity is invoked a flow starts in the initial node.
	<i>Activity Final Node.</i> An activity final node stops all flows in an activity.
	<i>Flow Final Node.</i> A flow final node stops a flow in an activity.

Table 2.1: Control nodes

Activity parameter nodes are object nodes at the beginning and end of flows that provide a means to accept input to an activity and provide outputs from the activity, through the activity parameters. An activity parameter node may have either all incoming edges or all outgoing edges, but it must not have both incoming and outgoing edges. Fig. 2.8 shows an activity with input and output activity parameter nodes. The pins with black background and white arrow are streaming nodes, which are parameter nodes through which tokens may pass while the activity is ongoing. The input parameter node with white background is a starting event, while the output parameter node with white background is a terminating event. The parameter nodes are used to couple the activity draw card with other collaborations.

A pin is an object node for inputs and outputs to actions. Pin rectangles

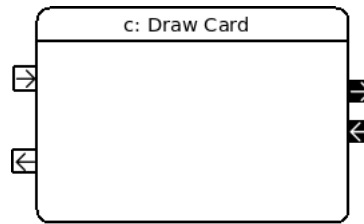


Figure 2.8: Simplified collaboration draw card with activity parameter nodes

are notated as small rectangles attached to action rectangles. Fig. 2.9 shows an example of the action *draw random card* which has an output pin that can hold a card.

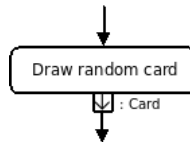


Figure 2.9: Example of action with an output pin

All object nodes specify the type of value they can hold. If no type is specified, they can hold values of any type [6].

Activity Edges

The activity nodes are connected by one of two kinds of activity edges; control flow and object flow. Control flow can only carry control tokens, while object flow can carry object and data tokens.

Control Flow A control flow is an activity edge that starts an activity node after the previous one is finished. Objects and data cannot pass along a control flow edge, it can only pass control tokens. Control flows may not have object nodes at either end, except for object nodes with control type. Fig. 2.10 shows an example of a control flow.

Object Flow Object flow is an activity edge that can have objects or data passing along it. Object flows may not have actions at either end. Object nodes connected by an object flow must have compatible types. Fig. 2.11 shows an example of an object flow. Note that the object flow has a object node at either end, in this example pins. To distinguish them from the

control flows denoted by black lines, object flows are denoted by blue lines in Arctis.

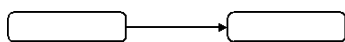


Figure 2.10: Control flow

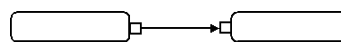


Figure 2.11: Object flow

Call Behavior Actions and Call Operation Actions

Call behavior actions and call operation actions are two central terms when talking about activities. A single behavior, like draw card, may be invoked in many activity diagrams or several times in the same activity diagram, but each invocation are represented by a separate instance of a call behavior action, all referring to the same behavior. This facilitates for reuse of collaborations.

The same applies to operations. A single operation behavior may be invoked several times in an activity diagram. Each invocation are represented by an instance of a call operation action. All call operation actions refers to the same operation and parameters [3].

Partitions

The activity *Draw Card*, depicted in Fig. 2.5, describes the behavior of the corresponding collaboration. It has one partition for each collaboration role; player and pile. As shown in Fig. 2.3 these roles are bound to player and draw pile. Partitions in SPACE are used to indicate what or who is responsible for the actions grouped by the partition. This means that a partition conforms to a UML class [7]. For call operation actions, this means that the class defines the invoked operation. For call behavior actions, this means that the class owns the behavior. In the example in Fig. 2.5, the class pile owns the call operation action *getStatus*, while the partition player has no call operation actions, but the class owns the behavior.

Variables

In addition to sending data in control flows, we also need to store values and retrieve them when needed. For example, players need to hold information of other participating players, how many cards they have, and their own cards. For this purpose, UML provides variables. [14] shows an example of how to use variables in SPACE.

A variable has a name and a type. The values contained in a variable must conform to the type of the variable. The UML standard does not provide a specific notation for the variable, but in SPACE we use the following

notation: *type : name*. For example, the name of the player is stored as a String: *String : playerName*.

Actions may access variables and perform operations on them, which means variables are a way of passing data between actions without using a data flow path. As we want to transform the activities into executable state machines for the implementation as described in [18], actions need to be localized. This means an action may only access variables owned by the partition that owns the action, as described in [14].

2.3 Extensions of UML by SPACE

In addition to the UML described above, SPACE offers some extensions to UML. This includes support for multi-sessions, and waiting decision nodes, which are described next.

2.3.1 Multi-Session Collaborations

Multi-session collaborations are described in [14], and are introduced to enable coordination of collaborations that are executed with several simultaneous sessions. Fig. 2.12 shows an example of a case where several collaboration sessions are needed. The example is a simplification of the *Playing* collaboration that will be described later in this work. The playing collaboration has two roles: player and discard pile. A player plays a card to the card pile in the sub-collaboration *PlayCard*. From the viewpoint of one player, there is one *PlayCard* collaboration session towards the discard pile. However, the discard pile has to maintain the sessions with each of the players, as all instances of the collaboration is executed at the same time. To express this, a stereotype multi-session is applied to the call behavior action, and a border is placed in the discard pile partition which have multiple sessions.

When a token enters a multi-session collaboration via a pin, a selection of sessions must be done. When a token enters the *CardUpdate* pin, it should go to all of the sessions, while the *Valid Card* should go to only the session where the card was played. Two operators were added to support this; select and exist. Fig. 2.13 shows the EBNF² definition for session selection and existence, which also allows for custom filters.

In the example in Fig.2.12, a player plays a card, a token is entered in the *Play Card* collaboration via the *Play Card* pin. The discard pile receives the token via the *New card* output pin. The card is treated in some way not shown in this figure, and if it is a valid card, the player must notify all players that a card has been played via the *Card Update* pin, and the player must receive a confirmation that the card was accepted via the *Valid Move*

²Extended Backus-Naur Form

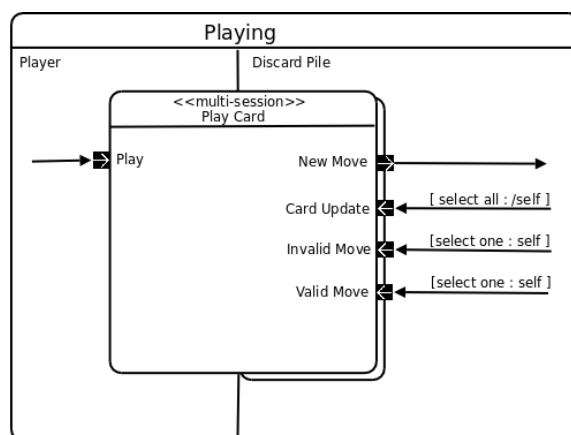


Figure 2.12: Example use of select

pin. The card update should be sent to all players, except the player playing the card. This can be expressed by **select all: /self**. The slash denotes exclusion. The card confirmation should only be sent to the player playing the card, which is expressed by **select one: self**.

```

select := 'select' mod ':' [{filter}] [ '/' {filter} ].
exists := 'exists' name ':' filter [ '/' {filter} ].
mod := 'one' | 'all'.
filter := name | 'self' | 'active'
         | 'id=' variable.
  
```

Figure 2.13: EBNF for select and exists (from [14])

2.3.2 Waiting Decision Nodes

A waiting decision node is an extension of a decision node, denoted with a filled diamond [14]. Waiting decisions are used in combination with join nodes to model the race between two or more flows. Fig.2.14 shows an example use of a waiting decision node. First, an incoming flow is split in three in a fork node. One flow starts a timer, another enters the waiting decision node, and the last continues to join node *j1*. If the timer expires, the token in the waiting decision node is pulled out of the waiting decision node, and the join node *j2* fires. If the flow *e1* arrives at join node *j1* before the timer expires, join node *j1* will fire.

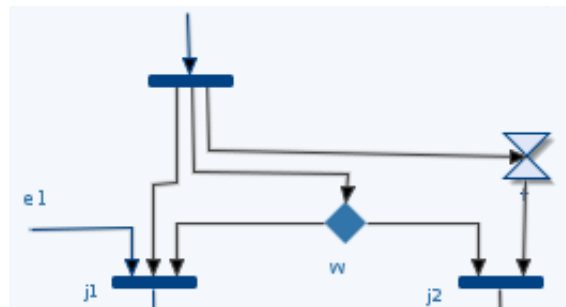


Figure 2.14: Example use of a waiting decision node

2.4 Current integration of Data in SPACE and Arctis

As mentioned in Sect. 2.2.2, UML provides some elements to handle data flow in activity diagrams: object nodes and object flows. Object nodes are needed to represent data as it flows in and out of invoked behaviors, or to represent collections of tokens waiting to move downstream [12]. Activity parameter nodes and pins are already supported in SPACE, as introduced in Chapt. 2. Object flows are needed for sequencing data produced by one node that is used by other nodes, as object and data cannot pass along a control flow edge.

2.5 UNO

UNO is a card game for several players. The game is normally played with physical playing cards and players located in the same room, sitting around a table. In this project we are going to design an electronic version of UNO, where players can play against each other even if they are physically separated. We assume that each player has a terminal, e.g. a PC or an Ipad touch, which all are connected to a game server. Fig. 2.15 shows how a UNO user interface may look like.

Because of the physical separation of the players, and the nature of an electronic game, we have to make adjustments to the game rules. The following section describes the official game instructions from Mattel [20], and discuss which adjustments we need to make.

2.5.1 Object of the game

“The object of the game is to be the first player to score 500 points. Points are scored by getting rid of all the cards in your hand before your opponents. Points are calculated from the cards remained in your opponents hands as follows:

- *All number cards: Face value*
- *Draw Two, Reverse, and Skip cards: 20 points*
- *Wild and Wild Draw Four cards: 50 points”*

[20]

Reaching 500 points may take a considerable amount of time. The players should be able to choose if they want to play a single game only, where the first player to get rid of all his cards wins.

2.5.2 Starting the game

“To decide who to deal every player picks a card. The person who picks the highest number deals. In this part of the game Action cards count as zero. The dealer shuffles the cards and deals each player seven cards. The remainder of the deck is placed face down to form a draw pile. The top card is turned over to begin a discard pile. The person left of the dealer starts the play.”[20]

In the electronic version the system will deal each player seven cards at startup of the game. A player is randomly chosen to start the game, and the system place a card on the discard pile.

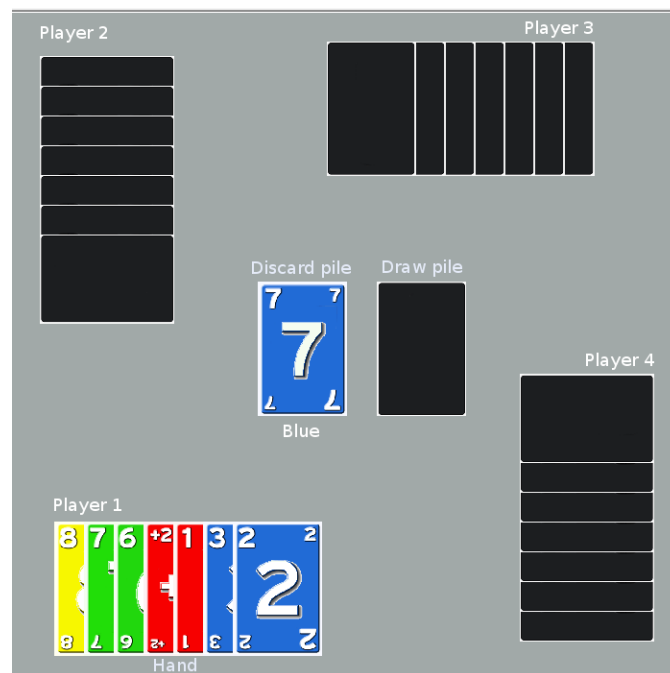


Figure 2.15: UNO game

2.5.3 Playing the game

"The game is played using 108 cards as follows:"

- 19 Blue cards** - 0 to 9
- 19 Red cards** - 0 to 9
- 19 Green cards** - 0 to 9
- 19 Yellow cards** - 0 to 9
- 8 Draw Two cards** - 2 in each color
- 8 Skip cards** - 2 in each color
- 8 Reverse cards** - 2 in each color
- 4 Wild cards** -
- 4 Wild Draw Two cards**

"For each turn, the player must play a card matching the card on the discard pile, either by number, color or symbol. Alternatively, the player can put down a wild card."[20]

In the normal version of the game, it is up to the other players to check if the played card is valid. In our electronic version, validation should not be

up to the other players. Thus, the system has to ensure that a player has played a valid card. An invalid card will be rejected.

“If the player doesn’t have a card to match the one on the discard pile, he must take a card from the draw pile. If this card can be played, the player may put it down in the same turn. Otherwise, the player have to say “pass”, and play moves to the next person in turn.”[20]

A card is taken from the discard pile by pushing a draw button. The player says “pass” by pushing the pass button. The system has to ensure that it is not possible to say pass without having drawn a card from the discard pile first. The system also have to ensure that a player draws at most one card for each turn.

“When a player has only one card left, he/she must yell “UNO”. Failure to do this results in having to pick up 2 cards from the draw pile.”[20]

Because the players are physically separated, a player must yell UNO by pushing a “UNO” button. The other players will then be notified that a player has yelled UNO. If a player forgets to push the UNO-button when he has only one card left, this is detected by the system, and the player has to pick up 2 cards.

“Players who make card-play suggestions to the other players must draw two cards from the draw pile.”[20]

In this game we assume that the players is unable to make such suggestions to other players.

“If the stock is emptied, the discard pile is shuffled and turned over to replenish the stock.”[20]

When the stock is empty, the discard pile is automatically shuffled by the system. It is important to keep the discard pile and the draw pile separated in our electronic game, as it should not be possible to draw a card from the draw pile that recently has been added to the discard pile. This way, we keep differences of the electronic and non-electronic version of the game to a minimum.

“If a player cheats, or accidentally plays a wrong card, and it is noticed by the other players, he must take the card back and take 2 extra cards from the draw pile.”[20]

In our electronic game, the system will automatically detect if a player plays a wrong card, and the player has to draw two extra cards. There are a number of reasons not to allow cheating. First, we think the game will be rather chaotic if we allow cheating, leaving all card validation to the players. Second, it is difficult to determine who actually cheated, since a players can not necessarily see who has played the card, and the previous card on the draw pile.

2.5.4 Action cards

There are a number of action cards that affects the course of the game in different ways. The action cards, their function, and their rules are given below. Fig. 2.16 show the action cards used in our electronic game.

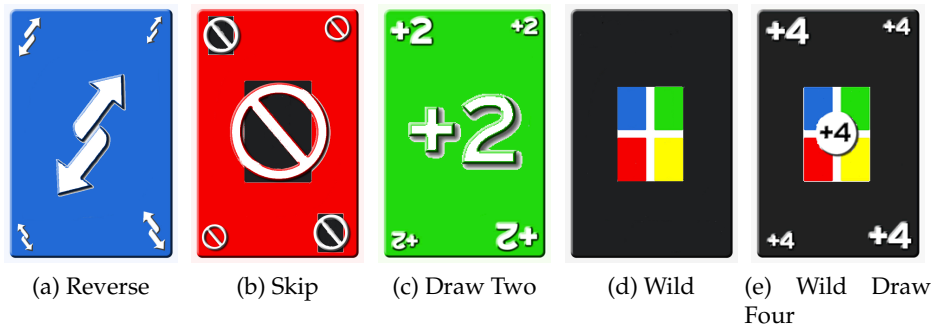


Figure 2.16: Action cards

Draw two: If a *draw two* card is played, the next person to play must draw 2 cards and miss his turn. This card can only be played on matching colors and other *Draw Two* cards. If turned up at the beginning of play, the same rule applies.

Reverse: If a *reverse* card is played, direction of play is reversed. The card may only be played on a matching color or on another *Reverse* card.

Skip: If a *skip* card is played, the next player after this card has been laid loses his turn and is "skipped". The card may only be played on a matching color or on another *skip* card. If this card is turned up at beginning of play, the player left of the dealer is skipped, and the player left of that player starts to play.

Wild: A *wild* card can be played at any time, even if the player has another playable card in the hand. The person playing this card calls for any color to continue the play, including

the one currently being played. If this card is turned up at beginning of play, the person to the left of the dealer determines the color, which continues play.

Wild draw four: If a *wild draw four* card is played, the person who plays it calls the color that continues play. Also, the next player has to pick up 4 cards from the *draw* pile and miss his turn. The card can only be played when the player do not have a card in his hand to match the color on the *discard* pile. If this card is turned up at beginning of play, it is returned to the deck and another card is picked. A player holding a Wild Draw Four card may choose to bluff and play the card illegally. When a *wild draw four* card is played, the player required to pick up the four cards can challenge the player playing the *wild draw four* card. When a challenge is issued, the hand of the player must be shown to the player who made the challenge. If the *wild draw four* card has been played illegally, the offending player must draw 4 cards. If the card has been correctly played, the challenger must draw 2 cards in addition to the 4.

It is possible to do a simplification in the electronic version, and let a player play a wild draw four card even if he has another card in his hand to match the color on the discard pile. Eventually, we can implement the challenge rule, and give the player having to draw four cards possibility to challenge the player playing the wild draw four card by pushing a challenge button. However, instead of the challenged player to show his cards, the system may determine if the wild draw four card has been played illegally or not. As we do not consider the challenge rule very central in this game, we will not include this rule in a first version.

2.5.5 Jump-in Rule

In addition to the official rules given above, there are a number of “house rules” that can be used when playing. A detailed description will not be given in this report. However, in our version of UNO we will use the “jump-in” rule as it provides some challenges in designing and implementing UNO. If a player holds a card that matches (identical color and number) the card on the *Discard* pile, the player can play a card, even if it is not his turn. This is called a “jump-in”. The game continues with the player next to the player doing the “jump-in”.

2.5.6 Special properties of the mobile version

As the players are not sitting face to face, they have no possibility to see how many cards the other players have. This information must be provided by the user interface. Neither can players hear the color chosen after a player has put down a wild card. Therefore the user interface has to provide information on which color is expected next, in addition to the top card in the discard pile.

During play, a player often has to draw a number of cards, for example when a wild draw four card or a draw two card is played. Instead of relying on the players to draw the correct number of cards, the system should push the correct number of cards on each player.

Chapter 3

Specification of UNO — Part 1: Object and Collaboration Structure

Before we introduce data handling in the next chapters, we study the requirements of UNO in Sect. 3.1. Section 3.2 discusses which objects will be needed in this game, while Sect.3.3 discusses attributes each of the objects need during the game. Lastly, the collaborations between the objects are described and discussed.

3.1 Requirements Capture

Requirements based on the rules and the discussion in chapter 2. The requirements are parted into 3, according to the three phases of the game, which corresponds to the collaborations *Setup*, *Playing* and *End*.

3.1.1 Starting the Game

1. A player has to register with the game controller to participate in the game.
2. At startup each player is dealt 7 cards each.
3. A player is randomly chosen by the system to start the game.
4. The system choose a card to begin the draw pile. If a wild draw four card is chosen, it is returned to the deck and another card is picked.

3.1.2 Playing the Game

5. A player need at all times to have information about:
 - How many cards each player has.
 - Whose turn it is.
 - Top card on the discard pile.
 - Which color is chosen when a wild card has been played.
 - Direction of play.

This information must be visible in the user interface.

6. For each turn, the player must play a card matching the card on the *discard* pile, either by number, color or symbol. Alternatively, the player can put down a wild card.
7. If the player does not have a card to match the one on the discard pile, the player must draw a card from the draw pile. If this card can be played, the player may put it down in the same turn. Otherwise, play moves to the next person in turn. It is not possible to draw more than one card for each turn.
8. If a player plays a wrong card, this will be noticed by the system, and the player must take the card back and take two extra cards from the draw pile.
9. When a player has only one card left, he must yell “UNO” by pushing a “UNO” button. Failure to do this results in having to draw two cards from the draw pile. The player must yell “UNO” within five seconds after playing his second to last card.
10. If a *Draw Two* card is played, the next person to play must draw two cards and miss his turn.
11. If a *Reverse* card is played, direction of play is reversed.
12. If a *Skip* card is played, the next player after this card has been laid loses his turn and is “skipped”.
13. A *Wild* card can be played at any time, even if the player has another playable card in the hand. The person playing this card calls for any color to continue the play, including the one currently being played. If this card is turned up at beginning of play, the person to the left of the dealer determines the color, which continues play.

14. If a *Wild Draw Four* card is played, the person who plays it calls the color that continues play. Also, the next player has to pick up 4 cards from the draw pile and miss his turn. The card can only be played when the player do not have a card in his hand to match the color on the discard pile.
15. If the stock is emptied, the discard pile is shuffled and turned over to replenish the stock.

3.1.3 Winning the Game

16. Once a player has no cards left, the hand is over. The player receives points for cards left in opponents' hands as follows:
 - All number cards: Face value
 - Draw Two: 20 points
 - Reverse: 20 points
 - Skip: 20 points
 - Wild: 50 points
 - Wild Draw Four: 50 points
17. When a hand is over, the other players have to be notified that the hand is over and the name of the winner.
18. The winner is the first player to reach 500 points.

3.2 Object-Oriented Analysis

The purpose of an object oriented analysis is to get an understanding of the problem at hand by discovering objects that form the vocabulary of the problem domain [8]. This section discuss which objects we need in UNO.

The objects we first think of is players. Each human player must be represented by a player object. A player has behavior, and is as such an active object. The player object communicates with a player via a graphical user interface. Another central entity in a card game is of course the card object. Each player has a number of cards at hand, and the discard pile and the draw pile have a number of cards. In total there are 108 different instances of the card class.

What other objects we need in the system are not that easily recognized. Normally, the game consists of a number of players sitting around a table holding a deck of cards divided into a discard pile and a draw pile. Thus, UNO is by default centralized, as illustrated in Fig. 3.1. A natural idea is to let the system roles consist of players, draw pile and discard pile. In

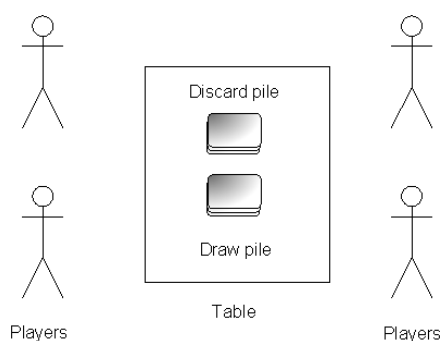


Figure 3.1: System view

general, players add cards to the discard pile, and draw cards from the draw pile. It should be possible to perform these tasks simultaneously, by different players. But only one at a time can add a card to the discard pile, and one at a time can draw cards from the draw pile. These two objects need to make sure only one player can access them at the time. Both the draw pile and the draw card are active objects.

However, this functionality is not sufficient to implement the desired behavior. In the real world the players perform tasks like deciding if the card played is valid, keeping control of whose turn it is, direction of play and so on. In case of disagreement, the players has to reach consensus before play continues. These decisions are made based on the rules the players keep in their mind. To mirror the real world, our electronic version should have a central controller unit mirroring the abstract functions the players perform together. Letting the players perform these tasks is not feasible in our electronic version of the game, as we assume physical separation of the players. The controller unit keeps track of participating players, how many cards they have, top card on the discard pile, the discard pile, who's turn it is, and current direction of play. In addition, the controller need to keep all players updated on what happens in the game. For example, when a card has been played, all players need to be informed. The controller is also an active object as it has behavior. Fig. 3.2 shows an illustration of the system with the controller unit. The players interacts with the discard pile by adding cards, and with the draw pile by drawing cards. The discard pile interacts with the controller for e.g. card validation, and the controller interacts with the players for game updates.

Having the controller and discard pile as different object is not a requisite for a good specification. In our specification we have chosen to model the controller behavior and the discard pile behavior as one entity, as shown in Fig. 3.3. This entity is named a discard pile, to mirror the physical version of the game. This follows the naming suggestions stated in [8], saying that things should be named in a way that their function is easily recognized by

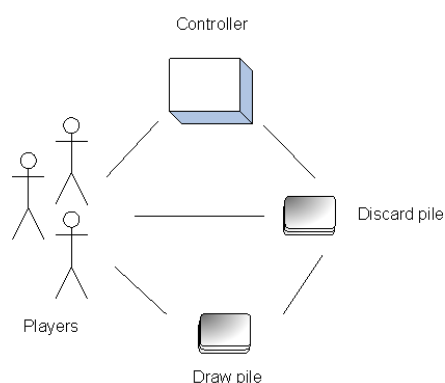


Figure 3.2: System view with controller unit

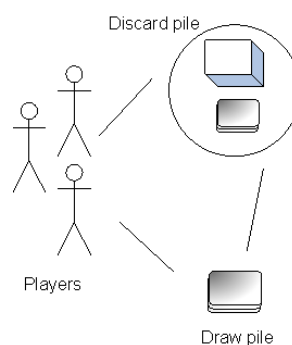


Figure 3.3: System view with the controller included in the discard pile

domain experts¹. Tbl. 3.1 summarizes the object oriented analysis.

Object name	Purpose
Player	Mirrors the real world player.
Draw pile	Mirrors the real world draw pile. Has to make sure that only one card at a time can draw cards.
Discard pile	Mirrors the physical discard pile. Makes sure that only one player at a time can add a card. Also includes the controller functions that players perform, like validating the cards
Card	Mirrors the playing cards.

Table 3.1: Objects in UNO

3.3 Object Attributes

Attributes describes an object's properties. Table 3.2 lists all objects and the belonging attributes.

A card has a color, symbol and value. The symbol is what is shown on the face of the card, like wild card, draw two, or the number 6. The value attribute is used when calculating score after a hand is over. All these attributes are of integer type.

Players are identified by a name and an ID, which is defined by the *me* attribute. The attribute has type *PlayerData*, which is a composite object for representing players. The reason for this is that players are active objects,

¹A domain expert is a person speaking the vocabulary of the problem domain, often just a user. In this case, it is a person playing the game.

which can not be passed around, as will be discussed later in this work. A player has a *hand*, which is the list of cards the players have available. The attributes *otherPlayers* and *otherHands* holds respectively a list of references to other players, and a list of how many cards each of the other players have. Players also hold the game status data; *topCard*, *topColor*, *turn* and *direction*. Top card has type *Card*, turn is of type *PlayerData* referring to a player, while the other attributes are of type *int*.

The draw pile has a *drawpile* attribute, which is a list of cards in the draw pile. When the draw pile is empty, the discard pile is emptied and put in the draw pile.

A discard pile has a *discardpile*, holding a list of cards that has been played. In addition the discard pile need all game data; *turn*, *direction*, *topCard*, and *topColor*, which is similar to the game data attributes of the player. This discard pile use this data to decide if the cards played are valid. The discard pile also has control over the participants of the game, and thus have a *players* attribute. The attribute *score* holds the score for each player.

We have chosen to let the players and the discard pile both hold the game data (*turn*, *topColor*, *topCard*, and *direction* and let both the players and the discard pile calculate these values when a move has been made. Then, the discard pile do not need to send these values to the players.

Card	
<i>color: int</i>	The color of the card
<i>symbol: int</i>	Card symbol, e.g. wild card, skip or 6.
<i>value: int</i>	Card value for calculating points when the hand is over.
Player	
<i>me: PlayerData</i>	The players reference to their own name and ID.
<i>hand: ArrayList<Card></i>	List of the player's cards.
<i>otherPlayers: ArrayList<PlayerData></i>	Reference to the other players.
<i>otherHands: int[]</i>	List of how many cards each of the other players has at hand.
<i>topCard: Card</i>	The card currently at top of the discard pile.
<i>topColor: int</i>	The current color. It is needed when a wild card is played, so the other players know which color is chosen.
<i>turn: PlayerData</i>	Who's turn it is.
<i>direction: int</i>	Direction of play, either clockwise or counterclockwise.
Draw pile	
<i>drawpile: ArrayList<Card></i>	List of the cards contained in the draw pile.
Discard pile	
<i>discardPile: ArrayList<Card></i>	List of all the cards that has been played.
<i>players: PlayerData[]</i>	List with all players participating in the game.
<i>turn: PlayerData</i>	Who's turn it is.
<i>direction: int</i>	Direction of play, either clockwise or counterclockwise.
<i>topCard: Card</i>	The card currently on top of the discard pile.
<i>topColor: int</i>	The color of the current topCard. In case of a wild card, it holds the color that has been chosen.
<i>score: int[]</i>	List of the score of each of the other players

Table 3.2: Objects attributes

3.4 Collaboration analysis

As found in the previous section, participants in the UNO service are players, draw pile, and discard pile. As discussed, the players should be able to place cards at the discard pile, and draw cards from the draw pile. The discard pile validates cards, and keeps all players up to date on what is happening in the game.

Before the system can start, players have to know who participates in the game, who should start the game, and which card begins the draw pile. The players, the draw pile, and the discard pile performs these tasks in a *Setup* collaboration.

For each card that is played, the collaboration have to check if the player had right to play the card and if the card played is valid. The collaboration need to inform other players that a card has been played. If players have no card to play which match the one on the discard pile, they must draw a card. All this is done in a single collaboration *Playing*.

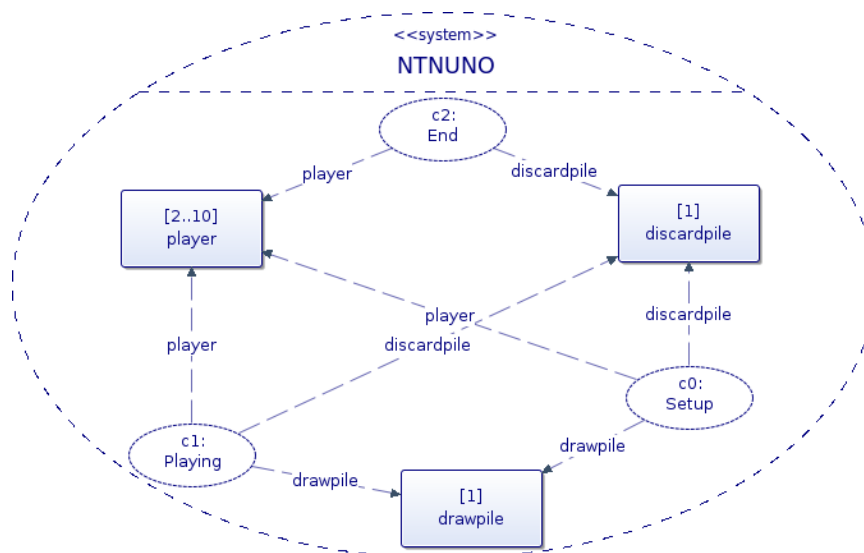


Figure 3.4: System collaboration

When a player has played all his cards, score has to be calculated and updated, the winner has to be announced, and if there is no winner, all cards have to be collected before a new game can start. This is done in the *End* collaborations. Fig. 3.4 shows that player, discard pile, and draw pile are interacting with collaborations *Setup* and *Playing*. Player and discard pile also interacts with the *End* collaboration. We have named the system `NTNUNO`, and with the stereotype `<<system>>` we express that Fig. 3.4 documents the highest system level. Discard pile, and draw pile has a default multiplicity of one, while there can be from two to ten players in a

game, according to the game rules.

The main idea with collaborations is to give a description of how the system works, while leaving out unimportant details. The collaborations shown in Fig. 3.4 show the three stages of the game, but it is not sufficient to describe how the system works. Thus, further level of detail is necessary.

The *Setup* collaboration is shown in Fig. 3.5. In this, the initialization of the game is done. This involves choosing a player to start the game, choose a top card to begin the discard pile, and deal each player seven cards. In addition, all players need to be informed of game participants, whose turn it is and which card begins the discard pile. These tasks are done in the respective collaborations *Select Turn*, *Select Top Card*, *Deal*, and *Distribute Players*.

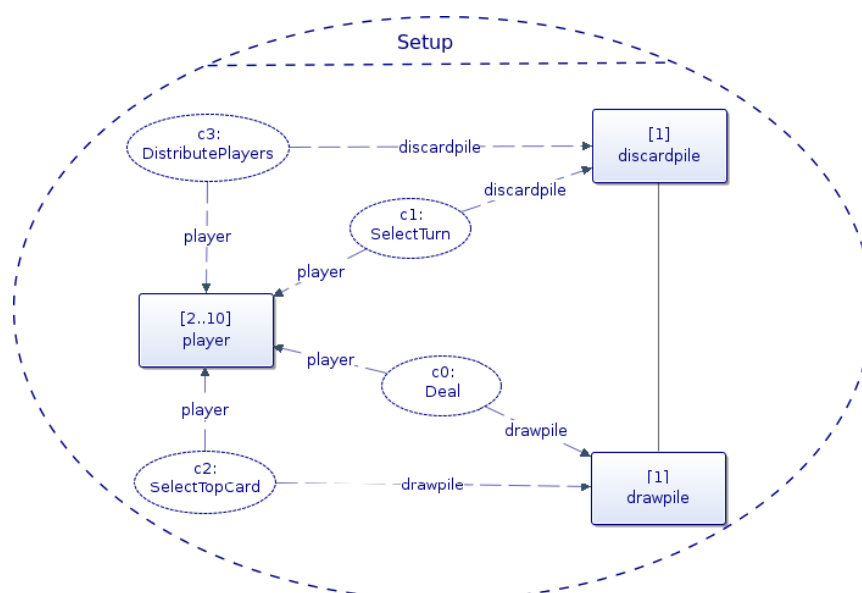


Figure 3.5: Setup collaboration

The *Playing* collaborations has several sub-collaborations, as shown in Fig. 3.6. This includes *Draw Card*, *Turn Pile*, *Make Move*, *Color Dialog*, and *Game Updates*. The *Draw Card* collaboration makes sure only one person at a time can draw a card, by using the well known binary semaphore pattern for mutual exclusion (for example described in [10]). To draw a card, a person must first ask for permission. The draw pile grants permission by giving a token. The token is returned when the player has drawn a card. This mirrors the real world, where people have to wait in turn to draw cards from the same pile.

If the draw pile is empty, the cards from the discard pile will have to be shuffled and put back in the draw pile. This happens in the *Turn Pile* collaboration. The collaboration *Make Move* takes care of the problem of

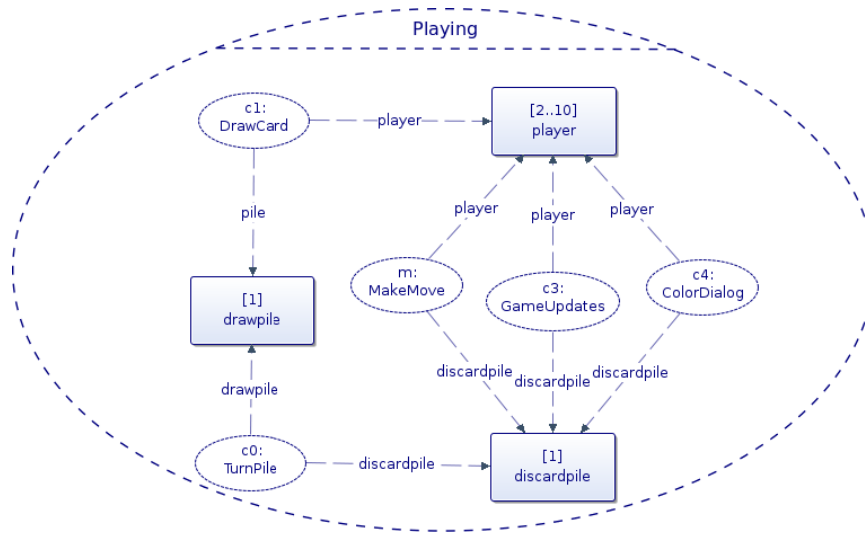


Figure 3.6: The playing collaboration

mixed initiatives. We can have mixed initiatives because a player can make a move, e.g. play a card, while another person can do a jump-in at the exact same time. As transitions takes time, a player can do a jump-in after another player has played a card, but before the card has reached the controller. This means that the discard pile will change before the jump-in card reaches the discard pile. How the *Make Move* collaboration will address this issue will be further discussed later in this work.

During the game, the controller will have to exchange data with the player. The players and controller exchange data via the *Game Updates* collaborations. When players play a wild card, they have to choose a color to continue the game. This is done in the *Color Dialog* collaboration.

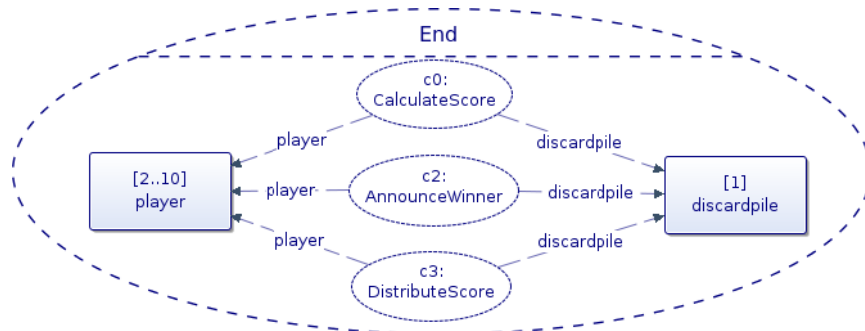


Figure 3.7: The *End* collaboration

The *End* collaboration has three collaboration uses, as shown in Fig. 3.7. The first is *Calculate Score* which calculates how many points the winner

scores based on the cards remaining in the other players hands. The player gives the remaining cards to the discard pile, which calculates all scores. If a player has reached 500 points, the game is over, and the discard pile announces the winner to all players. This is done in the collaboration *Announce Winner*. If no player has reached 500 points, the players are updated of the current score in the *Distribute Score* collaboration.

Introducing Data into SPACE

With the start of our work, SPACE focused on control flows and the Arctis tool did not support object flows or object nodes in activities. In the specification of UNO, however, object flows are needed as well, as it is necessary to distribute card data, player data, and other game data among the participants.

In this section, we discuss some patterns where object nodes and object flows are needed in the specification of UNO, and which elements UML offers to express the desired behavior. We also discuss how Arctis may be extended to support this behavior. Motivated by practical needs for implementing UNO, the following functionality has been added:

- Input and output parameter to actions.
- Actions for setting and reading variables.
- Sending several objects in same flow.
- Transforming types between object nodes.
- Output pins to accept signal actions.
- Input pins to send signal actions.
- Fork node with both object and control flow.

4.1 Input and Output Parameters to Actions

To examine and perform actions on data, we have need for actions to handle data input and output. Data input and output are denoted by pins, which are kind of object nodes [4]. Arctis generates Java code from the UML activities specification. Fig. 4.1a shows a sum action with two integers a and b as input parameters, and one integer as outgoing parameter, which is the sum of the input parameters. Fig. 4.1b shows the corresponding Java code. In Java, methods have at most one return value. To keep

the mapping between Java and UML operations simple, we permit only one outgoing pin from an action. For each input pin parameter type and parameter name has to be specified. Each call operation action must refer to a Java method, which must have the same number of input parameters as there are input pins to the call behavior action. The name and type of the input pins must match the name and type of the parameters in the Java method.

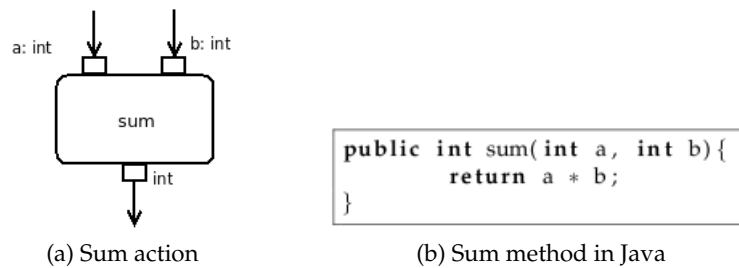


Figure 4.1: Sum action and corresponding Java code

An action starts when all its data inputs are available. In the example in Fig. 4.1a, if parameter b arrives before a , the action will store parameter b and wait for parameter a before it starts [4]. From a control-flow point of view, this implies a similar synchronization behavior as a join node.

4.2 Setting and reading variables

UML provides actions for saving and retrieving variables; *add variable value action* and *read variable action*.

- Add variable value action is a write variable action for adding values to a variable [12]. Add variable value action has one input pin and no output pin, as shown in Fig. 4.2.
- Read variable action is a variable action that retrieves the values of a variable. Read variable action has no input pin and one output pin, as shown in Fig. 4.3. The type of the output pin is the same as the specified variable.

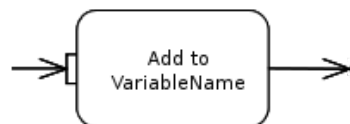


Figure 4.2: AddVariableValueAction notation

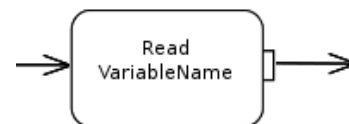


Figure 4.3: ReadVariableAction notation

Figure 4.4 shows an example for the usage of read and write actions. *Counter* is a building block used to keep count of how many times a repetitive task is performed. In this case, the task is to draw a given number of cards. The action starts when the starting event *start* arrives, specifying how many cards to draw. The variable *count* is set to number of cards to draw. Each time a *event* arrives, the *count* variable is decreased by one in the *decrease count* action. The *decrease count* action have access to the *count* variable. After the counter is decreased the *Card* object is sent on. Instead of passing the *Card* token through the *decrease count* action, the *Card* is saved to a variable, and retrieved after the *decrease count* action is finished.

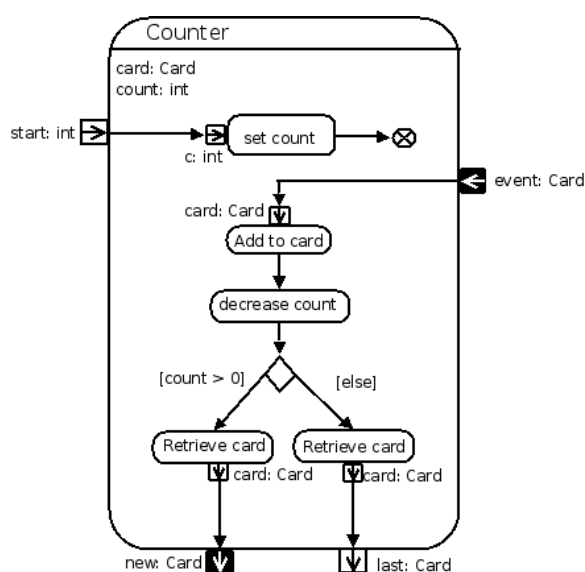


Figure 4.4: Counter

4.3 Several Objects in Same Flow

In UNO, many objects have to be exchanged between the participants, and often several objects which belong together need to be passed between participants. For example, players have to inform other players which card they have played. The other players need information about both the card and who has played the card. An object flow, however, can only hold a single object token. This means that card and player can not simply be sent in the same flow. A solution could be to send the objects in two different flows, and synchronize the flow at the receiving side by a join node. A join node offers token to the outgoing edge in the same order they were offered to the join [12]. This means that it is not possible to know in which

order the tokens will arrive. This again means that the card and player tokens have to be saved to variables before arriving at the join, to make sure the card value is saved to the card variable, and the player value is saved to the player variable. Sending several objects and synchronizing them at the sender side is untidy, and involves several operations that may be avoided.

A better solution is to add objects to a single object containing references to two objects. For example, the objects Card and Player may be merged to a Move object, as shown in Fig. 4.5. Fig. 4.6 illustrates the Move class.

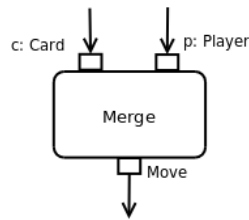


Figure 4.5: Merge action

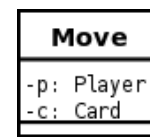


Figure 4.6: Move class

With this solution synchronizing the behavior is unnecessary, and there is no need to handle tokens arriving in different orders. We find this solution more clear, as there is only one flow exchanged between partitions instead of two. This is especially the case when we have more than two objects that need to be synchronized at the receiving side. For this reason, in addition to the fact that the solution is in accordance to the UML standard, the solution shown in Fig. 4.5 is adopted in the UNO specification. Another point of using this solution is that it could be automated by a specialized action in Arctis.

4.4 Transforming Types between Object Nodes

If an action requires a Card object, but is provided with a Move object, the downstream action has received an incompatible type.

UML 2.0 provides a transformation behavior to solve this problem. Fig. 4.7 shows the UML transformation notation. The action *Create Move* provides a Move object as output, while the action *Update Top Card* requires a Card object as input. The transformation behavior transforms the Move object to a Card object, and thus ensures that the downstream action receives a valid type.

The transformation behavior takes one input parameter and provides one output parameter. The input parameter must be of the same type as the object token provided by the source object node, and the output parameter must be of the same type as the object token expected by the end object node. This way, the UML specification constraint that object nodes connected by object flows must be of compatible types is not broken.

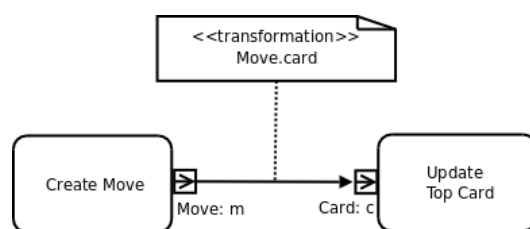


Figure 4.7: Transformation notation in UML

It is also possible to model the transformation behavior as an action with one input parameter and one output parameter, as shown in Fig. 4.8. This alternative is easier to support by Arctis, as it only uses the elements from call operation actions as introduced above. For this reason, we propose to use this form.



Figure 4.8: Transformation notation in Arctis

4.5 Output Pins for Accept Signal Action

Accept event actions for signals, also called accept signal actions, need support for data flows as well. In the current approach, these actions are used to receive signals from the environment, to which for example the user interface belongs. For example, when a player plays a card, an accept event action receives a signal. To validate the card, and inform other players of which card has been played, we need to know which card has been played. This information has to be extracted from the parameters contained in the received signal.

According to the UML standard [12], an accept event action may have zero or a number of output pins. The output pins of an accept event action may hold the received signal, or the attributes contained in the received signal. These two possibilities are illustrated in Fig. 4.9. Which of these two is chosen depends on the value of the attribute *isUnmarshall*. If *isUnmarshall* is set to true, the result output pins contains the values of the signal attributes, as shown in Fig. 4.9a. If *isUnmarshall* is false, the received signal is placed on the output pin of the action, as shown in Fig. 4.9b.

In UNO, we are not interested in the signal itself, but its parameters.

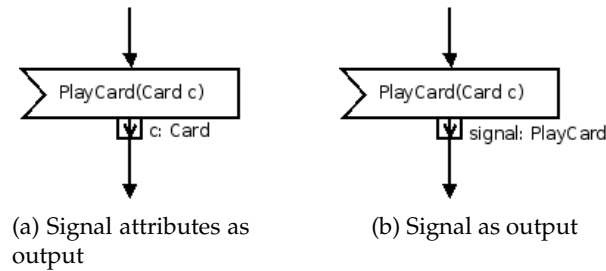


Figure 4.9: Alternative methods for extracting info from received signal

Therefore, we assume all signals to have `isUnmarshall=true`, and add one output pin for each parameter to the accept signal action.

4.6 Input Pins for Send Signal Actions

When updating the user interface with data, for example how many cards the opponent has, or the color chosen when a wild card is played, signals containing this data are sent to the user interface. To add data to signals, send signal actions are extended with input pins. UML states that the number and order of argument pins must be the same as the number and order of attributes in the signal. Fig. 4.10 shows an example of a send signal action where the signal to be sent is *ColorUpdate*, which has a parameter *color* of type *int*. The send signal action has one input pin which accepts a token matching the signal parameter. The signal is created and sent when all its input is available.

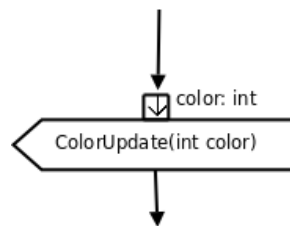


Figure 4.10: Send signal action with input pin

4.7 Fork Node with both Object and Control Flow

The UML standard states that the flows coming into and out of a fork node must be either all object flows or all control flows [12]. However, we found that this is not always suitable in UNO. Often one of the outgoing flows is

used for sending data, while another flow is used to perform an action where only a control flow is needed. Fig. 4.11 shows three fork nodes, $f1$, $f2$, and $f3$, where this is the case. Note that in Arctis, control flow is denoted with solid black lines, while object flow is denoted with dashed blue lines.

The example shows the activity *DrawCard*, which consists of two partitions, *player* and *pile*. *Player* contains a building block *Counter* which keeps count of how many cards that has been drawn. This building block has been described earlier in this chapter. The *DrawCard* activity starts at the arrival of the draw event. The object flow is forked at $f1$, and one flow starts the counter block, while the other flow is sent to the output pin *getAllowance*. This output pin does not require any data, and is therefore provided with a control flow. Fork node $f2$ shows a similar situation, where it receives a *Card* token, which is sent out via an outgoing flow, while the other outgoing flow are sent to *getAllowance*.

An alternative solution could have been to let both flows from the fork nodes be object flows. This would result in the parameter node *getAllowance* receiving an object token that it does not need. Another consequence is that the merge node in the *player* partition will receive different object tokens. This is not a problem here, as the object token has no influence on the behavior, but in general we do not want this to happen. Another reason for avoiding sending unnecessary information is that unnecessary information is a security problem in applications where security is important.

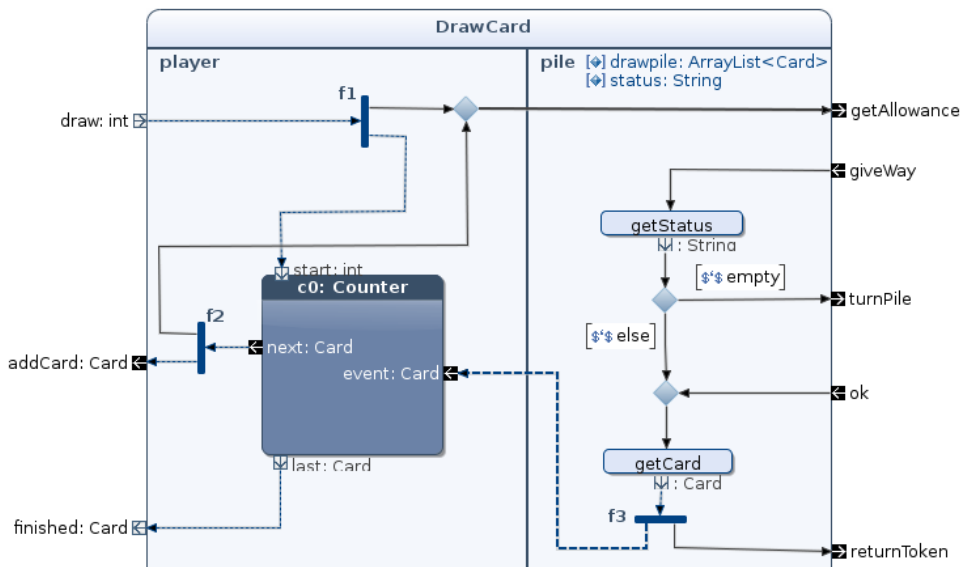


Figure 4.11: Screenshot of the *DrawCard* service designed in Arctis

To summarize, fork nodes with all control flows or all object flows are allowed, according to the UML standard. In addition, fork nodes with an incoming object flow may have control flows in addition to an control flow. These valid fork nodes are shown in Fig. 4.12b. Fork nodes with an incoming control flow can not have outgoing object flows, as the fork does not have any object tokens to pass on. Fork nodes with an incoming object flow can not have only outgoing control flows, as the object tokens are then unnecessary, and if it is not needed in the fork it should not be sent to the fork in the first place. The invalid for nodes is shown in Fig. 4.12a.

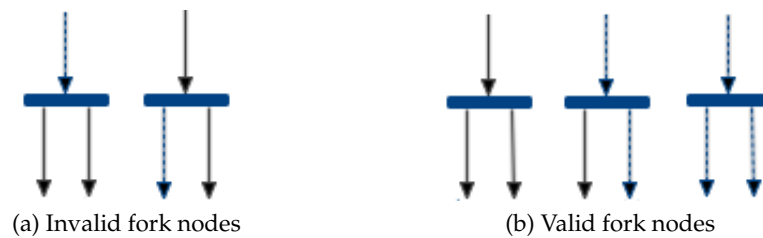


Figure 4.12: Examples of valid and invalid fork nodes

4.8 Decision Nodes

As stated in the UML standard [12], the input and output flows to a decision node must either be all control flows or all object flows. Guards of the outgoing edges, which are Boolean value specifications, are evaluated to determine which edge the tokens should traverse. Tokens are offered to all edges, but only one should pass, meaning that only one guard should evaluate to *true*.

When decision nodes receives a control flow, partition variables or methods are evaluated in the guard. An example is shown in Fig. 4.14. The building block *Simple Counter* has a decision node with two outgoing edges. The leftmost edge is traversed if the *count* variable of type integer is larger than 0. Else, the rightmost edge is traversed, which have an *else* guard.

In the example shown in Fig.4.13, the decision node receives an object token with a boolean value. The token is passed on to both outgoing flows. The two guards are false and true, respectively. If the token is false, the token traverses the leftmost node which has the *false* guard. If the boolean token evaluates to true, the rightmost edge is traversed.

In Arctis, each guard has a Java method, which may declare at most one parameter. In the case of control flows no parameters are needed, as the result of the guard does not depend on the token carried in the flow. With object flows, the guard method need the object flow token as parameter to determine if the token should be allowed to pass.

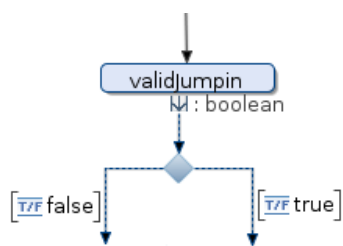


Figure 4.13: Example of decision node with object flow

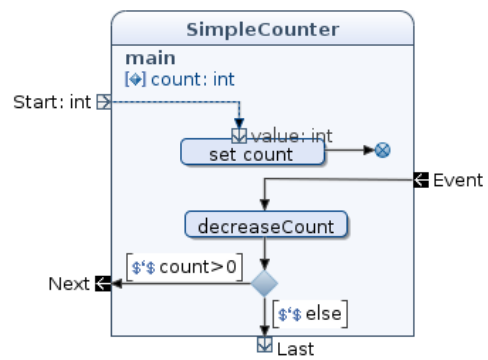


Figure 4.14: Simple counter

Chapter 5

Specification Guidelines

While specifying the UNO behavior, we soon realized there was an endless number of ways to specify a system. The systems can be correct with respect to the functional requirements implied by the game itself, but still not optimal with regard to reuse, clarity and intuitiveness. Several issues need to be kept in mind. Here we discuss some non-functional issues that are central for a good specification of a system. The first section addresses active vs passive objects. The next section discuss how to model large and complex functionality in a best possible way in regard to reuse and clearness.

5.1 Active and Passive Objects

In UNO, as in all other telecommunication systems, we have to distinguish between active and passive objects. Active objects have behavior, and are run within its own thread. An example of an active object in UNO is a player. A passive object is an object without a behavior. An example of such an object is a card.

A player object can not be sent between the players, in the same way a card may be sent around. But players need to have a reference to the other participating players. When a player plays a card, the other players are informed of which card has been played, and who played it. Since the player object can not be sent around, we have to make another object, e.g. player data, which is a reference to the player object. Thus, a player is identified by a *PlayerData* object. Each player has a *PlayerData* object for each of the other participating players. When a card has been played, each player receives an object containing the card that has been played, and a *PlayerData* object identifying the player of the card.

5.2 Modeling Issues in Regard to Variable Access and Clearness

As stated earlier in this chapter, it is necessary to keep data about the game in a number of variables. This includes direction of play, top card on the discard pile, and whose turn it is. Some of these variables will have to be changed every time a player plays a card, someone draws a card or someone passes. In UML, these variables are only accessible from the activity owning the variables. Consider the activity shown in Fig. 5.1. The playing activity consists of three partitions; player, discard pile and draw pile. Note that for simplicity, no behavior is included in this diagram. In the *Make Move* collaboration players either play cards to the discard pile, or passes. This collaboration decide who came first when two players play a card almost simultaneously. In the *Game Updates* collaboration, the players and the discard pile distribute game data, like number of cards at hand or the color chosen after a wild card has been played. In the *Draw Card* collaboration, the players draw card from the draw pile. In case of an empty draw pile, the discard pile must be turned. This happens in the *Turn Pile* collaboration. In this discussion we will focus on the discard pile partition in the top right of Fig. 5.1. Discard pile is the partition owning all game variables. In this example, the discard pile performs two tasks, but we will later see that it also has a number of other tasks. The two tasks are to validate the card and update all game data. When a card is played, the discard pile will check whether the played card is a legal one, which means having a correct color or symbol. If it is a legal card, all variables will be updated, as it now is a new card on the discard pile, turn is moved to the next player, and so on.

Problem context When specifying the validate card behavior we saw that it included many UML elements. As a consequence of this, we thought of having a building block for this function. Then we realized we would get problems with the variable access. However, not having a building block would result in a complex and large-scale specification. Thus, we had to find alternative ways to describe the wanted behavior.

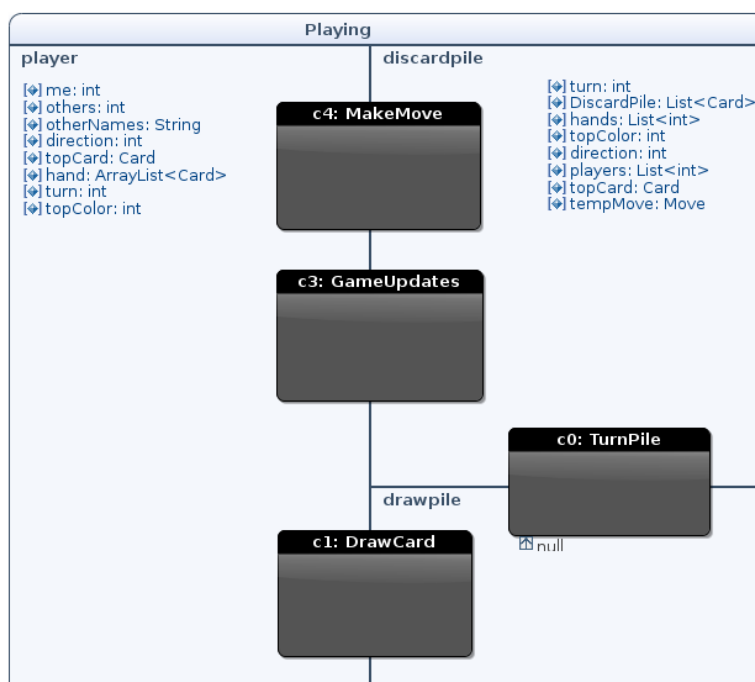


Figure 5.1: Simplified Playing activity in Arctis

Preview. The rest of this section presents and discuss the solutions we considered. This includes using building blocks with shared variables, a flat specification, more comprehensive building blocks, building blocks with variables provided as input parameters instead of sharing data, and finally typed collaboration roles. Table 5.1 gives a brief introduction of the solutions discussed in the rest of this chapter.

Preview	
<p>Alternative 1: Shared data between building blocks</p>	<p>The building blocks share data with the parent partition. Meaning that a building block may access the data from within the building block.</p>
<p>Alternative 2: A flat specification</p>	<p>Flat specification where no building blocks are used. This results in many UML elements on the same level.</p>
<p>Alternative 3: More comprehensive call operation actions</p>	<p>More of the behavior may be put in call operation actions, resulting in a less detailed activity diagram, thus less UML elements.</p>
<p>Alternative 4: Providing variables as input parameters to building blocks</p>	<p>All data that are needed in the building blocks are provided as input parameters to the block, meaning that no data sharing is used.</p>
<p>Alternative 5: Typed collaboration roles</p>	<p>The collaboration roles encapsulate data by interfaces. If collaboration roles want to share data with other collaboration roles, their interfaces implement interfaces of the collaborations roles it wish to share data with.</p>

Table 5.1: Preview of the solutions to be discussed

5.2.1 Alternative 1: Shared data between Building Blocks

The validation of a card is a task which is natural to separate from the rest of the system and make a unit for this task only. The solution first coming to mind is to make a building block. The card validation function is a small function in the system as a whole, but as mentioned before, it involves quite a few UML elements, as shown in Fig. 5.2. All the different UML elements take the focus away from the function itself, which is to validate the card. A building block *Validate Move* hides the complexity in another level of abstraction. A similar building block *Save Data* can be made for hiding the elements whose function is to update all the variables. Fig. 5.3 illustrates the approach.

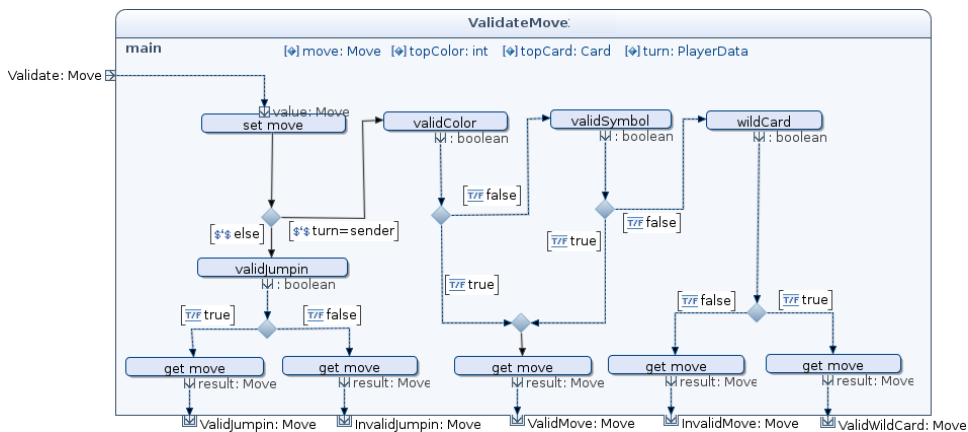


Figure 5.2: Building block Validate Move in Arctis

There is a problem with this solution. The building blocks inside the discard pile partition, *Save Data* and *Validate Move*, do not have access to the variables of the discard pile. It is possible to give these building blocks access to the variables of the surrounding partition. This will, however, break the principle that building blocks should be self-contained, and therefore independent of the surrounding elements and variables.

A possible solution is to let the building blocks maintain their own variables, normally a subset of the variables in the surrounding partition. These variables will have to be mapped to the variables in the parent partition. To differ between variables that are local and variables that are mapped to the surrounding partition, we can declare variables public and private. Public variables are mapped to the variable with the same name located in the surrounding partition. Making changes to the variable in the building block means that the variable in the service are changed simultaneously. Variables that are declared private may only be read and changed from within the building block.

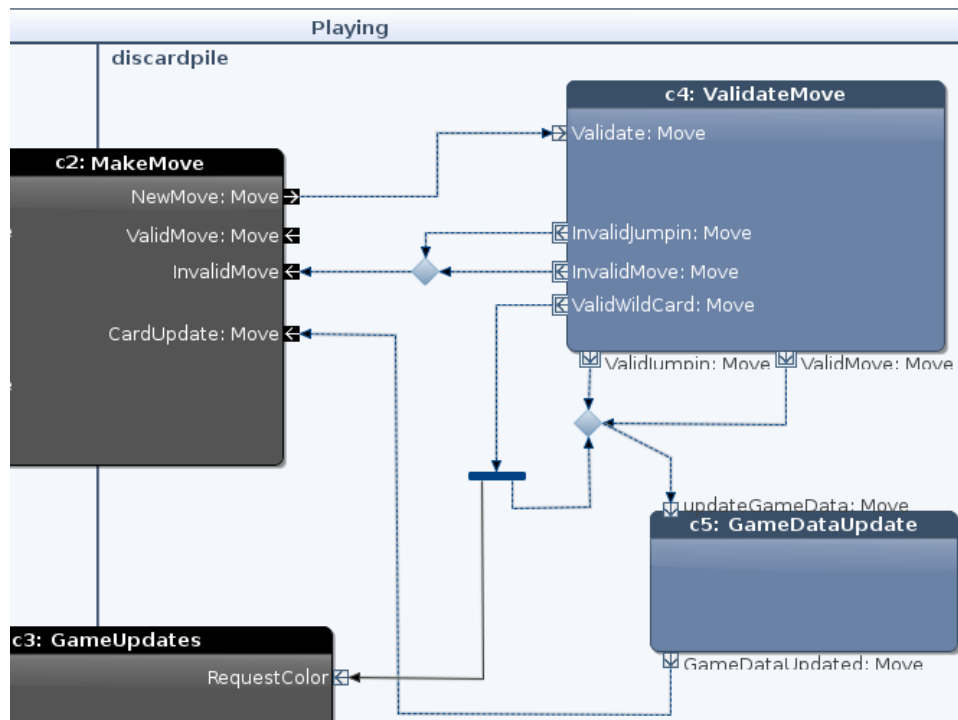


Figure 5.3: Screenshot of playing activity with building blocks

The building blocks are not likely to be reused in other parts of the system, or other systems as they are special for the UNO system, and does not perform a repetitive task. This in itself is a good reason not to use building blocks. Another downside with using building blocks with shared data, is it may lead to competition for resources and in the worst case a deadlock. It is also a possible source for bugs. Say, that variables are declared public that should not be public, and are overwritten in the parent partition because it had a variable with the same name.

5.2.2 Alternative 2: A Flat Specification

The straight forward solution to the problem with building blocks and shared data is to have no building blocks. However, the consequence of not using building blocks is a specification difficult to understand as we get a large number of UML elements on the same level. This is illustrated in Fig. 5.4. As mentioned before, one of the strengths of the collaboration and activities approach is that it should be easy to understand, so this is not a good solution either.

In many cases it may be a good solution to have all elements on the same level, but often there are too many actions. In these cases we need to find another solution, one that are robust and in conformance with the UML standard. One way to go is to try to reduce the number of UML actions to make the specification more clear.

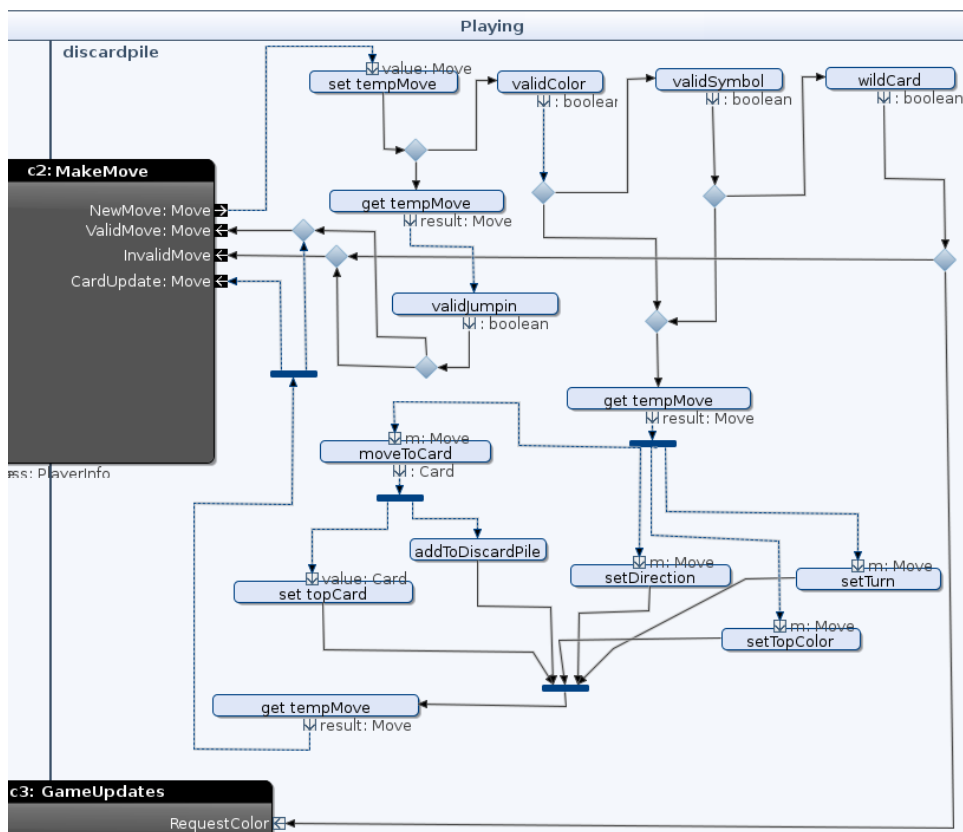


Figure 5.4: Solution with flat specification

5.2.3 Alternative 3: More Comprehensive Call Operation Actions

It is possible to have a single call operation action validate the move and another call operation action to save all data. This means expressing more behavior in Java and less using activities. This approach makes data sharing unnecessary, as all operations are on the same level. It also hides the complexity and less important operations, and makes it easy to understand the specification.

On the downside, there is more to program and less that can be automatically generated. The behavior is described in a less accurate way, making it harder to know exactly what happens by looking at the specifications. For example, for the call operation *save data*, you know that some data has been saved, but not exactly which data. In our UNO specification we can argue that knowing exactly which data has been saved is not important. Having a correct, easy to understand behavior is much more important, and to accomplish this, call behavior actions are a good solution, at least in this example. There may be occurrences where this is not satisfying, where building blocks or a more detailed description are better solutions.

Call operation actions are practical as long as a limited number of output parameters are needed. If several output parameters are needed, the call operation actions should be divided into several call operation actions. First, as Java only permits one return parameter, which means that all parameters must be aggregated into a single object or data structure. To extract information from this object, several operations and control nodes are needed to extract the necessary data from the aggregated object, and then decide what to do.

Validate move has several return parameters, and should for this reason be divided into several call operation actions. The operations in the green rectangle in Fig. 5.5 shows how validate move are divided. First, call operation action *validate Jump-in* or *validate Card* are called, depending on who's turn it is. If the card is a valid card, a call operation action are performed to check if it is a wild card, in which case the player must be requested to choose a color. Listing 5.1 shows the Java code for these three operations.

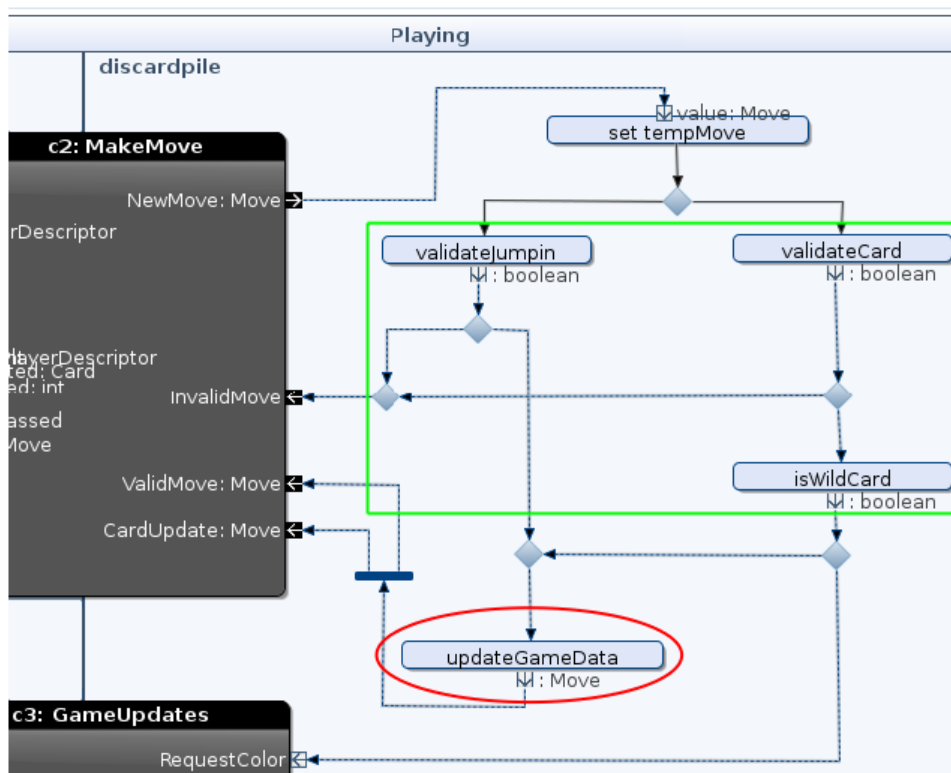


Figure 5.5: Screenshot of call operations actions

Listing 5.1: Java methods corresponding to call operation actions for validating a move

```
public boolean validateCard () {
    if (move.getCard ().getColor () == topColor ||
        move.getCard ().getSymbol () == topCard.getSymbol () ||
        move.getCard ().getSymbol () == WILDCARD) {
        return true;
    }
    else {
        return false;
    }
}

public boolean validateJumpin (Move m) {
    if (move.getCard ().getSymbol () != WILDCARD) {
        return move.getCard ().getColor () == topColor &&
            move.getCard ().getSymbol () == topCard.getSymbol ();
    }
    else {
        return false;
    }
}

public boolean isWildCard () {
    return tempMove.getCard ().getSymbol () == WILDCARD;
}
```

To save the game data, only one input parameter is needed. The move object is provided at output parameter, as it will be needed in the next input parameter nodes. Since it has only one output parameter, only one call operation action is needed. The call operation action is shown in the red circle in Fig. 5.5.

To summarize, call operation actions should only be used when:

- Making building blocks is a bad solution because of shared data and lack of generality.
- Call operation actions can be used to hide unimportant functionality, like set and get variable actions and create object actions.

How call operation actions should be used:

- Small enough call operation action so only one return value is needed.
- Large enough granularity to a reader can understand what it happening without reading the Java code.

5.2.4 Alternative 4: Providing Variables as Input Parameters to Building Blocks

The problem with alternative 1, having building blocks, is the need for data sharing between building blocks and partitions. It is possible to give variables as input to a building block, as illustrated in Fig. 5.6. This solution is not the same as sharing data, as the building block have copies of, not references to, the discard pile variables. This means the variables in the building block do not change if the variables in discard pile changes. Fig. 5.7 shows the details of the validate move building block.

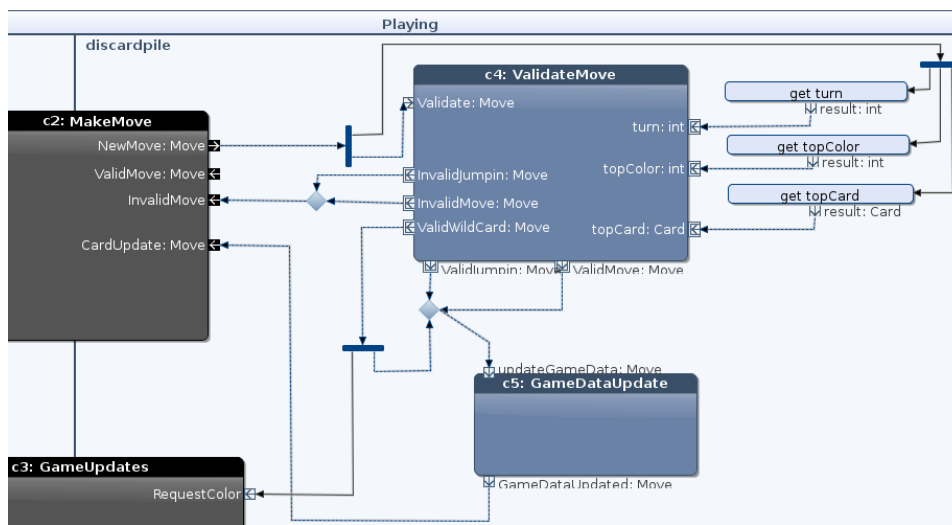


Figure 5.6: Building block with variables provided as input parameters

This solution works fine as long as the variables sent as parameters to the building block does not have to be changed within the building block. In the example of validate move, we only wish to check the variables values to be able to verify if a correct card has been played. In the example of save data, all variables will have to be sent to the building block, then saved in the building block, and finally have to be given as output from the building block so the data may be saved in the partition as well. This is clearly a very bad solution, as it involves more operations than a flat specification.

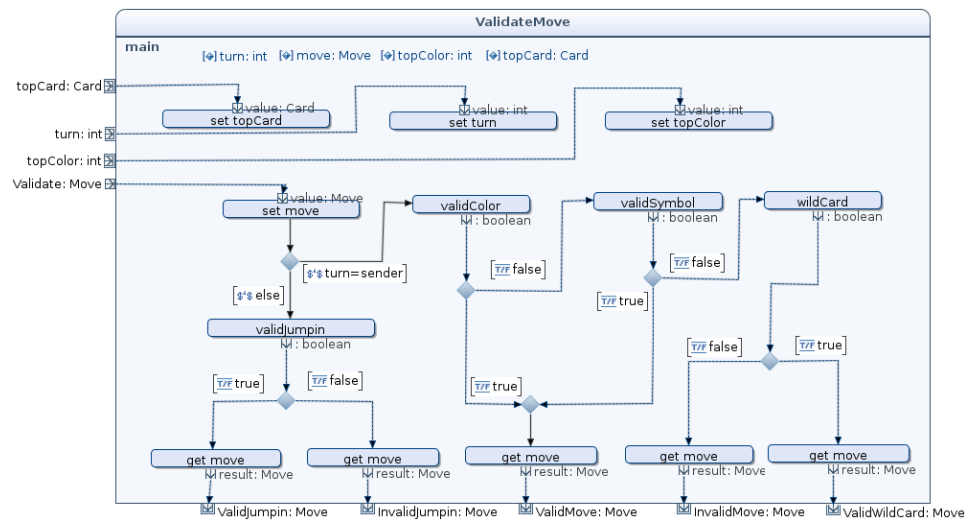


Figure 5.7: Building block *Validate Move* with variables provided as input parameters

5.2.5 Alternative 5: Typed Collaboration Roles

The fifth alternative takes a different approach. The idea is to let the collaboration roles encapsulate data by interfaces. In this way, variables which are needed in several collaborations and building blocks are available where they are needed, without having to send data between collaborations. The approach will be explained with an example, and then discussed in the end of this section.

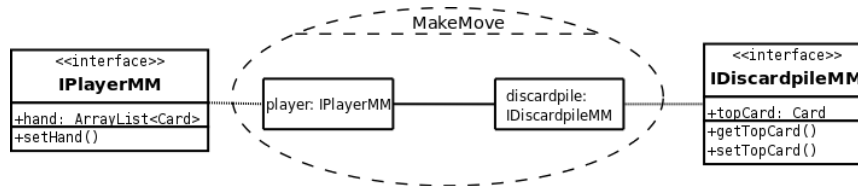


Figure 5.8: Collaboration Make Move

The internal structure of the *MakeMove* collaboration is shown within the UML collaboration in Fig. 5.8. The collaboration consists of two collaboration roles, player and discard pile. An instance playing the player role must possess the properties specified by the interface *IPlayerMM*,¹ and sim-

¹The name *IPlayerMM* is chosen over *IPlayer* to make it easier to understand which interface is discussed, as there will be several *IPlayer* interfaces.

ilarly for the discard pile role. The *IPlayerMM* and *IDrawpileMM* interfaces are shown in the figure, connected to the roles with dashed lines. Note that this is not UML standard notation, but for illustration purposes only.

Fig. 5.9 shows the incomplete activity diagram for the collaboration *Make Move*. The discard pile partition, which corresponds to the collaboration role discard pile, has two call operation actions *setTopCard* and *getTopCard*, referring to the methods specified in the *IDiscardpileMM* interface. It also has a variable, *topCard*, which refers to the variable with the same name in the *IDiscardpileMM* interface. Similarly, the player partition has a call operation action *setHand*, and a variable *hand*, corresponding to the method and variable in the *IPlayerMM* interface.

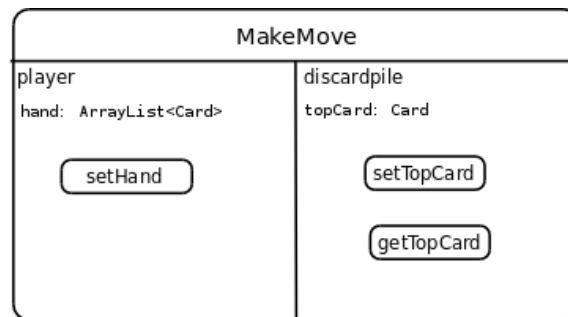


Figure 5.9: Activity diagram for collaboration *Make Move*

The lower part of Fig. 5.10 shows the *MakeMove* collaboration used within the playing collaboration, where the roles in the playing collaboration are bound to the roles in the *MakeMove* collaboration. The same roles are also bound to the roles in the game updates collaboration, which has a structure which is similar to the *MakeMove* collaboration.

The player role in the playing collaboration must possess the properties specified by *IPlayerMM*, in addition to the properties specified by *IPlayerP*. This is solved by letting the interface for the player in *Playing*, *IPlayerP*, implement the *IPlayerMM* interface in the *MakeMove* collaboration, as shown on the left in Fig. 5.10. Similarly, the *IDiscardpileP* interface must implement the *IDiscardpileMM* shown on the right side of Fig. 5.10. As *IDiscardpileP* inherit methods from the *IDiscardpileMM*, the collaboration role discard pile in the *Playing* collaboration can access methods from both interfaces. This is shown in Fig. 5.11, where the call operation actions call methods from both the *IDiscardpileMM* and the *IDiscardpileP* interface.

In addition to implement the interfaces in the make move collaboration, *IPlayerP* and *IDiscardpileP* in *Playing* must also implement the interfaces of the game updates collaboration, as the playing roles must possess the properties of the roles in the game updates collaboration to play those roles. Fig. 5.12 shows that the playing interfaces implement both the interfaces

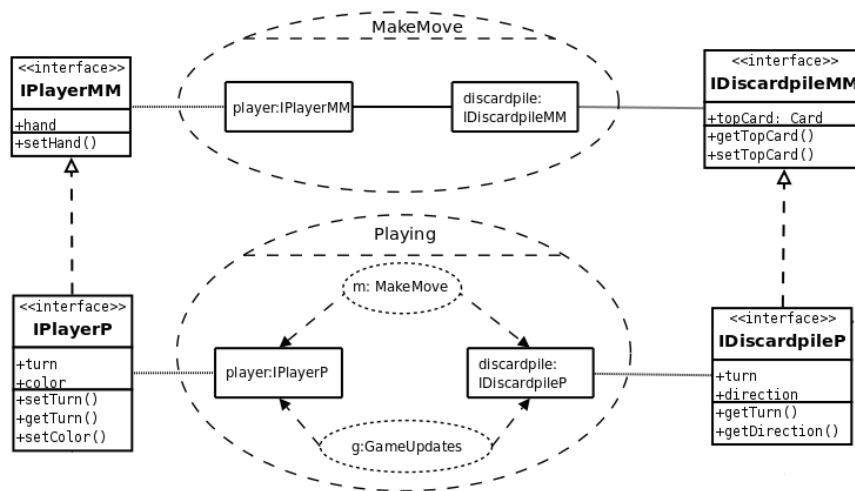


Figure 5.10: Interface implementation

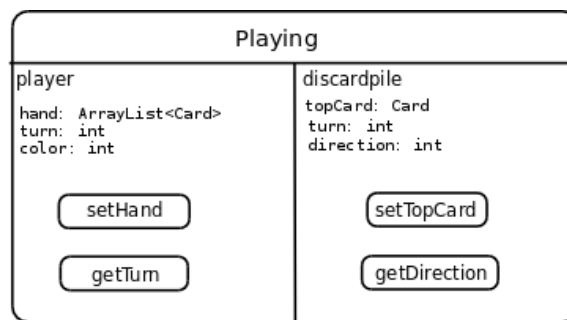


Figure 5.11: Activity diagram for collaboration Playing

of make move and game updates. At the bottom of Fig. 5.12 it is shown that the player role in the system collaboration NTNUNO must possess the properties of the player role in playing, and thus inherit these properties through interface implementation.

The result of this is that variables which are saved in e.g. the NTNUNO collaboration can be read from the *Playing* collaboration, given that methods for reading these variables are provided in the interfaces of the Playing collaboration. This means data which is needed in several collaborations do not have to be sent between the collaborations; defining methods for saving and reading these variables are sufficient. Fig. 5.13 shows how the top level activity NTNUNO looks like when sending data between collaborations when data is avoided. Fig. 6.1 shows the same activity where data is sent between collaborations.

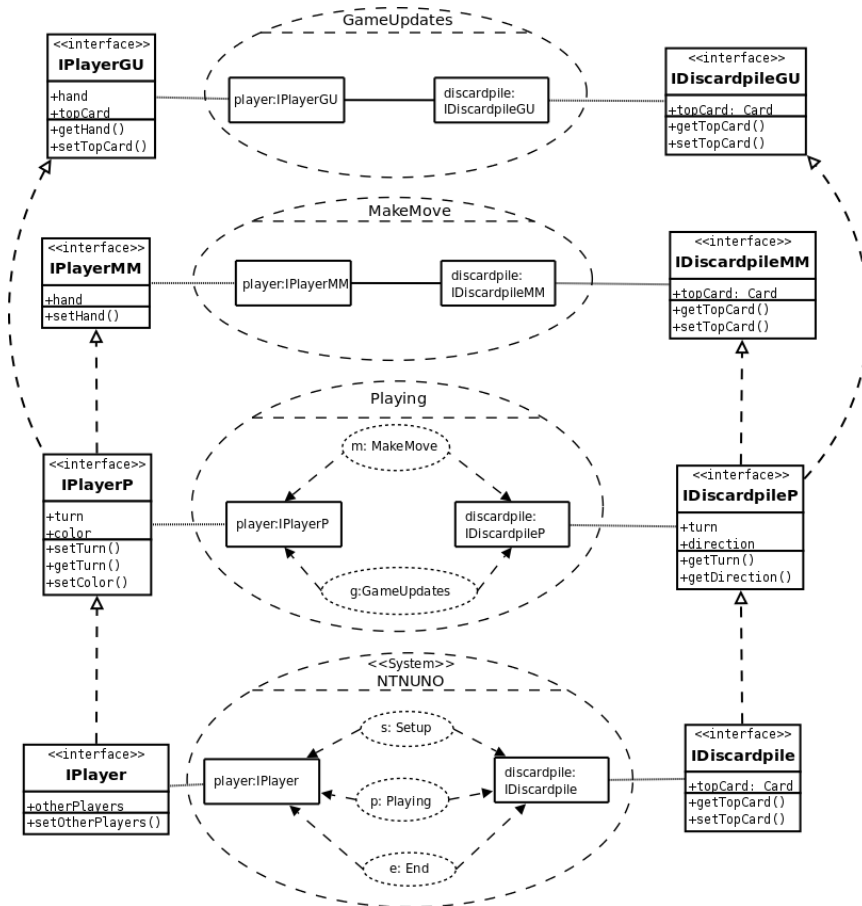


Figure 5.12: Multiple interface implementation

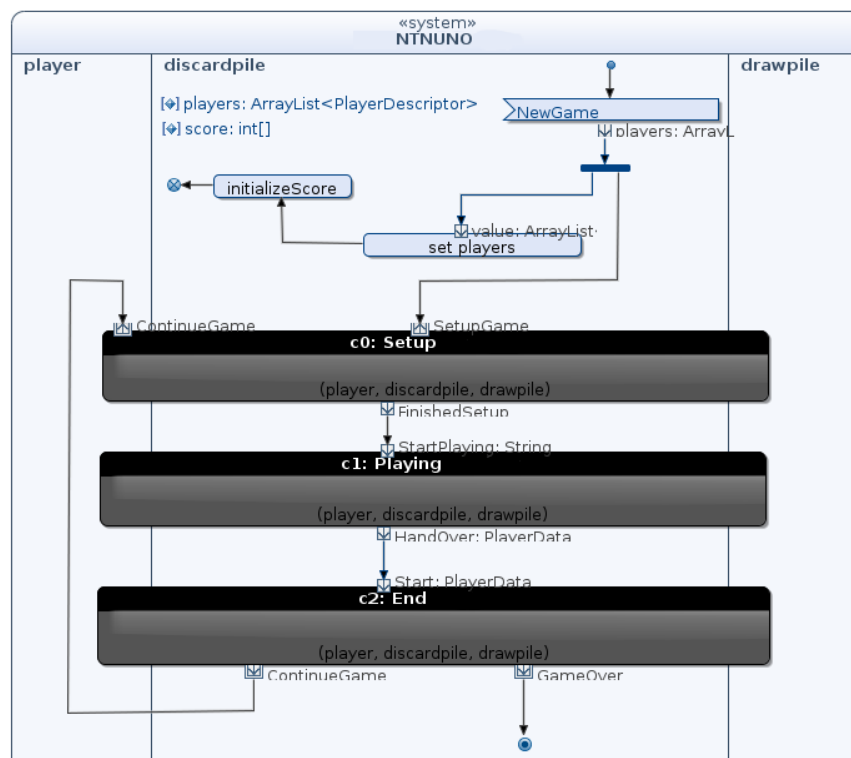


Figure 5.13: System activity using alternative 5

For building blocks like *Validate Move* to get access to these variables, the same approach is used. In this example, we wish to have a validate move block in the discard pile partition in the playing collaboration. Validate move may define its own parameters and variables, in an interface, which the entity playing the discard pile role must implement. This is illustrated in Fig. 5.14.

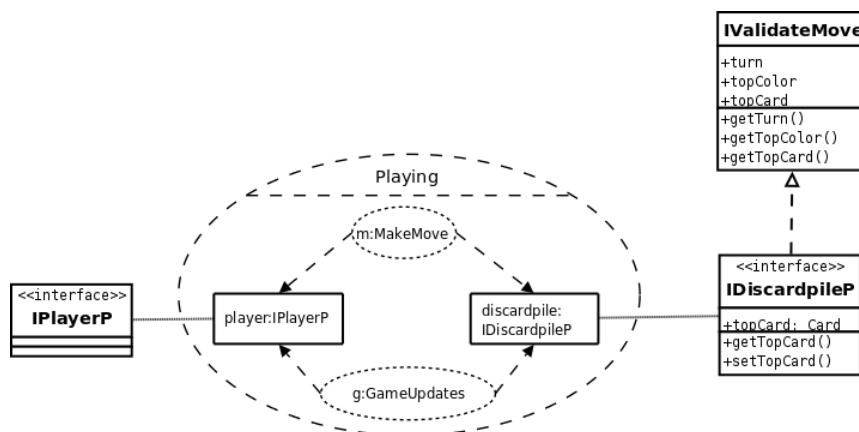


Figure 5.14: Inheritance of building block

This solution enables data sharing between collaborations and building blocks in a smooth and intuitive way, which gives a clearer specification as it is not necessary to model the exchange of data between collaborations and building blocks. Modeling of behavior will not be driven by where data is available, but by where it is natural for a user to place the behavior to make a clear specification. Thus, this solution will make the specification more intuitive as sending data between collaborations seems unnecessary and inefficient.

The multiple interface implementation process can be highly automated, and Java access methods can be generated automatically. This means that the complexity are hidden from users. However, how this can be solved in Artcis is not a focus in this work.

5.2.6 Discussion

While specifying the behavior of UNO, it is important that the specifications are compact and easy to understand. This means that a reader unexperienced with UML collaborations and activities should understand it without a throughout explanation. Reuse is also important, as it is one of the basic ideas behind collaborations and activities. A system consists of a number of building blocks that may be reused in new service specifications.

A third issue worth considering is how detailed the specification should

be. In general, we wish the specification to be detailed enough to describe all important decisions and operations.

To find out which solution are best, we have listed a few criteria.

- The behavior must be clear and easy to understand.
- Reuse must be possible.
- The specification should be in conformance with the UML standard.
- Reasonable granularity. Not too fine and not too coarse.

The solution first proposed, building blocks with shared data, are intuitive to understand. However, shared variables may have unwanted side effects. The solution is not in conformance with the UML standard.

A flat specification may take time to understand, even for a person experienced with these kind of specifications. The finer granularity, the more difficulties with understanding the specifications. However, there is no need for shared data, and the solution is in conformance with the UML standard.

More comprehensive call operation actions does not involve shared data, and it is possible to hide complexity. However, hiding too much of the complexity means a less descriptive behavior, which again means that it does not describe in detail what happens. This is not necessarily a bad thing as it is not all behavior that are important to know in detail. Another downside is that less of the behavior is automatically generated using Arctis, and more work has to be done manually by implementing the functionality in Java.

The solution with providing variables as input parameters to building blocks has a drawback as it can not be used when the input parameters need to be changed inside the building block. It may also seem a bit noisy if the building block needs a large number of input parameters, as each of these variables will have to be read outside the building block, and saved inside the building block. Advantages with building blocks are that there is no need for shared data, and it is possible to have a fine granularity within the building block, while the behavior specification look nice and clear from outside the building block.

Typed collaboration roles are in conformance with the UML standard, and makes it possible to share data between building blocks and collaborations, while ensuring possibility of reuse. Tbl. 5.2 summarizes the discussion.

5.2.7 Conclusion

In the example with validate move, the typed collaboration roles solution seems like the best choice, as it does not have any serious drawbacks. The

Solution name	Pros and cons
Share data between building blocks	<ul style="list-style-type: none"> + Intuitive and easy to understand - Building blocks depend on surrounding partition - Easy to make mistakes
A flat specification	<ul style="list-style-type: none"> + No need for shared data - Requires time to understand specification
More comprehensive call operation actions	<ul style="list-style-type: none"> + No data sharing + Hides complexity + Easily understood - More to implement, less generated automatically - Less descriptive behavior - Person looking at specifications do not understand what happens
Provide variables as input parameters to building blocks	<ul style="list-style-type: none"> + No need for shared data + Can have a fine granularity - Seems unnecessarily complex - Can not be used when the input data need to be changed
Typed collaboration roles	<ul style="list-style-type: none"> + Conformance with UML + Makes it possible to share data between collaborations and building blocks - May get race conditions

Table 5.2: Possible solutions with their pros and cons

solution with shared data should be avoided, as the typed collaboration roles solution solves the problem in a better way. Call operation actions, data input to building blocks, and a flat specification are all used in the UNO specification, but for the problem described in this section they was not ideal.

An example of where call operation actions can be used is with *save data*. Even if it is possible to make a building block for *save data* with typed collaboration roles, call operation actions may be a better solution. A single call operation action may be just as easy to understand as a building block, and defining exactly which data that is saved in the activity diagram is not important.

Specification of UNO — Part 2: Behavior

This section describes the behavior of UNO in the form of activity diagrams. Sect. 6.1 describes the top level behavior, while the three next sections describe the behavior of the three collaborations *Setup*, *Playing* and *End*. Then, in Sect. 6.5 lists the simplifications done on the UNO implementation.

6.1 System View

A screen shot from Arctis of the activity diagram describing the behavior of the NTNUNO collaboration is shown in Fig. 6.1. This activity diagram connects the collaborations *Setup*, *Playing*, and *End*. When the system starts, we immediately arrive at a receive signal action. Upon arrival of signal *New Game* containing the list of the participating players, we arrive at a fork node. The leftmost outgoing edge continues to save the players to the player variable, before creating the score list. The rightmost edge that leaves the fork node starts the *Setup* collaboration. The *Setup* collaboration performs the initialization of the game, like dealing cards to players and choosing a player to start the game.

In the *Playing* collaboration the participating roles need the data that are created in the *Setup* collaboration. The players need information about the card they have been dealt, whose turn it is, other players, and top card on the discard pile. The discard pile needs the same information except the card details, and the draw pile needs to get the remaining cards in the draw pile. This information is provided from the *Setup* collaboration to the input parameter nodes *Initial Data*, *Draw Pile*, and *Start Game*. When this data is available, the *Playing* collaboration starts.

When a player has no cards left, the hand is over. The output parameter node *HandOver* contains data that will be needed in the *End* collaboration. This includes the winner of the game, and the remaining cards in both the discard pile and the draw pile. The *End* collaboration starts when input tokens are available in the *Hand*, *FinalGameData*, and *Score* parameter nodes. If a player has reached 500 points and thus wins the game, the *End* collaboration terminates via the output parameter node *GameOver*. Then the NT-NUNO collaboration is terminated. If no one has won the game, the score is updated, and the *End* collaboration terminates via the output parameter node *RestartGame*. This token is provided to the *Setup* collaboration via the input parameter node *ContinueGame*.

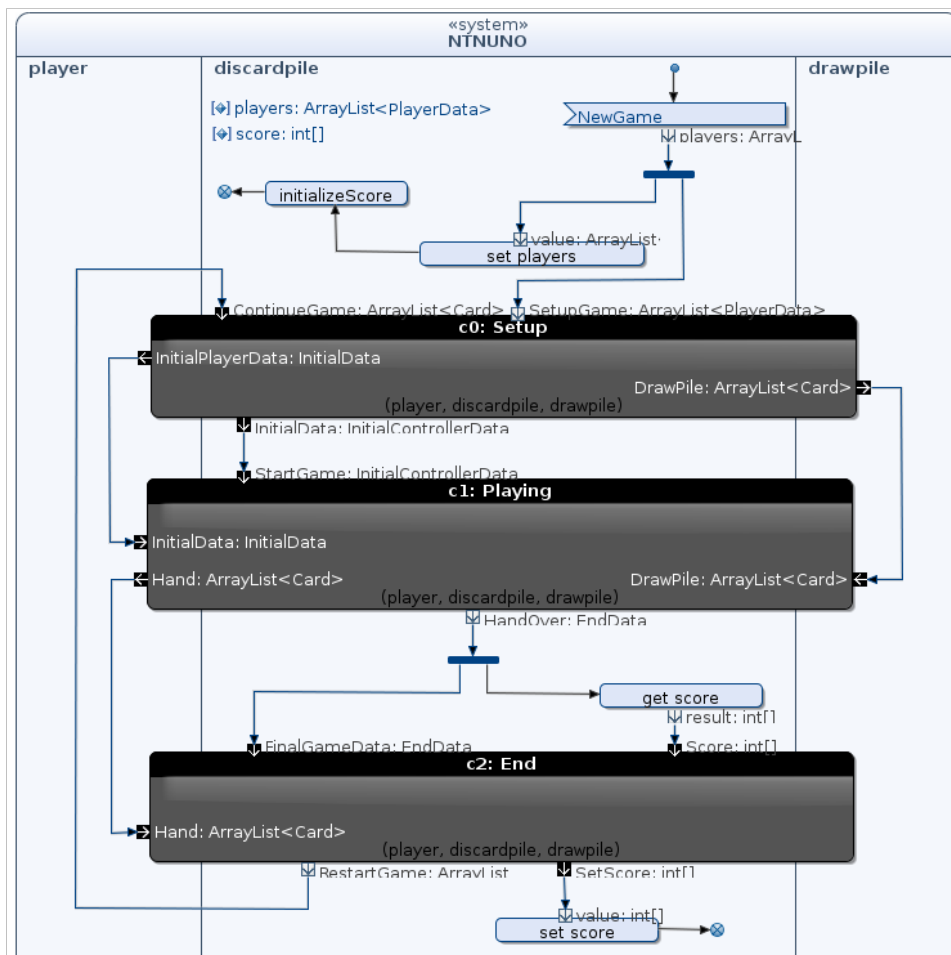


Figure 6.1: System activity

6.2 Collaboration Setup

The Setup activity describes the behavior of the Setup collaboration, and is shown in Fig. 6.2. As input, the activity receives the players participating in the game via the starting pin *SetupGame*. The input token immediately arrives at a fork. The fork has three outgoing edges:

- The first edge starts the collaboration *DistributePlayers*, which informs all players about the participants in the game.
- The next saves the players to the *players* variable. Then a player is randomly chosen to start the game, and the *SelectTurn* collaboration is started.
- The third outgoing edge from the fork enters the partition *draw pile*, upon which a number of things happens. First, the 108 cards used in the UNO game are created to form a draw pile. Then a card is randomly chosen to start the game. (If the card chosen is a wild draw four, it has to be put back in the pile, and another card is chosen). When a card is successfully chosen we arrive at a fork, where one outgoing edge starts the collaboration *Deal*. In this collaboration all players are dealt seven cards each. The other edge starts the *SelectTop-Card* collaboration, where all players are informed of the card chosen to begin the game.

Note that all collaborations that are started are multi-session collaborations, seen from the discard pile and the draw pile. Thus, when entering the collaborations, a selection of session must be done. When entering the sub-collaborations in the *Setup* collaborations, all sessions are chosen using **select all**. This is not yet implemented in Arctis, but is shown in Fig. 6.2 as an illustration.

When a *Deal* collaboration is terminated, we arrive at a decision node. A guard checks if more *Deal* collaborations exists in the guard **exists c0: active**. If some collaborations still are active, the flow is terminated, as it means that not all players has been dealt cards yet. If no *Deal* collaborations are active, the *Setup* collaboration can finish.

As discussed in the previous section, the players need data about players, top card, who start the game, and which card they have been dealt in the *Playing* collaboration, and thus has to be sent transferred from the *Setup* collaboration. When all this information is ready, it is packed in an object, and sent to the output pin *InitialPlayerData*. The discard pile need the same data, with the exception of the hand. An composite object is created, which is sent to the output pin *InitialData*. The draw pile will need the *drawpile* variable in the *Playing* collaboration. This is sent via the output pin *Draw-Pile*.

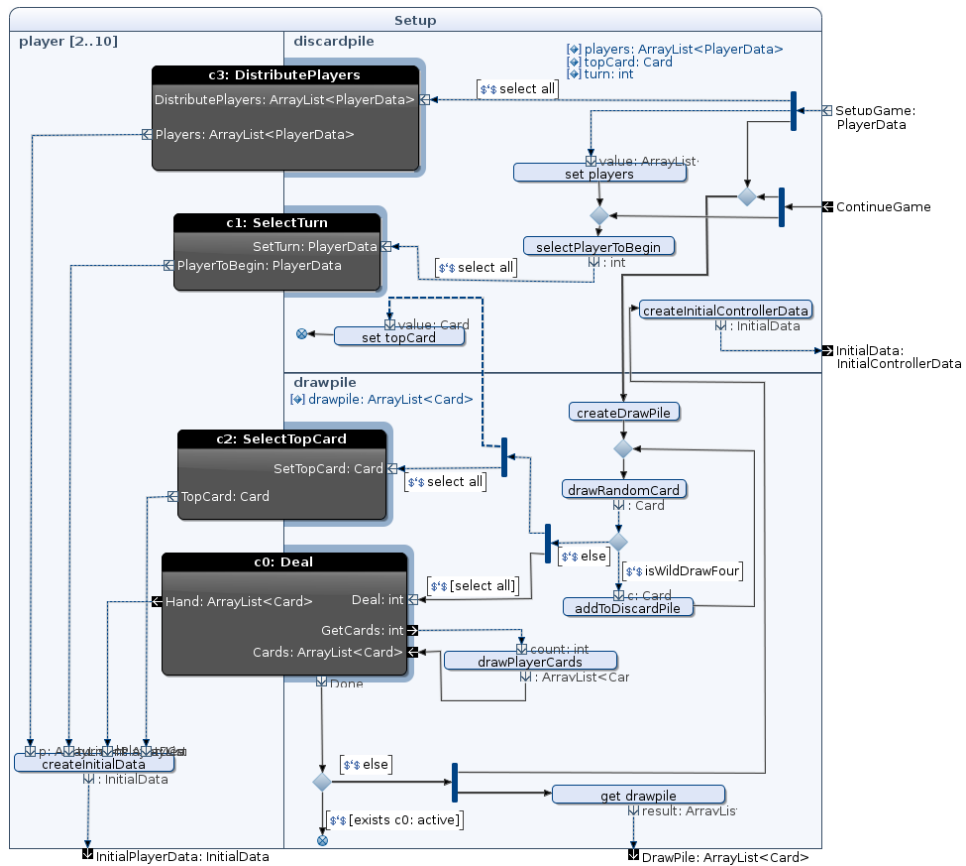


Figure 6.2: Activity Setup

If the game is played in several rounds until a player reach 500 points, the *Setup* collaboration may receive a token via input pin *ContinueGame* to make ready for another round. As the players already have information about their opponents, and the discard pile already have saved the players, these two functions are omitted, and else the same flow is followed as when receiving a token via the *SetupGame* input pin.

The *Distribute Players* activity is shown in Fig. 6.3. The activity starts when the input pin *DistributePlayers* receives a token. The token is sent to the discard pile which contains a fork. One edge is sent to the send signal action, which send the list of the participating players to the user interface. The other edge terminates the collaboration via the output pin *Players*.

The activities *Select Top Card* and *Select Turn* is similar to the *Distribute Players* activity. The activities are shown in Fig. 6.4 and Fig. 6.5, respectively.

The activity *Deal* is shown in Fig. 6.6. The activity starts with the arrival of a token of integer type in the input pin *Deal*. The token immediately arrives at the output pin *GetCards*. This is necessary as the draw pile in the

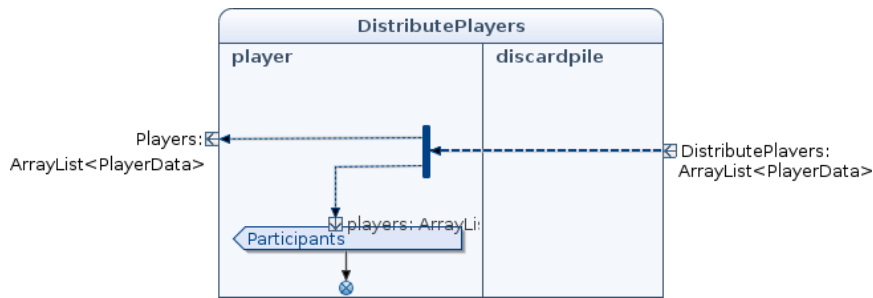


Figure 6.3: Activity *Distribute Players*

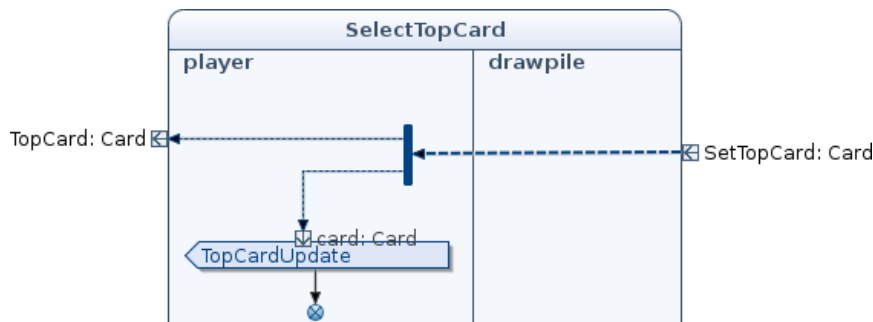


Figure 6.4: Activity *Select Top Card*

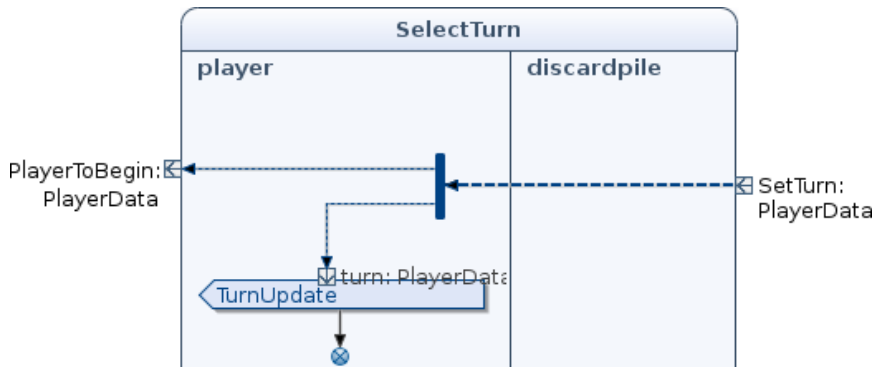
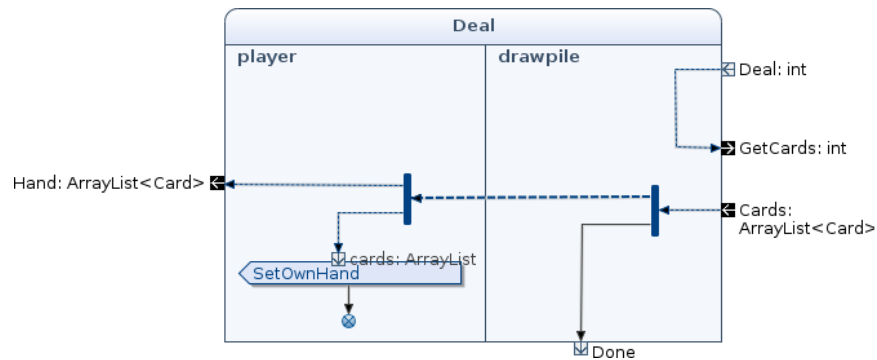


Figure 6.5: Activity *Select Turn*

collaboration has no access to the draw pile attribute. The cards arrives at the input pin *Cards*, and arrives at a fork. One of the outgoing edges arrives at the player, where the hand is sent to the interface, and leaves the collaboration via the outgoing pin *Hand*. The other outgoing edge terminates the collaboration.

Figure 6.6: Activity *Deal*

6.3 Collaboration *Playing*

The playing activity is shown in Fig. 6.7. To reduce the size and complexity, this diagram is modeled assuming use of typed collaboration roles, as described in Sect. 5.2.5. With typed collaboration roles, data can be shared between collaborations and building blocks, and more of the behavior in the *Playing* collaboration can be put in building blocks and collaborations.

As the *Playing* collaboration is a rather complex collaboration with many variables, several sub-collaborations and many things happening, the effect of using typed collaboration roles is in this case a significantly smaller and less complex activity diagram. However, it still is a complex activity diagram, and will not be described in details. The sub-collaborations and the most important building blocks will be described in detail in the next sections. The playing activity starts when a token is available in the input pin *StartPlaying*. We arrive at a fork, where the control token is duplicated twice. One outgoing flow is sent to the draw pile and arrives at a join node. The second outgoing flow creates the initial data, and activates the player buttons at the players side.¹ The third flow starts the collaborations *Make Move* and *Game Updates*. Now, the players can start playing UNO.

In the player partition, mainly two tasks are performed. One is to update game data when there has been a status change and informing the external user interface of these changes by sending signals. This is done in a number of building blocks. The second is to accept input from the user interface, and perform actions according to the type of input. For example, if a draw button is pushed, the collaboration *Draw Card* is started. If the player plays a card, the card is sent to the *Play Card* collaboration. Input from the user is handled in the building block *PlayerInput*.

¹For clarity, the control flow is not drawn across the whole diagram. Instead, the control flow goes to a circled *A*, and is continued in the similar token on the player side.

In the discard pile partition, the cards played are validated, and if they are valid, game data is updated. Further, if a wild card is played, the discard pile requests a color from the player playing the wild card. This happens in the *Color Dialog* collaboration. In the collaboration *Turn Pile*, the discard pile is emptied, with the exception of the topmost card, and added to the draw pile.

The draw pile makes sure that only one player at a time can draw a card. This is done using a join node, which received a input flow when the activity started. When a player wants to draw a card, he has to ask for allowance. This happens via the output pin *getAllowance* from the *Draw Card* collaboration. Then, the second token arrives at the join node, and the join node can fire, granting the user access to draw a card. Now there is no tokens at the join node, so if another player wants allowance to draw a card, he must wait until the player holding the allowance to return the token to the join node.

6.3.1 Collaboration *Make Move*

The *Make Move* collaborations handles mixed initiative, a problem common in reactive systems, which is hard to get right. Arctis will provide solutions to help the engineer handle mixed initiatives, but this function is still under development. For this reason we have specified the ESMs (External State Machines) for the *Make Move* collaboration instead of the activity diagrams. ESMs describe the externally visible behavior on each of the participants in the collaboration. An external state machine document in which sequence the pins of an activity may be invoked [19]. This description is sufficient for an engineer to use the collaboration as a building block, as we have done with the *Make Move* collaboration in the *Playing* collaboration described in the previous section. Knowing the internal details is not necessary. Fig. 6.8 shows the collaboration *Make Move*, to which we will describe the ESMs.

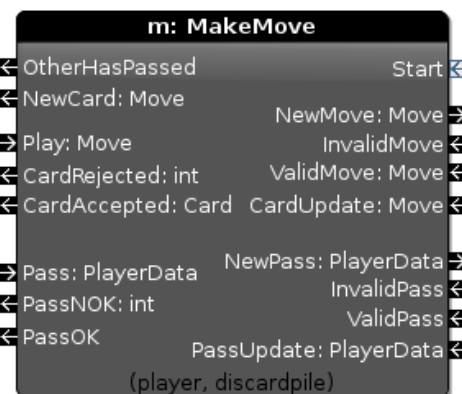
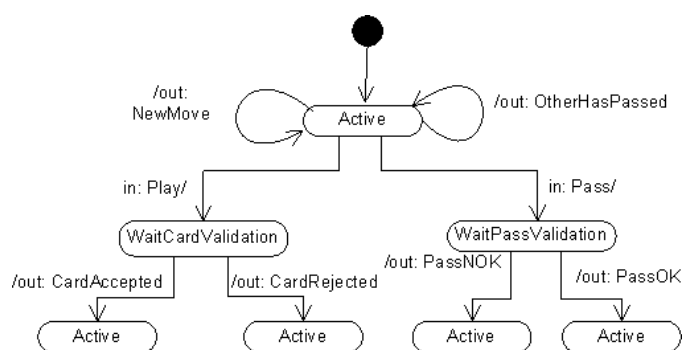
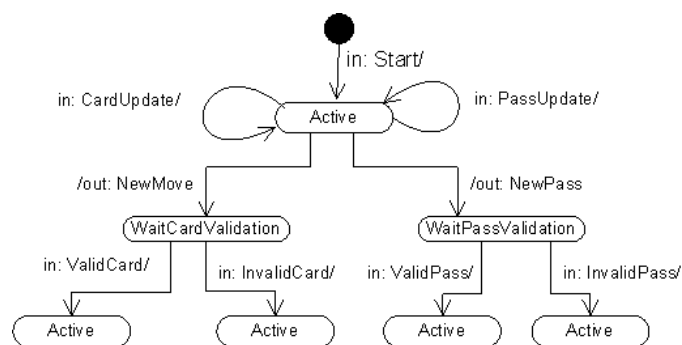


Figure 6.8: Collaboration *Make Move*

Figure 6.9: ESM *Make Move*, *Player* partition

The ESM for the player partition in the *Make Move* collaboration is shown in Fig. 6.9. This refers to the pins on the left in Fig. 6.9. Seen from the player side, the activity starts in state *active*. At any time, the player can receive updates from the discard pile, telling that another player has made a move. This is expressed by the self-transitions */out: NewMove* and */out: otherHasPassed*, which has the state *active* as source and target. Note that the slash is used to distinguish between ESM transitions triggered from outside or inside the block. When the player plays a card, the input pin *Play* is activated, and the state *WaitCardValidation* are entered. Then the card is either accepted or rejected. Similarly, when a player passes, the pass can either be accepted or rejected.

Figure 6.10: ESM *Make Move*, *Discard Pile* partition

The ESM for the discard pile partition in the *Make Move* collaboration is shown in Fig. 6.10. This refers to the pins on the right in the collaboration shown in Fig. 6.9. The externally visible behavior is triggered from the outside via starting pin *Start*. When the block is active, the discard pile may at any time receive a signal when a player has played a card or pushed a button via the output pins *NewMove* and *NewPass*, shown with the transition labels */out: NewMove* and */out: NewPass*. When the discard pile receives a *NewMove* it will validate the card, and respond with either a *ValidMove* or

InvalidMove. When a player has made a valid move, the other players are informed, represented in the ESM with the self-transitions *CardUpdate* and *PassUpdate*.

6.3.2 Collaboration *Game Updates*

The activity *Game Updates*, shown in Fig. 6.11, describes the behavior of the corresponding collaboration. A player has one collaboration with the discard pile, while the discard pile has one collaboration with each of the players. Thus, from the discard pile point of view, it participates in multi-session collaborations.

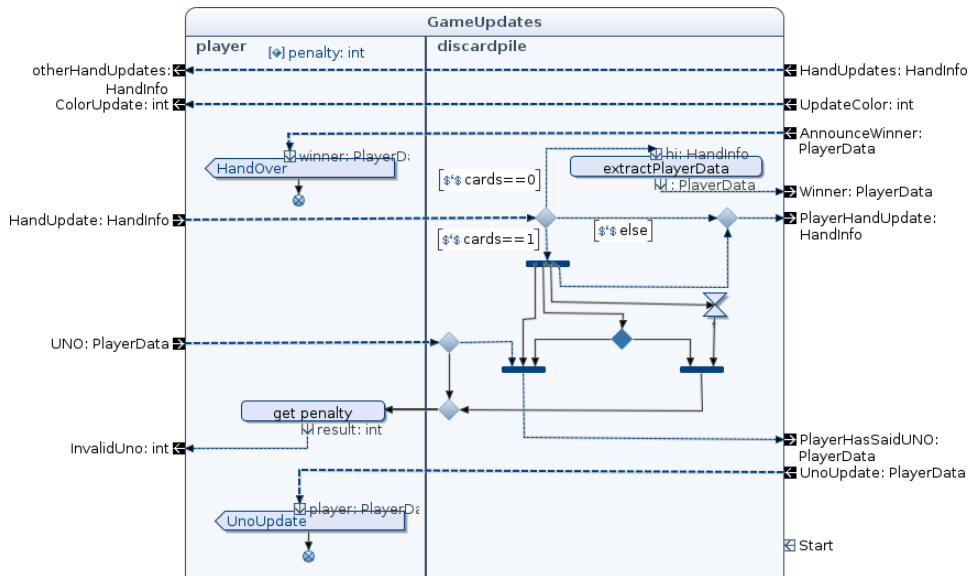


Figure 6.11: Activity *Game Updates*

The task of these collaborations is to distribute data among the players. This includes how many cards the player has at hand, notify other players when a player has yelled “UNO”, and color updates when a wild card has been played. When the player has chosen a color, the controller will inform all other players of their choice. The color is chosen in the collaboration, *Color Dialog*.

Another central task of the *Game Updates* collaboration is to check if the players remembers to yell “UNO” when they have only one card left, and to announce the winner if a player has no cards left. When players update their hand count, the controller checks how many cards the player has left. If the player has one card left, a timer is started, and a token is placed in a waiting decision node. The token remains there until one of the downstream joins may fire. If the player yells “UNO” before the time

expires, the leftmost join node fires, and the controller will inform the other players that a player has yelled "UNO". If the time expires, e.g. after five seconds, the right join node will fire, causing the player to receive a penalty. The same penalty applies if a player yells "UNO" when he has more than one card left.

If a player has no cards left, the player has won the game round. The controller will announce the winner to all other players. If a player has more than one card left, the controller will announce how many cards the player has to all participating players.

6.3.3 Collaboration *Draw Card*

The activity *Draw Card* is shown in Fig. 6.12. The partitions *player* and *pile* corresponds to the roles *player* and *draw pile*. The activity starts when there is a token available in the input parameter node *Draw*. The input token is a data token containing an integer specifying how many cards to draw. From the input pin, we arrive at a fork, upon which a token is sent to the output parameter node *getAllowance* asking for allowance to draw a card, and simultaneously start the *Counter* building block. When allowance is granted, the pile status is checked. If the draw pile is empty, the pile has to be turned, before a card can be retrieved from the draw pile. The counter block checks if more cards should be drawn. If not, the token is retrieved and the activity ends. Else, the whole process is repeated until all cards have been drawn.

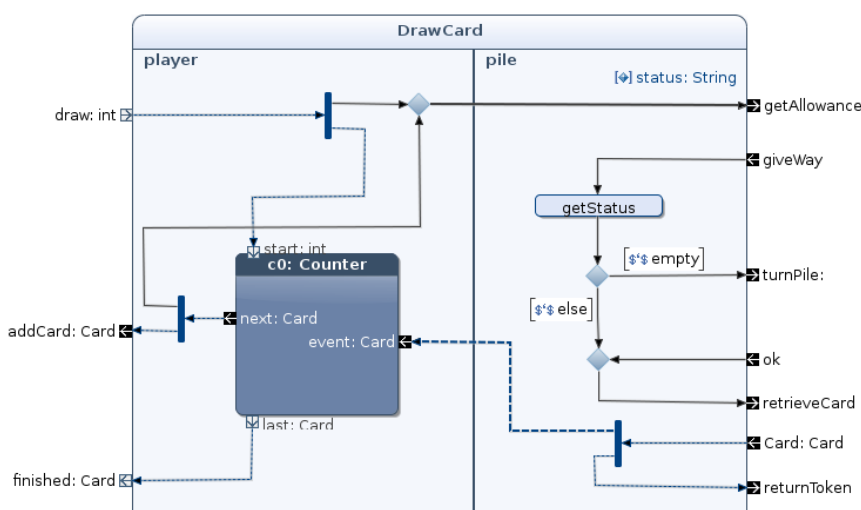


Figure 6.12: Activity diagram draw card

Building block *Counter*

A screenshot of the *Counter* building block is shown in Fig. 6.13. The building block makes sure that a task, in this case drawing a card, is performed a specified number of times. The counter building block is described in more detail in Chapt. 4.

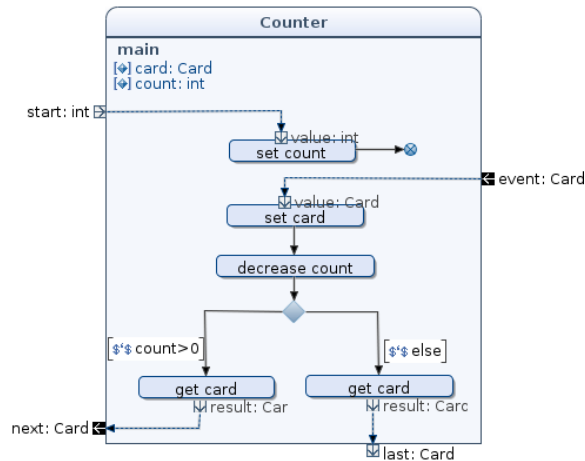


Figure 6.13: Building block *Counter*

6.3.4 Collaboration *Color Dialog*

The activity color dialog in the player partition of activity *Playing* is shown in Fig. 6.14. This dialog is activated when a player plays a wild card, and the controller requests the player to choose a color. A *Choose Color* signal is sent to the user interface requesting the player to choose a color. An accept signal event action waits for an incoming signal *Color Choice* holding the chosen color.

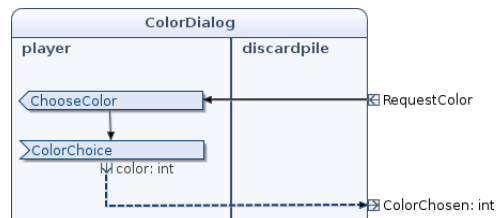


Figure 6.14: Activity *Color Dialog*

6.3.5 Collaboration *Turn Pile*

The *Turn Pile* activity is shown in Fig. 6.15. The partitions draw pile and discard pile are bound to the draw pile and discard pile roles. The activity starts when the input is available in the starting node turn pile. An output streaming node requests the cards from the discard pile, except the topmost card. An input streaming nodes contains a list with cards. If the list is empty, an error is output. Else an OK message containing the list of cards is returned.

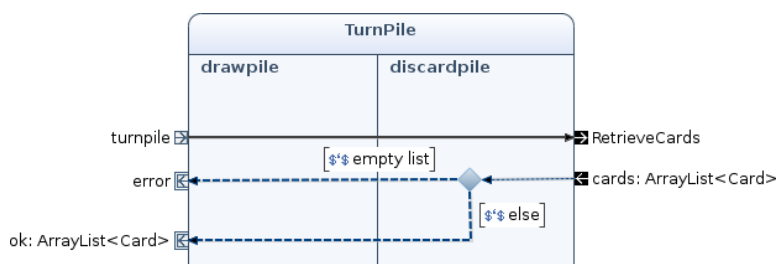
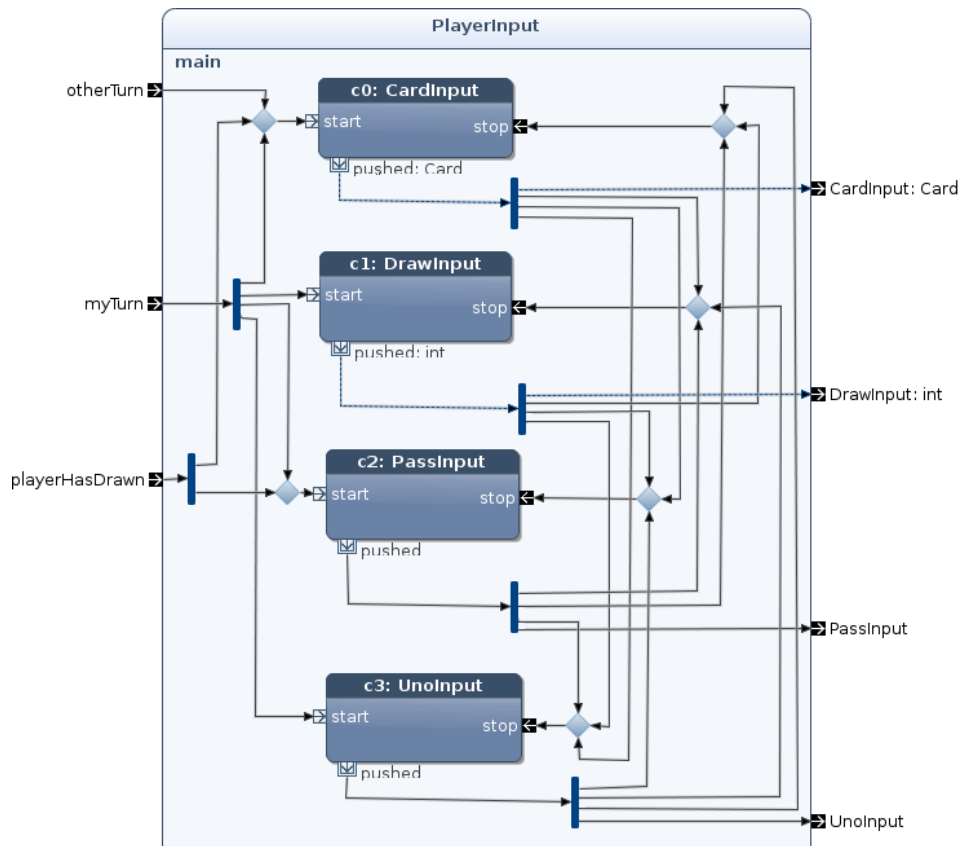


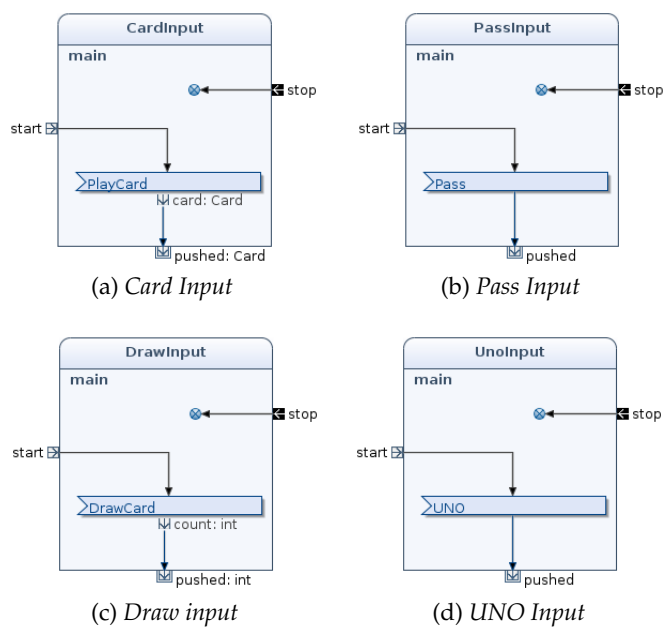
Figure 6.15: Activity *Turn Pile*

6.3.6 Building Block *Player Input*

The building block *Player Input* is shown in Fig. 6.16. The purpose of this building block is to accept player input from the user interface. The idea is when one input signal is received, no other signal is accepted until the previous signal has been processed. For example, when a player plays a card, the card input, pass input, draw input, and UNO input is deactivated. When the played card has been accepted or rejected, the input building blocks are activated again, and they continue to wait for signals from the user interface. Draw and pass input are only active when it is the player's turn, while players can play a card even if it is not their turn. In this case the player does a jump-in. If the players has drawn a card, the players can not draw another card until next time it is their turn.

Fig. 6.17 shows the activity diagrams for the input building blocks used in the *Player Input* activity shown in Fig. 6.16. They differ only in what signal the accept signal action will accept. The *Draw Card* signal contains an integer parameter specifying how many cards to draw. The *Play Card* signals specifies which card has been played. This parameters are extracted from the signal, and returned from the respective building blocks.

Figure 6.16: Activity *Player Input*

Figure 6.17: Input blocks used in the *Player Input* building block

6.3.7 Building Block Validate Move

The activity diagram for the building block *Validate Move* is shown in Fig. 6.18. In this building block, we assume that data is shared with the *Playing* collaboration using typed collaboration roles. The building block starts at the arrival of a *Move* token in the input pin *validate*. First, the *Move* token is saved, and a check is performed to find if it is a jump-in or a regular move. If it is a jump-in, the jump-in is validated, and the building block is terminated via input pins *ValidJumpin* or *InvalidJumpin*, depending on if it was valid or not. The jump-in is valid if card and color matches the top card at the discard pile.

If it is not a jump-in, the card is valid if has a valid color, valid symbol, or if it is a wild card. If it is a wild card, the building block terminates via the output pin *ValidWildCard*. Else it terminates via one of the output pins *ValidMove* or *InvalidMove*.

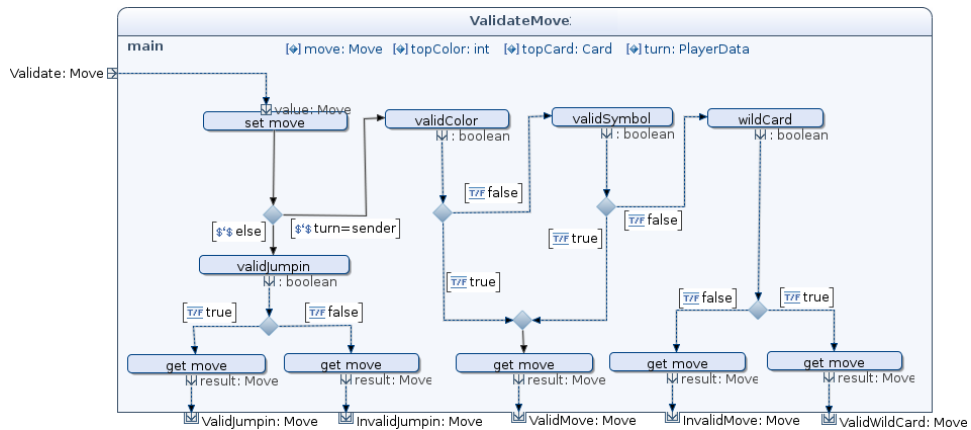


Figure 6.18: Building block Validate Move

6.4 Collaboration End

The *End* collaboration is activated when a player has got rid of all his cards in the *Playing* collaboration. The activity diagram for the *End* collaboration is shown in Fig. 6.19. The sub-collaborations of the *End* activity is shown in Fig. 6.20.

The activity starts when data is available in the input pins *Hand*, *FinalGameData*, and *Score*. From the input pins *FinalGameData* and *Score*, the variables *winner* and *score* are set, respectively. Then the tokens are terminated. From the input pin *Hand*, each player starts a collaboration *CalculateScore* with the discard pile. In this collaboration the points from the remaining cards on the opponents hands are calculated according to the rules listed in Sect. 3.1.3. The activity diagram for the collaboration *CalculateScore* is shown in Fig. 6.20a. After the score has been calculated, the collaboration ends, and the player's points is added to the winner's score. When score for all players has been calculated, a check is performed to see if a player has reached 500 points. If a player has reached 500 points, the game is over, and the winner has to be announced. This is done in the *Announce Winner* collaboration, shown in Fig. 6.20b. Then the *End* activity is terminated via the output pin *GameOver*.

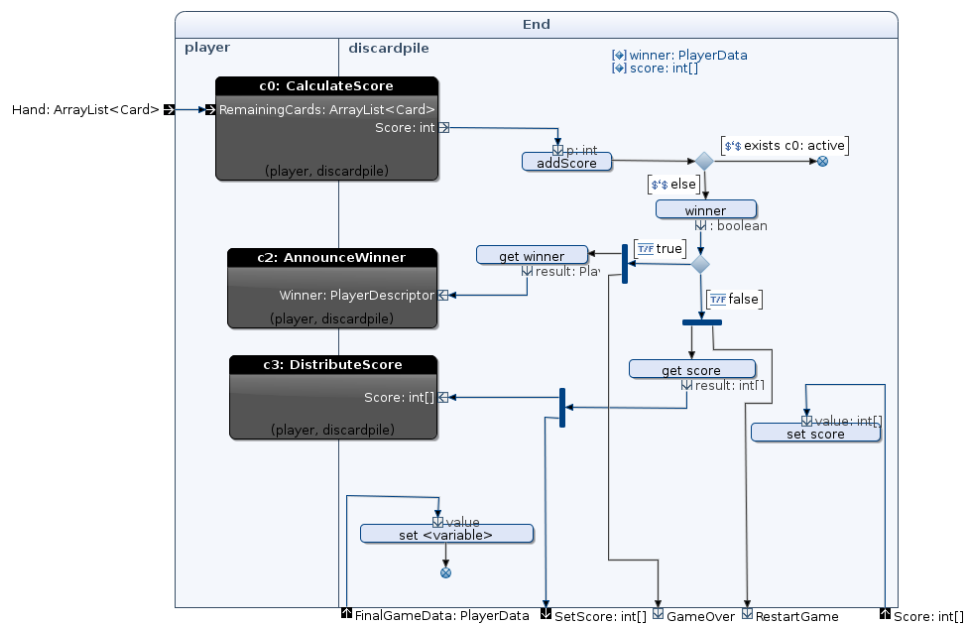
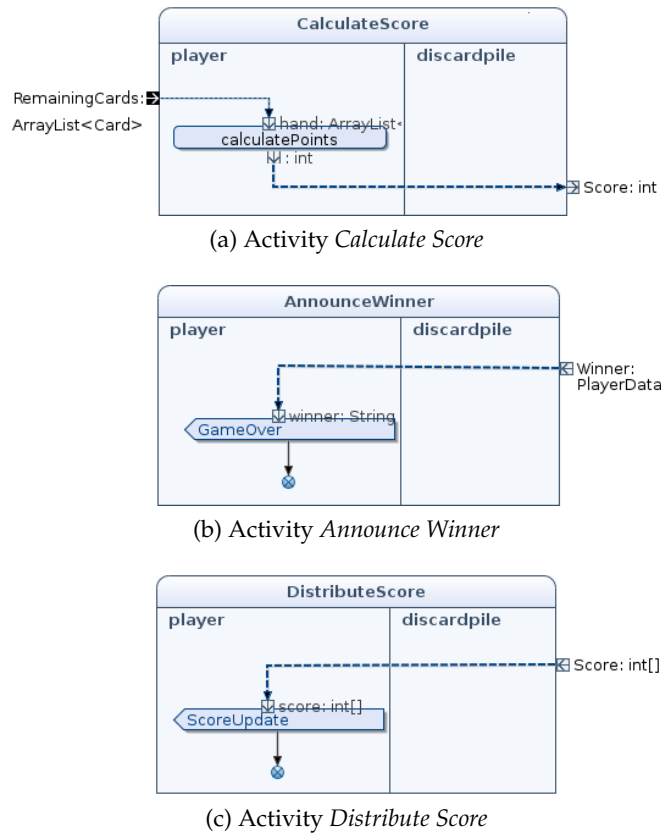


Figure 6.19: Activity *End*

If none of the players has reached 500 points, the updated score has to be sent to all participating players. This is done in the collaboration *DistributeScore*, shown in Fig. 6.20c. After the score has been distributed,

Figure 6.20: Sub-collaborations of the *End* collaboration

the *End* collaboration is terminated via the output pin *RestartGame*.

6.5 Simplifications

As designing a fully working UNO system was not the primary goal of this work, we have designed a slightly simplified version of UNO. Some of the functionality that we have excluded is listed below.

- Playing history. A player should get more detailed information on what has happened in the user interface.
- It should be possible to choose if the player first to get rid of all his cards wins the game, or if the player first to reach 500 points wins.
- No exception handling. For example, no functionality exists to handle that a player leaves the game in the middle of a game.
- It should be possible to choose how many cards to start with.

- The challenge rule should be implemented.

Discussion

Based on the needs for describing the UNO behavior, uncovered by doing an object and collaboration analysis, we have introduced data into SPACE. As data is central in all reactive systems, this is an important contribution to the work of developing a highly automatic engineering method. It is now possible to describe the complete behavior of systems using activities, from which state machines and executable code can be generated.

As discussed in this work, one of the strengths of describing behavior using activities lies in a clear, intuitive specification. However, with the introduction of data, the models grows considerable, resulting in large, complex activity specifications that are difficult to understand. A reason for this complexity is the difficulty of letting each collaboration model a clear, separate task, as all collaborations have needs for data owned by a single partition. Thus, the decision of where to place behavior is driven by where data is available, which clearly is not ideal.

A solution to this challenge is using typed collaboration roles for sharing data between building blocks. As data access no longer is restricted to the partition owning the data, this allows for decomposition of the system according to sub-functions, which in our opinion makes the specification clear and intuitive. Typed collaboration roles make the designing of behavior more intuitive as well, which again will speed up the work of designing systems.

Support for data and typed collaboration roles may be implemented in Arctis, so future users of Arctis may take advantage of our work. We think that this is another step towards a highly automatic model-based software design method.

UNO was chosen as an example application because we thought it was challenging due to high complexity, collaborative to a certain degree, and needed support for data. After doing the specification, we think that this was a good choice. It was fairly challenging and complex, and during the

design, several challenges and issues were uncovered and has been described in this work, which hopefully will be of help for later users of Arctis and the SPACE method.

Conclusions and Future Work

8.1 Conclusions

Our motivation was to examine the SPACE approach, and introduce mechanisms and suggest guidelines for designing large, complex systems that relies on data in SPACE. UNO was used as an example application, and the UNO system has been described in detail.

Motivated by the needs uncovered in UNO, elements for representing data in SPACE activities have been added. We have suggested how they may be implemented in Arctis, and in cases where the element syntax differs from the UML standard, we have suggested constraints that should be implemented in Arctis as well.

We have discussed how to best specify large and complex systems, where an important contribution is typed collaboration roles, which provides an elegant way for sharing data between building blocks. We think that this approach should be included in SPACE and implemented in Arctis. We have shown by examples that data sharing is essential for describing such systems in a clear and intuitive way, as describing behavior using data introduce many extra UML elements.

Our work makes it possible to describe the behavior of systems in a more complete and clear way using SPACE. We believe that this will contribute to the work of developing advanced telecommunication systems rapidly.

8.2 Future work

As shown in the specifications in Chapt. 6, some support for data is already implemented in Arctis. However, the elements have constraints and restrictions that should be checked for, both those discussed in this work,

and those stated by the UML standard. This is done by implementing so-called inspectors to support the user when designing services. While some inspectors already exist, more should be provided ensuring a proper application of the concepts to handle data as described in this thesis.

Support for typed collaboration roles should be implemented in Arctis. As discussed in Chapt. 7, typed collaboration roles are essential when modeling large systems where data is needed. Care should be taken to make this highly automatic and intuitive, even for users that have no experience with describing behavior of such systems.

When this functionality is implemented in Arctis, the UNO specification should be finished so code can be generated to make an executable system that can be used as an example application for future users of SPACE and the SPACE tools. This will also validate the model transformation correctness.

Bibliography

- [1] Luca Aceto, Anna Ingolfsdottir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.
- [2] Sebjørn Sæther Birkeland. A Pattern-Based Approach for the Correct Design of Interaction Interfaces. Master's thesis, Norwegian University of Science and Technology, June 2006.
- [3] Conrad Bock. UML 2 Activity and Action Models. *Journal of Object Technology*, 2(4), July-August 2003.
- [4] Conrad Bock. UML 2 Activity and Action Models, Part 2: Actions. *Journal of Object Technology*, 2(5), September-October 2003.
- [5] Conrad Bock. UML 2 Activity and Action Models, Part 3: Control Nodes. *Journal of Object Technology*, 2(6), November-December 2003.
- [6] Conrad Bock. UML 2 Activity and Action Models, Part 4: Object Nodes. *Journal of Object Technology*, 3(1), January-February 2004.
- [7] Conrad Bock. UML 2 Activity and Action Models, Part 5: Partitions. *Journal of Object Technology*, 3(7), July-August 2004.
- [8] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-Oriented Analysis and Design with Applications*. Object Technology Series. Addison-Wesley, 2007.
- [9] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL*. The BCS Practitioner Series. Prentice Hall, 1993.
- [10] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, second edition, 2008. Available at <http://greenteapress.com/semaphores/>.

-
- [11] Eclipse.org. <http://www.eclipse.org>. Last visited on May 1 2008.
- [12] Object Management Group. Unified Modeling Language: Activities. Technical report, August 2007.
- [13] The Object Managemet Group. <http://http://www.omg.org/>. Last visited on April 10 2008.
- [14] Frank A. Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer-Verlag Berlin Heidelberg, September 2007.
- [15] Frank A. Kraemer and Peter Herrmann. Design of Trusted Systems with Reusable Collaboration Models. 2007. Presentation at the Joint iTrust and PST Conferences on Privacy, Trust Management and Security.
- [16] Frank A. Kraemer and Peter Herrmann. Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In *Networking and Electronic Commerce Research Conference 2007 (NAEC 2007)*, October 2007.
- [17] Frank Alexander Kraemer. Arctis and Ramses: Tool Suites for Rapid Service Engineering. In *Proceedings of NIK 2007 (Norsk informatikkonferanse)*, Oslo, Norway. Tapir Akademisk Forlag, November 2007.
- [18] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.
- [19] Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of reactive Services. Preprint submitted to Elsevier, May 2008.
- [20] Mattel. UNO card game. <http://www.mattelgamefinder.com/overview.asp?redirectID=uno>.
- [21] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language*, pages 136–143. Addison-Wesley, second edition, 2005.