Capturing the Essential: Use Case and Service Specification Modelling in UML

Introduction

Software modelling practices need to evolve to keep pace with new paradigms. For example, traditional system-oriented Use Case modelling [1] is at odds with the concept of Service Oriented Architecture (SOA), in which functionality is provided by business services and user access is via portals or mashups, rather than stand-alone applications. A different modelling approach is needed, one that focuses on describing the essential aspects of the user interaction and the business functions provided by the business services.

In this paper we present an approach to using UML (Unified Modelling Language) for modelling Service-Oriented Business Applications (SOBAs), where a front-end, usually web-based, provides access to a set of business services. Rather than modelling use cases within the context of a system boundary, we model them as consumers of business services. We also emphasise the functional contract between the consumer and provider of the service and the need to model the business information known to the service in order to specify the contract.

This approach is a natural extension of the use case specification technique described in a previous paper (http://www.rdfgroup.com/software_development_white_papers), entitled "Making Use Cases Precise".

We make use of the following parts of UML [2]:

- Component diagrams - to describe how use cases and business services are composed to form a SOBA.
- State models – to describe user interaction, including complex user interfaces where the user can move between different use cases.
- Type (class) models – to describe the external business-oriented effect (functional contract) of a business service without giving away details of how the service is implemented.
- Optionally, interaction diagrams – to illustrate how services interact in key scenarios.

Our approach has several benefits:

- Gives a service-oriented view of application functionality, allowing reuse of business services in different use cases/models/applications.
- Focuses on the aspects that are relevant to business stakeholders, that is user navigation and business functionality.
- By modelling use cases as state machines, we can compose use cases from other use cases using the notion of composite state.
- Utilizes a precise subset of the UML and provides precise guidelines for using this subset.
- Provides clear and precise specifications that can be used to develop and test system functionality, while keeping the implementation of that functionality hidden from view.

The high-level picture

To illustrate our approach we use a simple example of a hypothetical set of requirements for a pre-paid payment card issuer. More specifically we will focus on a single business process, "activate card". Pre-paid cards must be activated before first use. Cards can be activated via an automated Interactive

Voice Recognition (IVR) system, or by ringing a Call Centre, or by self-service on the card issuer web site. Batches of cards sold to corporate clients can be activated with a single operation.

Here is a "Service Dependency" diagram showing the actors, use cases and services involved.

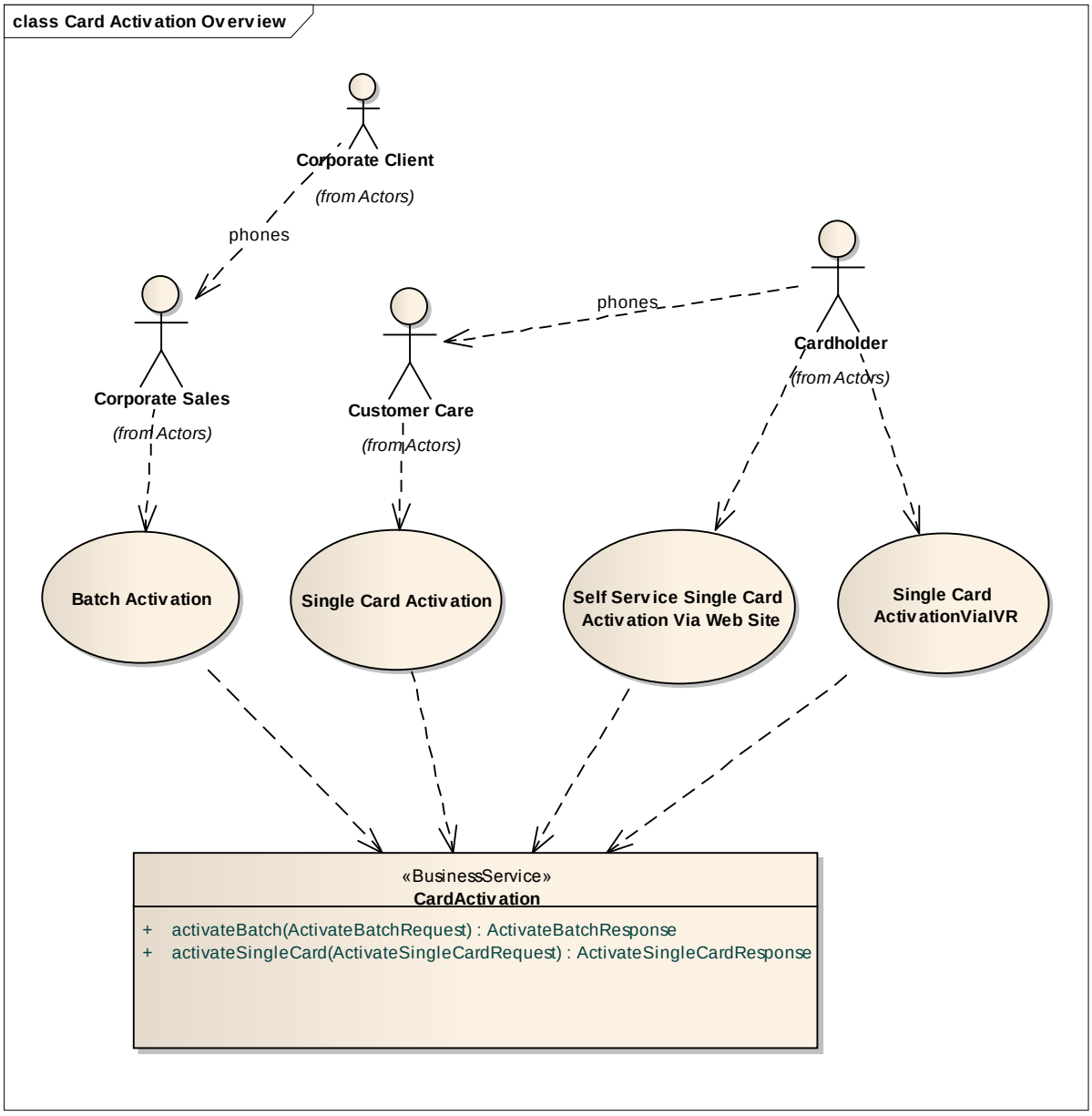

Figure 1: Dependencies between Actors, Use Cases and Business Services

Use Case Models

The key complexity in interactive use cases is UI navigation, so we model them using a dynamic modelling technique, UML state machines, which cater nicely for the event-driven nature of user dialogues. Below is a simplified model for the activation of a single card by a Customer Care advisor (use case Single Card Activation).

Note the composite state at the top of the diagram, stereotyped as <<use case>>. This stands for another use case from where the user can request the activation of a card, and where the user returns to on completion. That use case is itself modelled as a state machine interacting with Business Services, however its behaviour is not relevant in the context of CardActivation.

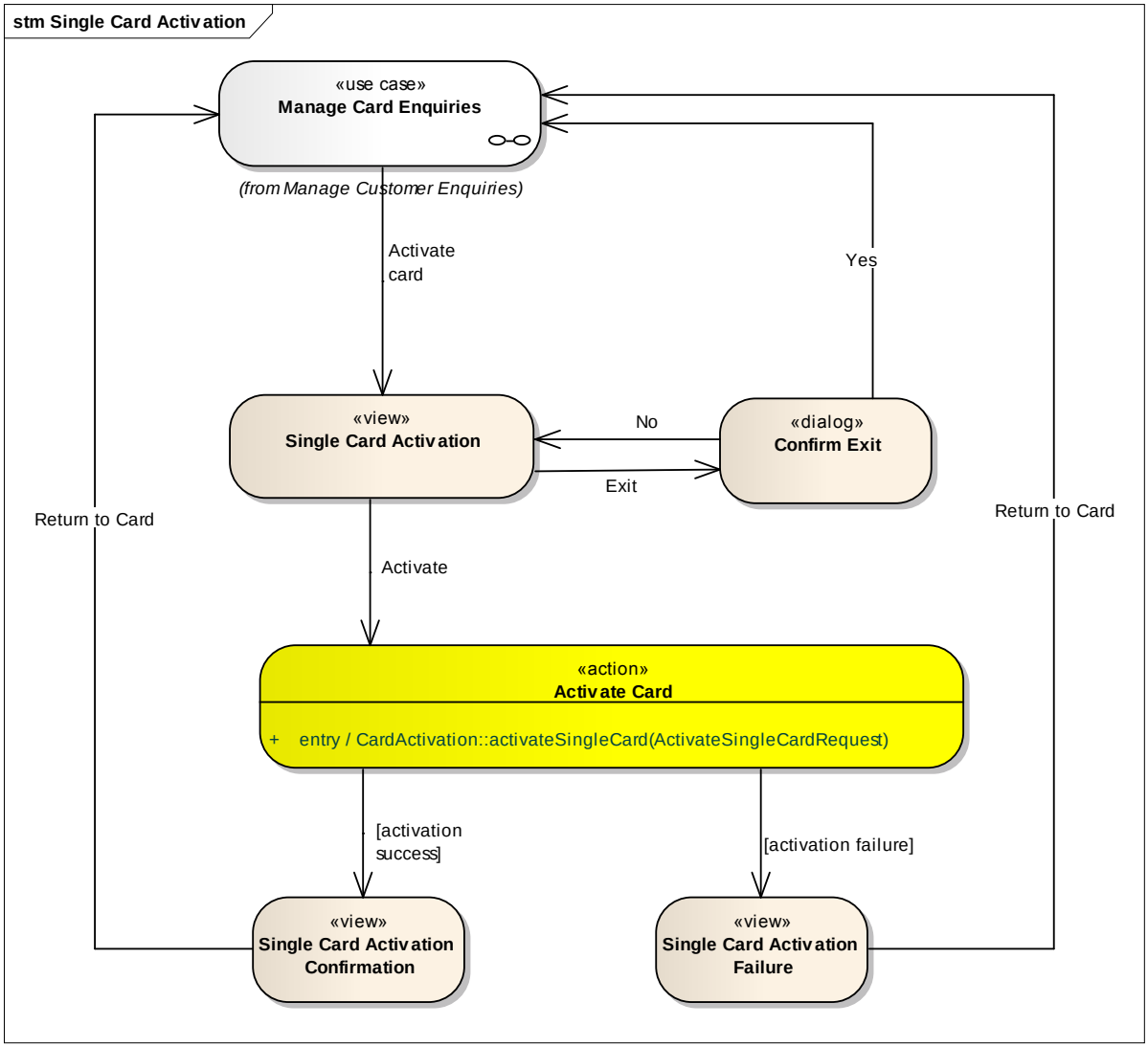

Figure 2: Use Case Modelled as State Machine

States stereotyped as <<view>> represent the display of a web page or, more generally, a view that can be made up of several tabs/pages, to the end user. States stereotyped as <<action>> represent the execution of a system action and are highlighted using a different fill colour. Transitions out of <<view>> pages are triggered by events (i.e. user actions on the UI) whereas transitions out of <<action>> states have no event trigger, since they take place automatically when the action completes.

The points where use cases need to invoke business services are denoted by <<action> states with an "entry" activity that invokes the service, that is, the service is invoked when the state is entered. The two transitions leading out of the <<action>> state in this example have associated mutually exclusive guard conditions, which relate to the result of the service operation call (the call is synchronous). The state is exited as soon as the operation completes.

In our method, individual views are also described by non-UML artefacts such as page mock-ups, so we don't usually feel the need to model in UML the data involved in these views, unless we deal with complex/large use cases where the structure of the information held in the user session deserves a model of its own. In such cases we model UserSessions and ServiceSessions in UML, showing the effect of actions on these objects.

We also produce use case/UI description documents, according to an in-house template that includes a description of the fields and user actions available on each view. Each field is commented by relating it to a type and attribute in a business service type model – see Figure 4.

You may be wondering why we do not follow the practice of keeping use cases separate from user interface specs. The answer is that we consider it unnecessary and even harmful to the clarity of the specifications. Use cases represent interactions or dialogues between a system and an actor, so we create different use cases for different interactions with different actors from different UIs/channels and we include user interface navigation and view details in the use case description. Common functionality is not in the use cases, it is encapsulated in business services.

We have also used UML activity models, rather than state machines, to model use cases. However we have found this to be a clumsy notation for UI event-driven behaviour, being more appropriate to describe use cases with little or no user interaction, such as batch processes or long-lived business processes.

Business Service Specification Models

A Business Service is modelled as a UML interface. The interface contains the signature of each operation provided by the service. The Request and Response messages of each operation are therefore modelled as UML classes. Here is a UML diagram showing the Request and Response message of the single card activation operation:

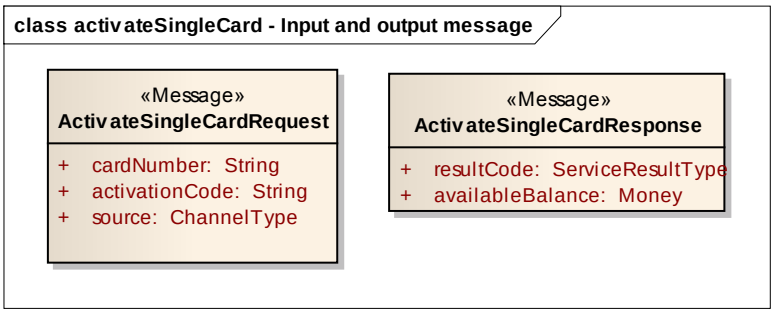

Figure 3: Input and output messages in UML

As well as describing the signature of each operation, we are interested in describing the outcome of the operation. Consider the operation activateSingleCard(). The operation takes a card number, an activation code and a source channel indicator as input data (request message) and responds with a success/failure indicator and the card available balance. To describe the outcome in precise business terms, we need to refer to the domain objects that the operation needs to access/manipulate. In this case, we need to check the status and activation code of the PrepaidCard, do some validation, update the PrepaidCard status and create a Transaction to record the event.

We are very careful in laying down a precise contract between the client of a service operation and the provider. The functional side of the contract is expressed in a declarative fashion using preconditions and postconditions (see box below).

Contract specification with pre- and postconditions
The postcondition of an operation is a boolean expression that is guaranteed to be true when the operation completes, provided that the precondition, itself a boolean expression, is true when the operation is invoked. Thus, from a contractual perspective, the precondition is an ***obligation*** for the consumer of the operation, who has to ensure it is true before the operation is invoked, and a ***benefit*** for the provider, who only has to provide a business outcome when the precondition is met. The postcondition is an ***obligation*** for the provider, who has to ensure the corresponding outcome, and a ***benefit*** for the consumer, who knows that the outcome will have been created.

In order to be useful as a specification, a postcondition must be ***complete***. That is, it must cover the entire business functional outcome of the operation. If things are left unsaid, they will not be implemented or, worse, their implementation will be unpredictable or incorrect.

Pre- and postconditions can be expressed in natural language, but must use a restricted vocabulary, provided by the model of the service.

It is important to emphasise that pre- and postconditions are declarative expressions, not state-changing statements. As such they can only express how things are related to each other, not how they change. A postcondition, for example, can express the value of an output field in terms of the value of input fields, but cannot say what steps are carried out to get there.

A postcondition cannot say anything about the state of the service while the operation is being executed. It can only mention state before and after the execution. Hence writing a postcondition forces you to consider the effect of an operation without considering its internal algorithm, a useful discipline to avoid mixing design considerations with business results.

Note that in order to provide a precise vocabulary to express the contract, we need to model the *state* of the service, as well as the operation input and output messages. By *state* we mean a conceptual representation of what the Service needs to know about relevant business entities in order to satisfy the contract. For example, the CardActivation service needs to know that a set of PrepaidCards exist and must know all the details of these cards, such as their card number, activation PIN, status etc. Whether the service actually has access to a card database, or delegates the persistence to another service or system does not concern us at this level of specification. When we talk about *state* in this context, we only refer to a specification artefact, a vocabulary to write a contract.

To explain this point from a different angle, let us remind ourselves what specifications are for. Service specifications need to include everything that potential consumers need to know about the service, as well as stating everything that a provider of the service needs to know in order to implement the service. In the case of the Card Activation service, a consumer needs to know that when a card is activated, a memory of this event is kept so the card can be used. So it is not sufficient to model the service operation input and output data. It is also essential to model what 'memory' the service needs to maintain. Another piece of memory is the activation code associated with the card. The service needs to know the correct activation code and match it to the one provided by the cardholder.

Figure 4 shows a simplified model for the state of the CardActivation service. The Service can activate cards by virtue of the fact that its 'specification' state is made up of a collection of instances of PrepaidCard, modelled as the *allCards* '1 to many' association between the Service Interface and the PrepaidCard class. The classes of these instances do not have behaviour themselves. Only Services have behaviour. We use these *specification* classes simply as a language for specifying the outcome of a service, by denoting the types of things that the service specification needs to mention. Hence we refer to *types* and *type model* rather than classes and class model. This technique to support operation specifications has been made popular by various object-oriented methodologies in the '90s (e.g. [3], [4], [5]) and is well described in [6].

We ignore at this stage how data is stored or represented - data is modelled in an abstract fashion as instances of 'specification' types. These types can have attributes, associations and other queries and constraints, but cannot exhibit behaviour, otherwise we would be trespassing into the realm of service design. We also deliberately ignore, as much as possible, the individual steps/algorithms by which an operation computes or derives its results. By not attaching behaviour to classes, we avoid the temptation of designing the solution while specifying the business outcome.

As we write the pre- and postcondition in the contract, we check that all the terms we use are supported by the service type model – see Model Check boxes in the precondition and postcondition examples below.

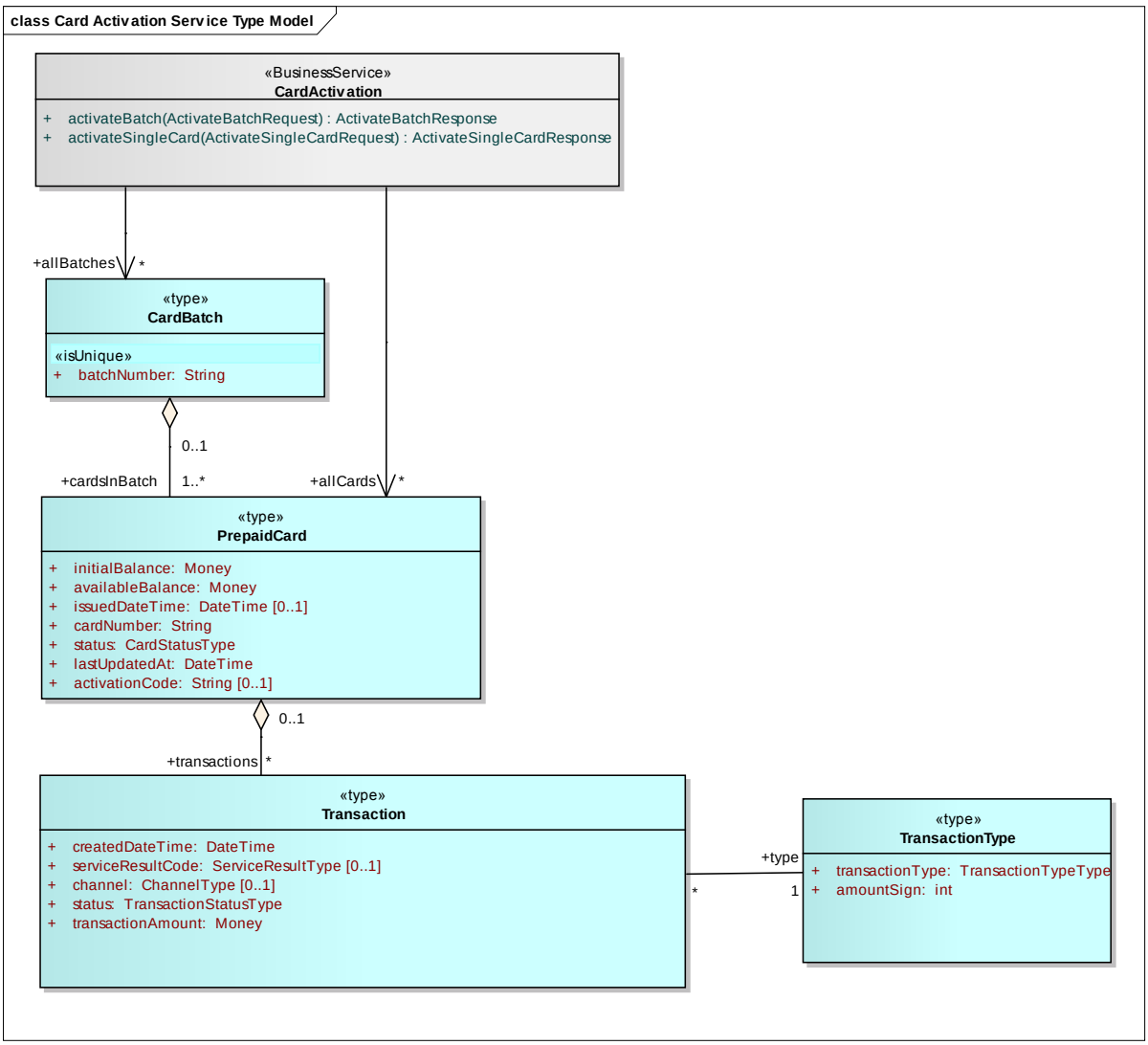

Figure 4: Service Type Model

Example of Service Operation Specification

Signature:

CardActivation::**activateSingleCard**(request: SingleCardActivationRequest) : SingleCardActivationResponse

Precondition:

The card number in the request matches an existing PrepaidCard card number. It is assumed that this operation is called as part of a use case where a valid card number is already available.

Model check: the collection *allCards* must include an instance of PrepaidCard where cardNumber is the same value as request.cardNumber

Postcondition:

Success outcome

Provided the PrepaidCard status is not 'active' or 'blocked' or 'expired', and if the PrepaidCard activation code matches the activation code in the request, then the operation succeeds and has the following outcome:

- The PrepaidCard status is changed to 'active'
- A new Transaction of type "Activate" is created and linked to the PrepaidCard. The status of the Transaction is 'approved', the transactionAmount is zero, the channel is copied from the request, the serviceResultCode is 'success'. The response contains a 'success' serviceResultCode, and the PrepaidCard available balance.

Model check: the status attribute of a PrepaidCard can be one of: INACTIVE, ACTIVE, BLOCKED, EXPIRED (see enum CardStatusType). The activationCode attribute of PrepaidCard must match request.activationCode.
A new Transaction is added to the *transactions* collection linked to the PrepaidCard. This instance is linked to the ACTIVATE instance of TransactionType. The Transaction attributes are all in the model.

Failure outcomes – ….
Activation code incorrect…
Card already active…
Card blocked….

The failure outcomes in the example have been left incomplete for brevity. They can be specified in a similar style and the model checked to ensure that it contains all the properties mentioned in the specification. Things to consider: is a Transaction created if activation fails? How many different service result codes are required? Answers to these questions cannot be left to the service designer/developer. Instead they must be part of the contract, as visible obligations of the service provider, which must be known to the service consumer.

Service Design

As mentioned earlier, the service specification must state all that the service consumer needs to know, as well all that the service provider needs to know. The service type model will indicate to the service designer (the provider) what persisted entities the service needs access to. In our design, this is delegated to an Entity Service 'Cards' (see Figure 5), whose responsibility it is to persist PrepaidCards, CardBatches and associated Transactions. This is a common design pattern: business service implementations are stateless and contain business rules for carrying out business operations. Entity service implementations are stateful and maintain their state across use cases. They do not contain logic specific to business operations. Entity Services may access and manipulate database entities directly or via calls to legacy systems.

Thomas Erl [4] first proposed the categorisation of services as Entity, Task and Process. In Erl's classification, CardActivation would be a Task Service.

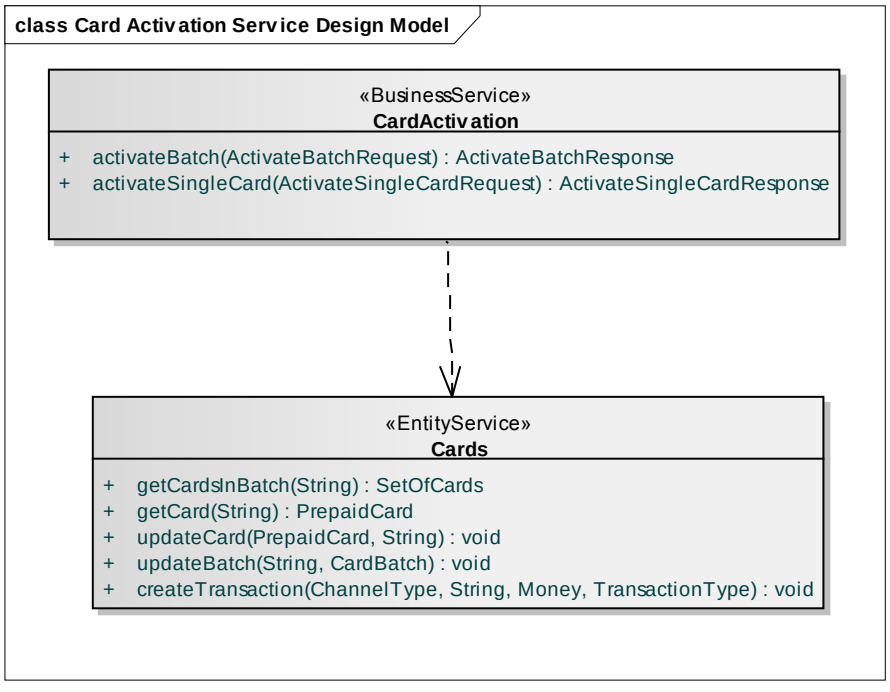

Figure 5: Service Design Dependency

We would typically use an Interaction diagram to illustrate service interactions within an operation. The design of the activateSingleCard() operation is left as an exercise for the reader.

Conclusions

We have shown an effective technique for producing specification models of service-oriented business applications. The technique centres around the use of state machines (use cases) communicating with business services (interfaces). We have explained the need for clear functional contracts for business services and shown how they can be provided by UML type models.

For the sake of brevity, we have ignored the need to model other types of use cases and services, such as batch processes, or how this technique can be integrated with business requirements expressed as business process models. This will be the subject of further papers.

We have also deliberately steered clear of including SoaML [5] elements in our models. SoaML is an OMG backed extension to UML. Modelling with SoaML is not simple, as it contains a host of new and complex concepts which are as yet unfamiliar to most practitioners. We have not yet decided whether and how SoaML can become part of our analysis and specification practices at RDF Group.

References

[1] Ivar Jacobson, Grady Booch and James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley 1999

[2] Grady Booch, Ivar Jacobson and James Rumbaugh, *The Unified Modeling Language Reference Manual*, 2nd Edition, Addison-Wesley 2005

[3] Derek Coleman, P.Arnold, S.Bodoff, C. Dollin, H.Chilchrisr, F. Hayes and P. Jeremaes, *Object-Oriented Development: the Fusion Method*, Prentice-Hall, 1994

[4] Steve Cook and John Daniels, *Designing Object Systems - Object Oriented Modelling with Syntropy*, Prentice-Hall, 1994

[5] Desmond D'Souza and Alan Wills, *Objects, Components and Frameworks with UML: the Catalysis Approach*, Addison-Wesley, 1999

[6] John Cheesman and John Daniels, *UML Components: A Simple Process for Specifying Component-based Software (Component-based Development),* Addison-Wesley, 2000

[7] Thomas Erl, *Service-Oriented Architecture – Concepts, Technology and Design*, Prentice-Hall, 2005

[8] Service Oriented Architecture Modeling Language (SoaML) http://www.omg.org/spec/SoaML/