Rigorous Object-Oriented System Design

Anthony J H Simons, University of Sheffield,
Monique Snoeck, Université Libre de Bruxelles

Contact Information

Dr Anthony J H Simons, Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.

Email: A.Simons@dcs.shef.ac.uk Tel: (+44) 114 222 1838 (direct; +voicemail)
WWW: http://www.dcs.shef.ac.uk/~ajhs/ Fax: (+44) 114 278 0972

Abstract

One of the least systematically explored stages in object-oriented design is the process whereby groups of classes are organised into progressively larger units. The identification and evaluation of subsystems is still largely an intuitive, or expert activity. An effective design process should be capable of layering systems, yielding subsystem units which are cohesive internally and loosely coupled externally. This paper evaluates and compares responsibility-driven and event-driven design techniques for their ability to layer systems. Each of these approaches elevates a different modularising principle: contract minimisation and existence dependency. The expertise is distilled as sets of semi-formal rules, which may be applied to effect system transformations. Important design patterns, such as *Mediator*, *Chain of Responsibility*, *Template Method, Command* and *Composite* are shown to be derived automatically during the transformation process. Finally, a complete formalisation of one design method is presented.

Conference Stream

Research paper, topic areas of design and system transformation.

Subject Areas

Object-oriented design, system layering, subsystem identification, design patterns, responsibility-driven design (RDD), event-driven design (EDD), data dependency, minimisation of contracts, existence dependency

Rigorous Object-Oriented System Design

Anthony J H Simons, University of Sheffield,
Monique Snoeck, Université Libre de Bruxelles

Abstract

One of the least systematically explored stages in object-oriented design is the process whereby groups of classes are organised into progressively larger units. The identification and evaluation of subsystems is still largely an intuitive, or expert activity. An effective design process should be capable of layering systems, yielding subsystem units which are cohesive internally and loosely coupled externally. This paper evaluates and compares responsibility-driven and event-driven design techniques for their ability to layer systems. Each of these approaches elevates a different modularising principle: contract minimisation and existence dependency. The expertise is distilled as sets of semi-formal rules, which may be applied to effect system transformations. Important design patterns, such as *Mediator*, *Chain of Responsibility*, *Template Method, Command* and *Composite* are shown to be derived automatically during the transformation process. Finally, a complete formalisation of one design method is presented.

1. Introduction

The vast majority of object-oriented analysis and design methods agree that the identification of subsystems is an important task. Subsystems are the building blocks that allow a system to be decoupled for various reasons, such as (i) to run on different processors; (ii) to be developed by different teams; (iii) to compile as a separate module; (iv) to facilitate substitution and extension; or (v) simply because the subsystem is itself an important domain abstraction. However, not many object-oriented methods offer any kind of systematic process, in the form of axiomatised steps, for developing subsystems that are optimally partitioned according to some design criteria. Indeed, not many methods offer any systematic criteria for evaluating subsystems. We interpret this vagueness as part of a general problem in object-oriented design: there is a singular lack of attention spent on system-level modelling in object-oriented design, such that object-oriented implementations tend to reflect more the initial analysis model, or "business object" model.

1.1 Naïve "Methods" and the Seamless Transition

Largely, this stems from a naïve[1] "think of an object" approach, and the rush into notations which fix lasting design abstractions during the initial concept identification phase. Coad and Yourdon [CoYo91a; CoYo91b] offer a five-stage development approach, based on directly implementing the first-cut domain abstractions. A subsystem in this approach is called a *subject*, defined retrospectively as the root class of an aggregation or generalisation hierarchy (see also section 3.3). In the rush to engage in "intuitive" object modelling, systems can be produced with cross-linked couplings, complex creation and deletion dependencies, call-back messaging arrangements and so forth. The appeal of new standardised design notations [Rati97, FHG97] will tend to promote these "instant designs", based on a misapprehension that simply using a standard notation equates with following a systematic design process.

Elsewhere, a renewed emphasis on the so-called *seamless transition* [WaNe95, HeEd96] has reinforced a belief that object-orientation promotes the *direct* transfer of domain analysis models into design. (Meyer's original weaker claim [Meye88] was that *domain object concepts* survive across development stages and during system evolution). Instead, we should expect a system to alter, sometimes radically, during the design phase in order to establish a proper modular structure which exhibits loose coupling between modules and encapsulates highly cohesive components inside modules. The seamlessness argument has been challenged from the opposite perspective: that analysis models are unnatural when motivated *a priori* by object-oriented design concerns [HoSi93].

1.2 Intuitive Methods - Layers and Partitions

OMT [RBPE91] and Booch [Booc94] identify subsystems according to an informal appreciation of layers and partitions in a system. A *partition* is part of a system handling a distinct operational concern, such as the database back-end, the domain model, or the user interface. A *layer* is a horizontal substrate which communicates with the layer below, such as the BIOS, the operating system, or the application layer. Booch [Booc94] also has the concept of the *package*, or *module*, fairly coarse-grained units of compilation and distribution; and various medium-scale snapshots of groups of collaborating classes, which we shall call *clusters*.[2] However, the reader is hard-pressed to establish what systematic process should be followed to identify these subsystems; and how to evaluate different possible partitions and

[1] In the sense of primitive, ingenuous, rather than in the sense foolish (necessarily).

[2] Note that UML is now, confusingly, adopting the term "package" to refer to this concept.

boundaries. Where classes are grouped by layer or partition, these categories are usually imposed externally, determined according to intuitive semantic properties, rather than because of any proven internal coupling characteristics.

Where high-quality, optimised modular designs are given, the means of their discovery often remains a mystery. A fine example is in [Booc94], chapter 9, where the architecture of the *Booch Components* [Rati93] is presented. These have been factored along five orthogonal dimensions: collection semantics, element type, storage policy, concurrency control and exception handling. A footnote explains that the given architecture was the fourth revision, begging the question: exactly *how* was this architecture reached? How was it judged better than the previous three? If the solution was developed intuitively, by trial and error, how can the process be made systematic and repeatable [Hump95]?

1.3 More Systematic Approaches - Clustering and Layering

Many of the now-famous design patterns [GHJV95] operate at the level of clusters. Each pattern is a solution to a small-scale design problem, created according to the single principle: "Encapsulate the part that changes". Patterns as diverse as *Abstract Factory* (creational), *Composite* (structural) and *Command* (behavioural) all rely directly on this principle, by reorganising designs around polymorphic plug-in points, which may subsequently be filled by specialised concrete components. This is the kind of quality design activity which is lacking elsewhere in object-oriented methods, although here it is still presented intuitively, on a case-by-case basis.

In the rest of this paper, we describe and evaluate two *systematic* techniques for identifying, reorganising and developing high-quality subsystems. The focus is on uncovering the system modularisation principles used in each approach and making these explicit. Section 2 presents the *Responsibility-Driven* Design method from a more systematic viewpoint, especially the much-neglected system design stage, which elevates contract minimisation as its modularising principle. Section 3 presents an original *Event-Driven* approach developed by the second author and her colleagues, which elevates existence dependency as its guiding principle for modular decomposition. Each method is applied to a different case study which best brings out some of the modularising and transformational principles particular to that method. In our assessment of these two contrasting methods, we use Design Patterns in an unusual way: to evaluate the quality of the system-level designs generated by the methods. Normally, Design Patterns are applied intuitively to particular problem/solution spaces [GHJV95]. Here, we allow the rigorous application of the methods themselves to generate the Patterns which they naturally support. We regard the emergence of Design Patterns as evidence of the quality of

the methods, and the generation of different Design Patterns as an indication of the particular bias of the methods. This linkage has not been made before.

2. Responsibility-Driven Design: Contract Minimisation

Responsibility-Driven Design (RDD) regards objects as behavioural abstractions, characterised at a coarse scale by the "responsibilities" that they bear, which translate 1:M at a finer scale into the services they provide [WiWi89]. Data attributes are assigned later, on a need-to-know basis [Budd91]. The design method [WWW90] operates in two phases: the first generative phase produces new object abstractions using the CRC-card modelling technique [BeCu89]; and the second transformational phase identifies tightly-coupled regions and layers the system using a coupling metric called "minimisation of contracts". RDD is especially good for decentralising control, distributing system behaviour throughout a society of objects [Wirf96].

Most second-hand treatments of RDD [Budd91, Booc94, HeEd96] mistakenly focus only on the informal aspects of the first phase; and then sometimes misunderstand its purpose. It is true that RDD and CRC-card modelling are helpful to promote more active (*viz* behavioural) object concepts, such as *manager* or *controller* abstractions [Booc94]. However, the generative phase of RDD is best applied *ab initio*, not after the prior construction of "object models" (extended Entity-Relationship Diagrams) of some kind. It is important to keep entity boundaries plastic while responsibilities are being elicited and redistributed - prior object modelling tends to fix these boundaries too early. RDD is compatible with other behaviour-centred approaches [Gibs90, RuGo92, Grah95] which use scripts/scenarios/use-cases [JCJO92] to explore system requirements before assigning behaviours to objects. However, very few authors have picked up on the systematic layering offered by the second transformational phase of RDD, which we believe has been unfairly neglected.

2.1 The Rules of RDD

We are chiefly interested in RDD for its power to transform system designs, especially the much-neglected and often misunderstood second phase. However, for completeness' sake, the whole RDD process has been codified in the following 10 rules, shown in Table 1, distilled mainly from [WWW90, Budd91]. We have made certain aspects of the process more explicit (rules 3, 4) and introduced a decision function (rules 5, 6) and a coupling weighting (rule 8) of our own, which we have found useful in the *Discovery* method [Simo98]. In the rules, "entity" is used where the original treatment had "class", because technically classes are identified later, when the basic concepts have solidified.

RDD rule 1: Identify entities on the basis that they fulfil a small (2-7) cohesive set of responsibilities, each a coarse-grained statement of (part of) the purpose of the entity; concepts which bear no responsibility are either simple attributes, or vacuous.

RDD rule 2: Consider how each entity fulfils its responsibilities, establishing collaborations with subcontractor entities, to which it delegates some parts of its responsibilities.

RDD rule 3: Add data attributes, on a need-to-know basis, to those entities bearing a primary responsibility for managing the data; convert passive concepts into attributes.

RDD rule 4: Continue subcontracting until the coarse-grained statements of responsibility reach the fine granularity of single services (methods).

RDD rule 5: If an entity acquires too many responsibilities, and these are cohesive, restate the responsibilities more generally and delegate the detail to new (invented) subcontractors.

RDD rule 6: If an entity acquires too many responsibilities, and these are not cohesive, partition the entity into two or more peer entities according to grouped responsibilities.

RDD rule 7: For each entity, group its services into contracts, one contract per set of services invoked by a distinct set of clients; index the services in each contract.

RDD rule 8: Draw a collaboration graph, linking clients via directed arcs to contracts indexed in each server entity; log the per-service weighted strength of each collaboration.

RDD rule 9: Aggregate tightly-coupled subsystems inside new mediator entities; uncouple the components and have their contracts migrate outwards to the aggregate entity.

RDD rule 10: Generalise groups of entities that offer, or that invoke the same, or similar contracts; merge communication paths to and from the general entity; add dynamic binding.

Table 1: Ten Rules of Responsibility-Driven Design

Rules 1-3 govern the initial conceptualisation of domain entities. Rules 4-6 generate more esoteric entities to decentralise computation; and determine their final granularity by the size constraint and single-purpose requirement. Rules 7-10 govern the systematic restructuring of the system, generating design-level entities needed to reduce system coupling ("minimise

contracts", in [WWW90]). RDD is therefore a *responsibility-driven* approach, which optimises the communication pattern among entities, by transferring the responsibility for handling message requests around the system. The cleverness in RDD lies in its ability to merge communication paths, thereby reducing the degree of static inter-entity coupling required. This is consonant with Parnas' dictum on modularity [Parn72].

The terms used in RDD are often misconstrued, in particular: *responsibility, collaboration* and *contract*. A *responsibility* is not necessarily the same thing as a service or method, but may be (rules 1 and 4); keeping a coarser-grained view affects the operation of rules 5-6. A *collaboration* is best thought of as a connection, or coupling, between a client and a server, rather than the messages sent between them; the coupling view is needed for rule 9 to operate correctly. The transformational stage depends crucially on identifying *contracts*, sets of protocols in a server class's interface *that are used by common sets of clients*. Meyer's use of the term "contract" is different and more specific [Meye88], standing for the reciprocal agreement between a client and a server governing correct invocation and exception-handling *in a single method*. Henderson-Sellers and Edwards distinguish such "method contracts" from "class contracts", understood to be the set of method contracts used by each client [HeEd96]. Each client-server collaboration (*viz* coupling) would then be governed by a single contract. RDD is slightly more subtle than this, grouping services into contracts according to *each distinct set of clients* which invoke them. This means that a given client-server collaboration may eventually be governed by one or more contracts, depending on whether the server has other clients which invoke intersecting groups of services. This distinction affects the operation of rule 10 above.

2.2 Transformations in RDD

A version of the well-known ATM banking machine example is presented at different stages of the RDD process in Figures 1 and 2. In Figure 1, the initial communication pattern has been established between the first-cut domain entities (rule 8 has been applied prematurely, for illustrative purposes). Nouns from the original problem description have been retained only if they can be conceived as bearing some kind of responsibility (rule 1), so concepts like *Money* do not survive, except as attributes of an *Account* entity (rule 3). Early generalisation of *Account* interfaces (rule 10) establishes the inheritance structure as illustrated. In Figure 2, the design process is more advanced, but not yet complete. The *CardReader* has delegated responsibility for verifying the PIN against the *Account* number and secret PIN to a new *Verifier* entity mainly because *CardReader* has no responsibility for retaining the PIN number (rule 3) once it has read the card and PIN number [Budd91]; and because that task of communicating via the datalink is a cohesive part of "read and validate customer a/c number

and PIN" (rule 5) but sufficiently complex (rule 2) for this responsibility to be hived off. Notice how this is an instance of the *Chain of Responsibility* pattern [GHJV95].

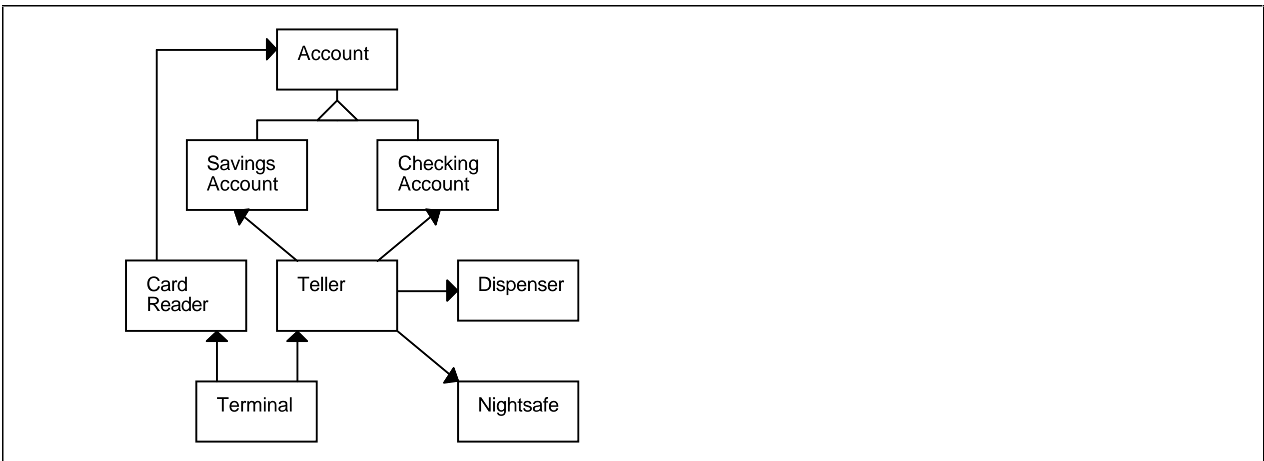


Figure 1: Pre-transformed RDD collaborations

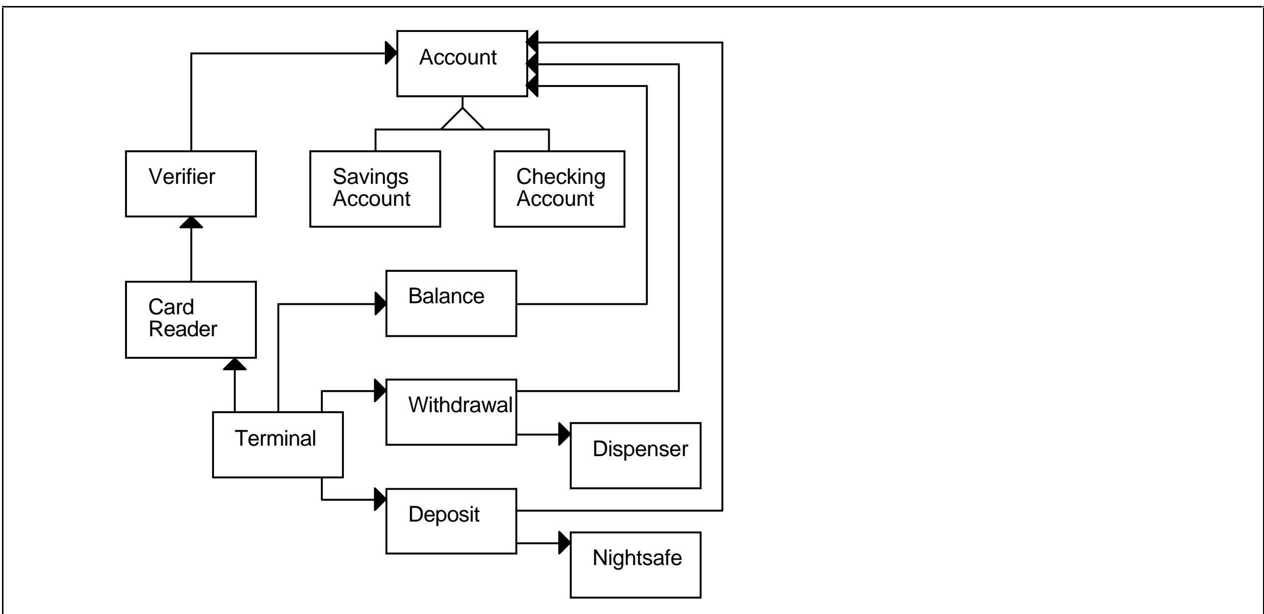


Figure 2: Partially-elaborated RDD collaborations

The part of the system which undergoes greatest change is the *Teller*. Initially, it has the responsibility to "handle banking transactions", but when these are refined into individual services (rule 4), the *Teller* entity must be partitioned. The too-many responsibilities are not cohesive (rule 6) because "deposit money" requires collaborating with the *NightSafe* and *Account*, whereas "withdraw money" requires collaborating with the *Dispenser* and *Account* and lastly, "inspect balance" only requires collaborating with the *Account*. So, three peer "manager" entities (rule 6) are devised to handle each type of transaction. Initially, these are

presumed to be quite different in their behaviour (consider: "deposit", versus "withdraw") and so unrelated.

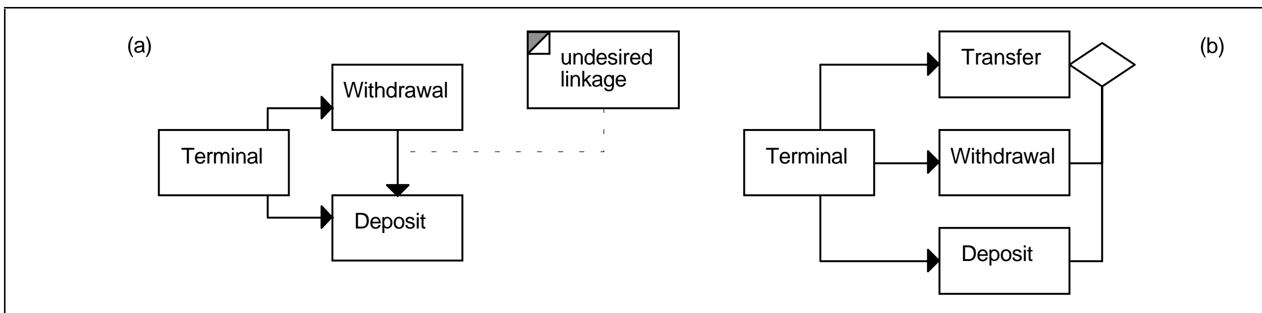


Figure 3: Aggregating over a closed subsystem

If one of the kinds of withdrawal to be supported is really a transfer of funds, this may lead to the undesired cross-coupling shown in Figure 3 (a) which is subsequently resolved, in Figure 3 (b) by aggregating (rule 9) over the *Deposit* and *Withdrawal* entities using a *Transfer* entity. Notice how this is an instance of the *Mediator* pattern [GHJV95]: the *Transfer* entity coordinates the sequence of interactions between the *Deposit* and *Withdrawal* entities, such that these no longer need to maintain references to each other; the contract, renamed "transfer money", migrates outwards to *Transfer*.

The last group of transformations involves considering the contracts of *Account*. We assume this entity has acquired a number of different services (rule 4), invoked directly by the following clients:

Balance: "inspect balance";
Deposit: "inspect balance", "deposit amount", "commit changes";
Withdraw: "inspect balance", "request withdrawal", "withdraw amount", "commit changes";
Transfer: "inspect balance"; *(other requests are indirect, via Withdraw and Deposit)*;
Verifier: "valid a/c?", "valid PIN?", "frozen?".

According to rule 7, *Account* offers five contracts: (1) "inspect balance" is offered to all clients except *Verifier*; (2) "make deposits" is offered to *Deposit*; (3) "make withdrawals" is offered to *Withdrawal*, grouping together the services "request withdrawal" and "withdraw amount"; (4) "commit changes" is offered (directly) to *Deposit* and *Withdraw*; and finally (5) "a/c open?" is offered to *Verifier*, grouping together the services "valid a/c?", "valid PIN?" and "frozen?".

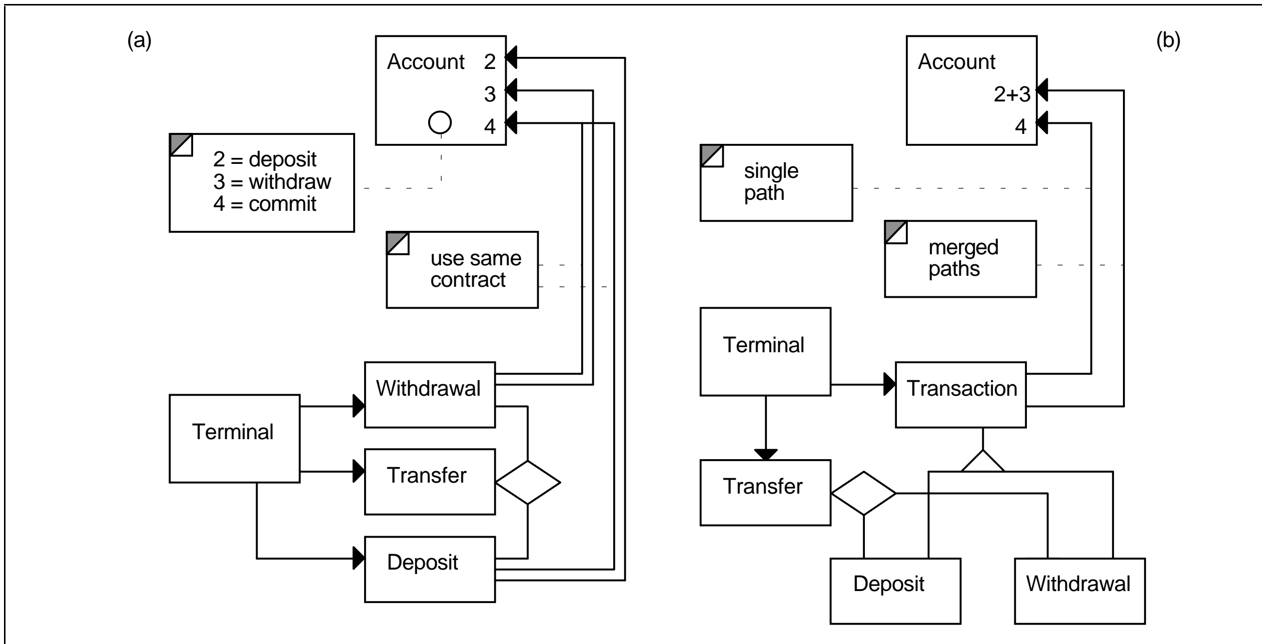


Figure 4: Generalising on commonly-invoked contracts

By drawing the collaboration graph (rule 8) after the proper determination of contracts (rule 7), we see in a more visual way how individual clients enter multiple contracts with *Account* and how different groups of clients use some of the same contracts. A fragment of this system in Figure 4 (a) shows how *Deposit* and *Withdrawal* invoke contract (4) in common, but otherwise invoke apparently separate contracts (2) and (3) each. This is strongly suggestive (rule 10) that some generalisation of *Withdraw* and *Deposit* should handle the communication with *Account*. Calling this new entity a *Transaction* manager, the responsibility for invoking *Account* contracts migrates upwards. Contract (4) is invoked directly by *Transaction*. Contracts (2) and (3) are sufficiently similar, from the perspective of "performing a transaction", that a Template Method may be provided in *Transaction* which is subsequently dynamically bound in *Deposit* and *Withdrawal*, to make the appropriate deposit or withdrawal. The effect of this transformation is to merge the communication paths between *Deposit* and *Account* with those between *Withdrawal* and *Account*. In Figure 4 (b), this is shown by migrating the source of the collaborations upwards, and merging the identical and similar collaboration paths. First, contracts (2) and (3) are merged (via polymorphism); then later it becomes clear that the contracts (2+3) and (4) are only used by the client *Transaction*, so these are merged (rule 7). Notice how this transformation leads systematically to a *Template Method* pattern [GHJV95], in which *Transaction*'s main *handleRequest(Account&)* method invokes a virtual *transact(int)* method stub, followed by a concrete *commit()* method, on an *Account* instance. *Transaction*'s descendants will provide appropriate concrete implementations for *transact(int)*.

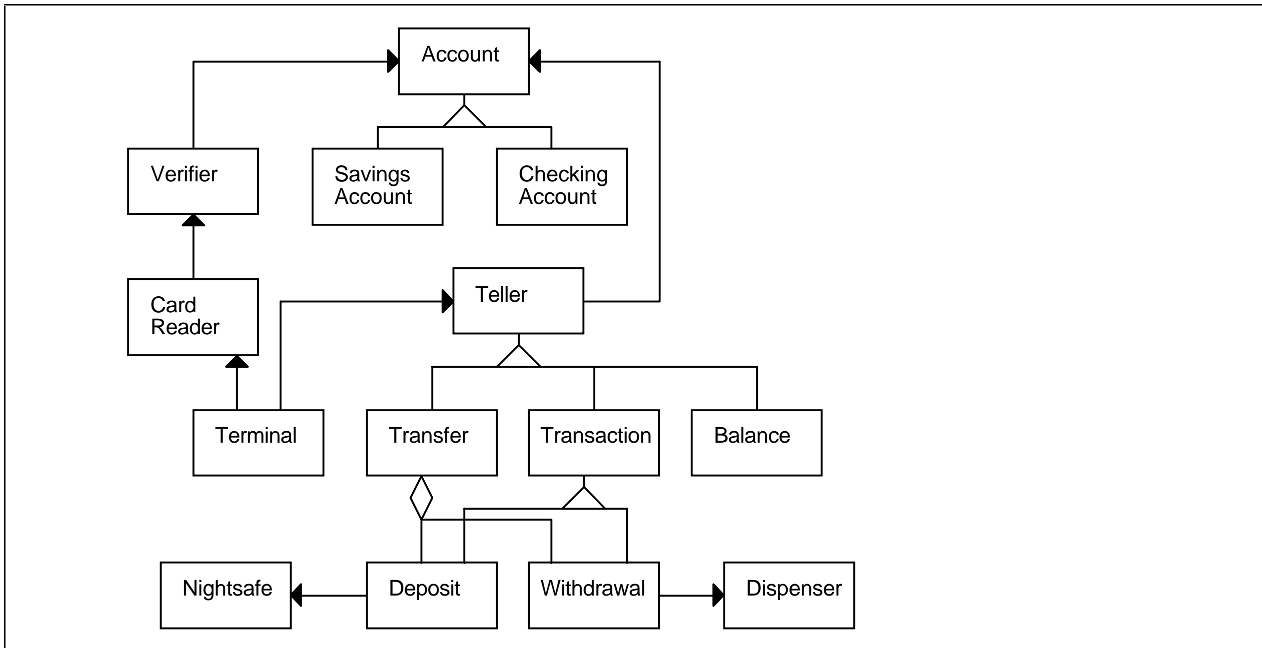


Figure 5: Fully-transformed RDD collaboration graph

A similar process of generalisation (rules 10, 7) results in *Balance*, *Transfer* and *Transaction* being attached under *Teller*, which is reintroduced as an abstract superclass in the final design in Figure 5, having a single contract (1+2+3+4) with *Account*. Notice how this is an instance of the *Command* pattern [GHJV95]: *Teller* encapsulates different kinds of banking requests, fielded by its subclasses. Further merging of *Teller* and *Verifier* is prevented by their too-different external interfaces.

2.3 Subsystems and Coupling in RDD

The kinds of subsystems identified by RDD are equivalent to well-factored modules with minimal inter-module procedure calls. We emphasise that it is the systematic application of rules 7-10 which layers systems properly; and this is the aspect of RDD which is most often neglected. We introduced the per-service weighting measure (rule 7) to let the designer see how many services each collaboration was carrying, in tightly-coupled systems. It provides a rationale for placing subsystem boundaries: you aggregate over the most tightly-coupled parts of the system (with the highest per-service counts) and break the system at weakly-coupled points (with the lowest per-service counts). RDD subsystems are eventually much better motivated than Coad-Yourdon *subjects* [CoYo91a].

RDD supports the bottom-up discovery of *Mediator* patterns, where each *Mediator* is a properly-layered subsystem. The aggregate subsystem *Transfer* obviates the need for its component *Transaction* managers to be coupled directly to each other. Instead, it initiates the communcation between them, handling the transfer of requests and money in a controlled

sequence, possibly recording state information in the process (rule 3). For example, the withdrawal request may be refused, in which case the deposit cannot go ahead. This is ideally handled internally by the *Transfer* manager.

Most methods encourage clustering of classes with similar external interfaces (we assumed this with the grouping of *SavingsAccount* and *CheckingAccount* under *Account*), in other words, their similar behaviour is grouped according to *how they act as servers*. RDD is unique in its ability to cluster classes systematically according to *how they invoke their clients*. We emphasise how clever this is - it is the only approach which can optimise the opposite (usually invisible, encapsulated) end of the collaboration relationship. Through the partitioning of class services into contracts (rule 7) and the construction of fine-grained collaboration graphs (rule 8) RDD supports the bottom-up discovery of *Template Method* and *Command* patterns. In particular, it is the *per-client-set* identification of contracts which allows the designer to see similarities in the global pattern of invocation. Coarser-grained definitions of a collaboration graph [HeEd96, Rati97] do not show patterns of invocation; but only patterns of coupling. This will permit the aggregation activity (rule 9) to proceed, but not the generalisation activity (rule 10).

3 Event-Driven Design: Existence Dependency

The second object-oriented design method we consider is an original one, based on a process algebra [Snoe95, DeSn95] and a conceptual modelling approach [SnDe96]. We call it *Event-driven design* (EDD) because it takes the viewpoint that all computation is made up of events, on which objects must synchronise in order to participate. The notion of event participation is deliberately abstract, avoiding early assignment of responsibility to objects for carrying out actions. A motivating example is where a *Copy* of a library book is taken out on loan by a *Borrower*: which object is responsible for performing this action? The event-driven approach says that neither is, instead both participate in a "borrowing" event. This viewpoint is similar to the view of communication defined in CSP [Hoar85]; whereas traditional message-passing is more like CCS [Miln80].

Entities are identified initially as simple data abstractions and are inserted into an object-event table (OET). Every entity should have one or more associated creation and deletion events bounding the lifetime of its existence; these are logged in the table. Further events, which trigger the main system operations, are also logged against all those entities which participate in each event. An existence dependency graph (EDG) is constructed, in parallel with the OET. This is different from an entity-relationship diagram in that every link is an existence- or lifetime-dependency relationship, between a master and one or more dependent entities. For example, a library may acquire a new *Title* and several *Copies* of that book. The existence of

the *Copies* is directly dependent on that of the *Title*; without the *Title* first being created, no *Copies* can exist; and if the *Title* is ever withdrawn, then all *Copies* must also be removed. The EDG starts as a set of nodes, only some of which may initially depend on each other and so be connected. Eventually, the EDG becomes an acyclic graph (transitive, antisymmetric, non-reflexive) as further nodes and connections are added.

The system elaboration phase extends the OET and EDG by considering groups of entities which must synchronise to participate in events. If they are not already linked by dependency in the EDG, then some new entity must be invented to represent the time-bounded association between the participating entities. This is added to the EDG and appropriate creation and deletion events are logged in the OET for the new entity. An example is the *borrow* event, in which a *Copy* of a book and a *Borrower* participate. This event marks the creation of a *Loan* entity, representing an association between the *Copy* and *Borrower*, which is deleted when a corresponding *return* event signals the return of the book to the library. The *Loan* encapsulates the keys (pointers, IDs) of its participants.

In the system consolidation phase, methods are devised corresponding to each event handled in each entity. The flow of control is from the dependent associative entity to the participating master entities, each of which must have a version of the method to fulfil its part of the jointly-held responsibility for the event. The *borrow* event is the constructor for a *Loan*, which tells a *Borrower* to decrement his/her book allowance and tells the *Copy* to mark itself as unavailable to other library users.

3.1 The Rules of EDD

Onece more, we are interested in EDD as a systematic design process, as described in Table 2.

> EDD rule 1: Entities are data or association concepts, existing for a period of time, bounded by one or more creation and deletion events and involved in possibly many other events.
>
> EDD rule 2: Primary data entities group atomic, non-overlapping sets of attributes, which they are responsible for maintaining.
>
> EDD rule 3: Associative (dependent) entities group the keys of the master entities on which they depend; and may manage further relationship attributes.
>
> EDD rule 4: Events are defined as atomic, non-decomposable actions which (C)reate, (I)nvolve or (D)elete entities; an atomic event must impact on a finite, known number of entities.

EDD rule 5: An object-event table arranges entities (x-axis) against events (y-axis); C, I, D are entered at appropriate intersections; every entity should have at least one C and D; every event should have at least one C, or I, or D.

EDD rule 6: An existence dependency graph connects 1:1 and M:1 simultaneous dependents to their master(s); the lifetime of each dependent is strictly contained within that of its masters.

EDD rule 7: A new association entity is created for each distinct group of entities participating in 2 or more common events; the C, I, D events for this new dependent entity must correspond respectively to: [C or I], I, [D or I] events for its masters.

EDD rule 8: Continue the process until all nodes in the EDG are connected; and all joint participations in events in the OET have been encapsulated in dependent associative entities, or all but one, since two events are needed to bound the lifetime of a dependent entity.

EDD rule 9: All events become methods invoked on the dependent entities, delegating to the participating master entities; dependents handle the intersection of their masters' events.

EDD rule 10: Branches in method-trees are renamed according to the rôles played by each participating entity; similar rôles are clustered; degenerate methods are eliminated.

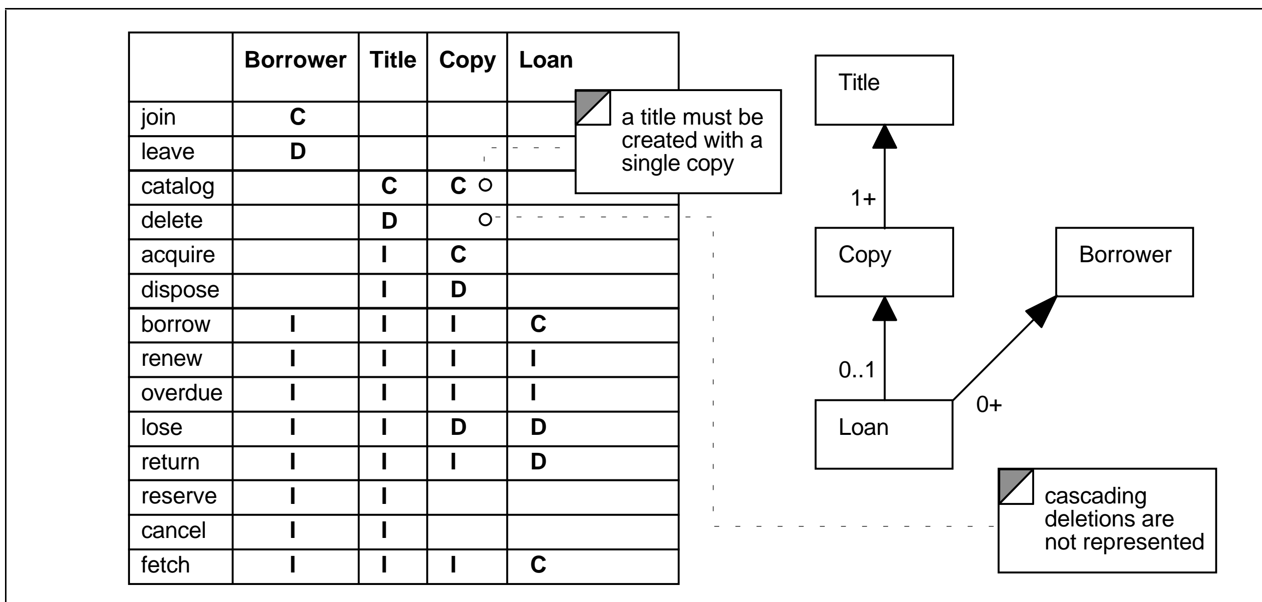
Table 2: Ten Rules of Event-Driven Design

Rules 1-4 govern the identification of entities and events; rules 5-8 govern the elaboration phase which layers the system according to the principle of existence dependency; and rules 9-10 govern the consolidation phase which converts events into chains of methods. There is a pleasing simplicity about the EDG, since all relationships have the same semantics and are already normalised when they are constructed. Also, the mutual influence of the OET and EDG allows the two principles of *event participation* and *existence dependency* to drive the invention of associative entity-abstractions.

3.2 Transformations in EDD

Most of the system layering activity is performed during the elaboration phase, in which new entities are devised according to the principle of existence dependency. Less structural re-design is required, since the event-participation model deliberately leaves the initial message pattern plastic; however, transformations are made to the OET. Figures 6 and 7 illustrate the

lending library system before and after a *Reservation* entity has been added (rules 5-8). In Figure 6, the *Loan* entity manages the common events *borrow, renew, overdue* and *return*, in which *Copy* and *Borrower* participate. Note how the OET contains (I)nvolvement entries for all the master entities (rule 7) impacted by *Loan* events which may have been identified later, such as *renew*. This allows the consequences to propagate to master entities (*eg* the *Borrower* may have certain privileges restored by renewing an overdue book; the *Copy* may have its time-to-inspection reduced); but it is difficult to imagine what impact this might have on *Title*. The consolidation phase (rule 10) eliminates such degenerate methods which were created to respond to events.



	Borrower	Title	Copy	Loan
join	C			
leave	D			
catalog		C	C o	
delete		D	o	
acquire		I	C	
dispose		I	D	
borrow	I	I	I	C
renew	I	I	I	I
overdue	I	I	I	I
lose	I	I	D	D
return	I	I	I	D
reserve	I	I		
cancel	I	I		
fetch	I	I	I	C

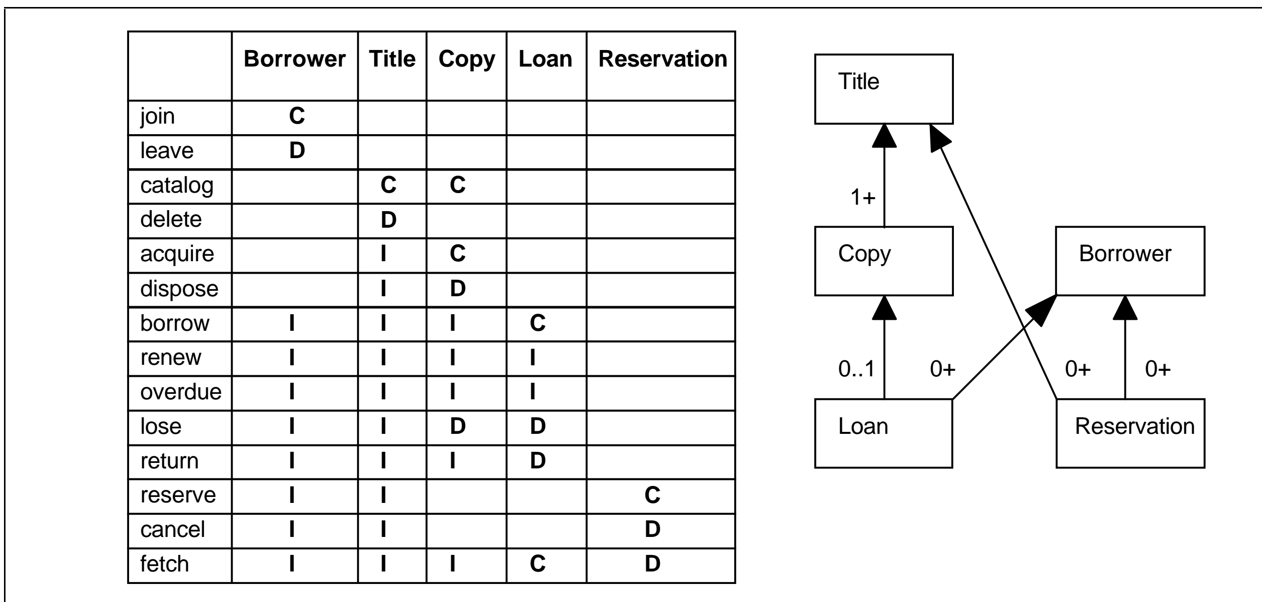
Figure 6: OET and EDG after addition of Loan

The existence of at least two events (*reserve*, *cancel*) which involve two participants (*Title*, *Borrower*) not already covered by the existing *Loan* association motivates the creation of *Reservation* (rule 7). After due consideration (rule 1), the *fetch* event is also identified as a deletion-event for *Reservation*. This is the only event to involve both a *Loan* and *Reservation*, having no duration, so no new associative entity may be created (rules 7, 8).

3.3 Subsystems and Coupling in EDD

Dependent entities in EDD have some of the characteristics of ERM's linker entities (they represent associations; they store foreign keys) but also have characteristics of RDD's *Mediator* patterns (they are devised in response to events; they fulfil a real need to communicate). However, data aggregations may be handled differently in EDD than in other object modelling approaches [RBPE91, CABD94, Rati97, FHG97]. Aggregations

representing existence dependencies are modelled the same way: *eg* the *Lines* of an order are dependent on the *Order*. However, new associative entities must always be devised to relate an assembly to its *non-existence-dependent* parts, such as the components of a PC. Here, an associative entity manages the collaboration between the whole and each part, which is presumed to have a separate existence (it may be exchanged, substituted into other PCs). This tends to promote a distributed pattern of control: the logic of the PC is handled by a society of existence-dependent controllers governing the throughput between the PC and each of its hardware components.



	Borrower	Title	Copy	Loan	Reservation
join	C				
leave	D				
catalog		C	C		
delete		D			
acquire		I	C		
dispose		I	D		
borrow	I	I	I	C	
renew	I	I	I	I	
overdue	I	I	I	I	
lose	I	I	D	D	
return	I	I	I	D	
reserve	I	I			C
cancel	I	I			D
fetch	I	I	I	C	D

Figure 7: OET and EDG after addition of Reservation

EDD optimises the construction order of a system. It is easy to draw entity life history diagrams [AsGo90] for each entity and derive the life-history of the system from this. The logic handling other events during the life of an entity is either pure selection (all events equally likely), or some sequencing of events is required. EDD layers the composition structure similarly to RDD; but it suggests a quite different generalisation structure. By modelling methods on events, one is tempted to give similar names to reciprocal parts of the same interaction. It would be wrong to consider that a *Borrower* and a *Copy* are specialisations of a *Loan* because they respond to a superset of *Loan*'s events (cf rule 9) - fortunately, renaming (rule 10) will tend to isolate the separate rôles in interactions (*viz lend* versus *borrow*). But consider instead that the interfaces of *Loan*, *Borrower* and *Copy* may all be generalised using a *Composite* pattern [GHJV95], which delegates messages from composites to their components. EDD produces large numbers of composite patterns, because of the emphasis on shared participation in events. More important entities will participate in more than one composite pattern, suggesting the use of multiple inheritance from several abstract base

classes. Where one or other master entity is chiefly accountable in handling an event, this is also an instance of *Chain of Responsibility* [GHJV95]. Exploitation of these patterns will tend to reduce the duplication present in the proliferation of co-ordinating abstractions (such as the PC-controllers, above).

4. Towards Systematic Object-Oriented Design

This paper has examined two different approaches to object-oriented design, each of which elevates a different modularising principle: contract minimisation and existence dependency. The design methods were presented in a semi-formal way, partly to show how a good informal method can be improved, clarified and so applied in a rigorous way; and partly to act as an introduction to a fully formal presentation, which we give in an appendix. Being precise about the application of a method is important; however, it is at least as important to consider the quality of the design artifacts produced by the method.

4.1 Coupling, Interdependency and Layering

The kinds of subsystems and layering suggested by each approach are different. EDD promotes unidirectional data coupling in its modelling, but would require the use of an *Observer* pattern (not derived automatically) to handle inverse effects, such as cascading deletions. RDD is more successful in eliminating mutual and closed-loop couplings because of the perspective offered by the collaboration graph. In the same circumstances, where EDD requires an *Observer* pattern, RDD will generate a *Mediator* pattern. RDD is unique in its generalisation strategy, because it merges communication paths at both the source and destination ends. RDD and EDD contrast strongly in the way they generalise - whereas RDD will generate *Command* and *Template Method* patterns, EDD will generate *Composite* and *Chain of Responsibility* patterns.

Both approaches reduce the number of subsystems which interact directly. In some cases, they will suggest the same structures, but for different reasons. A *Purchaser*, *Vendor* and *Product* will end up encapsulated in a *Sale* using both approaches. In RDD, *Sale* will be invented at a late stage to aggregate over the closed ring of collaborations involved in transferring money, goods and ownership; whereas in EDD, *Sale* will necessarily exist, by virtue of the existence dependency rules, but only for the duration of the agreement to purchase until the final transaction is complete.

4.2 Pattern Metrics for System Design

This paper claims that object-oriented design, in particular system modelling, can be made much more systematic. We have emphasised the system modelling aspect, because this has only been treated informally in many other presentations [RBPE91, JCJO92, Booc94, HeEd96]. The distilling of a set of ten rules for each design approach goes some way towards demonstrating our point; but we were further rewarded during the investigation. Initially, we had set out simply to codify and then compare two design approaches. When we applied the semi-formal rules to example system designs, we found again and again that recognisable design patterns emerged [GHJV95]. In particular, we gave examples of *Mediator, Command, Chain of Responsibility*, *Template Method* and *Composite* patterns that were generated automatically by the rules. This reinforces our confidence in the quality of the designs generated by rule and quantifies to some degree the particular bias of the two design methods examined. We note that Design Patterns have not been used in this manner before - as evaluation metrics for methods.

4.3 Design as a Transformational Activity

We have also emphasised that system modelling is a transformational activity, something not truly appreciated by seamless approaches [CoYo91a, CoYo91b, WaNe95, HeEd96]. This is also exemplified in the rulesets. Each ruleset may be partitioned, retrospectively, into three kinds of rule: elicitation-, elaboration- and transformation-rules. Elicitation-rules are almost entirely dependent on the developer for basic information. Elaboration-rules may use internal completeness indicators to request identified pieces of missing information. Transformation-rules apply on parts of the system that are recognised to be in a sub-optimal state and generate new entities and new communication patterns. In conclusion, we feel that system transformation is an important, but neglected area of object-oriented design; the rule-based approach adopted here bears further investigation into whether high-quality designs can eventually be generated automatically from naïve analysis data.

5. Appendix: Formalising RDD

Due to the limitations of space, we are only able to present a formalism for one of the two methods. A newly-published specification for EDD is given in [SnoDe98]. It turns out that RDD is more challenging to specify, since the logic of existence-dependency is already a natural fit with the $\forall x, \exists y$ style of quantification in predicate logic. Many of the analytical procedures in RDD are quite hard to quantify, being based on judgements of meanings and purpose. Clearly, the later transformational rules do apply in specific syntactic circumstances, for which appropriate patterns have to be matched in collaboration diagrams, but the earlier

rules depend on interpretations of statements of responsibility. Nonetheless, it is possible to construct a purely syntactic scheme for RDD, which we develop below.

5.1 Syntactic Elements of RDD

Let N be the set of *nouns* in the domain of discourse, from which labels for entity-concepts and attributes may be drawn. We may now specify certain other label sets of interest, and assert that these sets, when marked with a prime ($'$), also include the undefined element $\perp$:

> $A \subseteq N$, the set of salient *attribute*-nouns;
> $E \subseteq N$, the set of salient *entity*-nouns;
> $S = E \cup A$, the set of *salient* nouns, where $A \cap E = \emptyset$.

Now, let R be the set of *atomic* natural-language statements of responsibility. By atomic, we mean that these statements do not contain "and" or "or". A grammar P for labels describing purpose may then be given, to allow the construction of non-atomic statements of purpose:

> P ::= R | R "and" R | R "or" R | (R)

A candidate *entity* is defined as a map from salient labels S to 5-tuples representing *purpose*, *superclass*, sets of *responsibilities*, *collaborators* and *attributes*.

$$\text{entity} : S \rightarrow (P' \times E' \times \boldsymbol{P}(P) \times \boldsymbol{P}(E) \times \boldsymbol{P}(A)), \qquad \text{where } \boldsymbol{P} \text{ is } \textit{powerset}.$$

This function is defined for $s \in S$ according to the conditions:

$$\forall s \in S,\ \forall p \in P,\ \forall e \in E,\ \forall r \in \boldsymbol{P}(P),\ \forall c \in \boldsymbol{P}(E),\ \forall a \in \boldsymbol{P}(A)\ .$$
$$(\text{entity}(s) = (p \times e \times r \times c \times a) \wedge p \neq \perp \wedge 2 \leq \mathbf{card}(r) \iff s \in E) \wedge$$
$$(\text{entity}(s) = (\perp \times \perp \times \emptyset \times \emptyset \times \emptyset) \iff s \in A)$$

This defines the membership of A, E and S; and implicitly the syntactic meaning of *salience*, by ruling out other possible *entity* structures (RDD rules 1 and 3, above). Hereafter, we are chiefly interested in the subset domain $\forall s \in E$, and define the accessors:

purpose : $E \rightarrow P$	$\forall e \in E$. purpose(e) = entity(e)[1];
superclass : $E \rightarrow E'$	$\forall e \in E$. superclass(e) = entity(e)[2];
responsibilities : $E \rightarrow \boldsymbol{P}(P)$	$\forall e \in E$. responsibilities(e) = entity(e)[3];
collaborators : $E \rightarrow \boldsymbol{P}(E)$	$\forall e \in E$. collaborators(e) = entity(e)[4];
attributes : $E \rightarrow \boldsymbol{P}(A)$	$\forall e \in E$. attributes(e) = entity(e)[5];

Now, let F be the set of atomic services. Each service $f \in F$ is a function defined in the context of an entity. Each f typically acts on many inputs and yields a single output in S, but may also

read or modify an entity's attributes and, by invoking other services, may affect the state of other referenced entities. So, we formalise *f* as a map from 3-tuple to 3-tuple:

$$f : (\boldsymbol{P}(S) \times \boldsymbol{P}(A) \times \boldsymbol{P}(E)) \rightarrow (S \times \boldsymbol{P}(A) \times \boldsymbol{P}(E)), \qquad \text{for each } f \in F;$$

Let us now interpret the a meaning of a *purpose* as a map from labels to sets of services and define that, for any $p \in P$, the evaluation *refine(p)* will yield the corresponding services:

$$\text{refine} : P \rightarrow \boldsymbol{P}(F) \qquad \forall\, e \in E, \exists s \in \boldsymbol{P}(F) \,.\, \text{refine}(e) = s.$$

This allows us to relate syntactically the "statement of purpose" and the individual "statements of responsibility" in an entity. The responsibilities map to sets which exhaustively partition the set mapped from the purpose; and each responsibility maps to a nonempty set:

$$\forall e \in E, \forall p \in \text{responsibilities}(e) \,.\, \text{refine}(\text{purpose}(e)) = \bigcup \text{refine}(p_i), \qquad i = 1..n;$$
$$\forall e \in E, \; \forall p,q \in \text{responsibilities}(e) \,.\, p \neq q \;\Rightarrow\; \text{refine}(p) \cap \text{refine}(q) = \emptyset;$$
$$\forall e \in E, \; \forall p \in \text{responsibilities}(e) \,.\, \text{refine}(p) \neq \emptyset;$$

We introduce various shorthand functions below to facilitate access to services and to their arguments, results and other entity-scoped variables and references:

$$\text{services} : E \rightarrow \boldsymbol{P}(F) \qquad \forall e \in E \,.\, \text{services}(e) = \text{refine}(\text{purpose}(e));$$
$$\text{arguments} : F \rightarrow \boldsymbol{P}(S) \qquad \forall f \in F \,.\, \text{arguments}(f) = \mathbf{dom}(f)[1];$$
$$\text{result} : F \rightarrow S \qquad \forall f \in F \,.\, \text{result}(f) = \mathbf{ran}(f)[1];$$
$$\text{variables} : F \rightarrow \boldsymbol{P}(A) \qquad \forall f \in F \,.\, \text{variables}(f) = \mathbf{dom}(f)[2] \cup \mathbf{ran}(f)[2];$$
$$\text{references} : F \rightarrow \boldsymbol{P}(E) \qquad \forall f \in F \,.\, \text{references}(f) = \mathbf{dom}(f)[3] \cup \mathbf{ran}(f)[3].$$

We may also characterise the "need to know" assignment of attributes in RDD rule 3 by specifying that attributes may only be used or modified by services owned by the same entity, and must be read/written at least once:

$$\forall e \in E, \forall a \in \text{attributes}(e), \exists f, g \in \text{services}(e) \,.\, a \in \mathbf{dom}(f)[2] \wedge a \in \mathbf{ran}(g)[2];$$
$$\forall d,e \in E, \forall a \in \text{attributes}(e), \forall f \in \text{services}(e) \,.\, d \neq e \;\Rightarrow\; a \notin \text{variables}(f).$$

5.2 Elaboration Rules in RDD

It is fairly easy to formalise syntactic conditions for the RDD elaborational rules 2, 5 and 6. The basic constraint is that when the size of the responsibility-set exceeds seven, the entity must be split into two or more equivalent entities. The decision whether to subcontract (rule 5) or split into peers (rule 6) is encoded as a predicate on the atomicity of purpose.

$$\forall e \in E, \exists c,d,e' \in E \;.\; \mathbf{card}(\text{responsibilities}(e)) > 7 \;\wedge\; \text{purpose}(e) \in R \;\Rightarrow$$
$$\mathbf{card}(\text{responsibilities}(e')) < \mathbf{card}(\text{responsibilities}(e)) \;\wedge$$
$$\text{services}(e) \supseteq (\text{services}(c) \cup \text{services}(d)) \;\wedge$$
$$\text{services}(c) \cap \text{services}(d) = \varnothing \;\wedge$$
$$c \in \text{collaborators}(e') \;\wedge\; d \in \text{collaborators}(e');$$

$$\forall e \in E, \exists c,d \in E \;.\; \mathbf{card}(\text{responsibilities}(e)) > 7 \;\wedge\; \text{purpose}(e) \notin R \Rightarrow$$
$$\mathbf{card}(\text{responsibilities}(e)) = \mathbf{card}(\text{responsibilities}(c)) + \mathbf{card}(\text{responsibilities}(d)) \;\wedge$$
$$\text{services}(e) = (\text{services}(c) \cup \text{services}(d)) \;\wedge$$
$$\text{services}(c) \cap \text{services}(d) = \varnothing.$$

The first formula above captures RDD rules 2 and 5, in which entity e is replaced by e', which subcontracts to c and d. The second captures RDD rules 2 and 6, in which entity e is replaced by peers c and d. The re-statement of responsibilities in more or less general terms may be captured in a formula asserting the equivalence of the set of services to which they (the individual responsibilities; and hence the corresponding purposes) map. This covers the outstanding aspects of RDD rules 2, 5 and 6:

$$\forall e \in E, \exists e',e'' \in E \;.\; \text{services}(e') = \text{services}(e) = \text{services}(e'') \;\wedge$$
$$\mathbf{card}(\text{responsibilities}(e')) < \mathbf{card}(\text{responsibilities}(e)) < \mathbf{card}(\text{responsibilities}(e'')).$$

To this, we add a predicate for determining when an entity is in canonical form and so provide for testing when the RDD elaboration process is complete (rules 1, 4):

$$\text{canonical} : E \rightarrow \text{Boolean}$$
$$\forall e \in E, \forall p \in \text{responsibilities}(e) \;.\; \text{canonical}(e) \;\Leftrightarrow\; \text{purpose}(e) \in R \;\wedge\; p \in R \;\wedge$$
$$2 \leq \mathbf{card}(\text{responsibilities}(e)) \leq 7 \;\wedge\; \mathbf{card}(\text{refine}(p)) = 1.$$

5.3 Transformation Rules in RDD

The construction of a collaboration graph depends on formalising the notion of client-server dependency. Each service $f \in F$ is defined in terms, not only of its simple inputs and outputs, but also in terms of the attributes and references it uses. Eventually, we are interested in capturing all client-server connections, whether permanent or temporary, so define a constraint on *collaborators* accordingly:

$$\forall c,e \in E, \exists f \in \text{services}(e) \;.$$
$$c \in \text{collaborators}(e) \;\Leftrightarrow\; c \in \text{arguments}(f) \vee c \in \text{references}(f).$$

It is possible to construe all collaborators as *permanent* initially and progressively transfer some of these to *temporary* arguments as the permanent entity coupling is reduced, during system transformation:

$$\forall c,e \in E, \exists e' \in E \,.\, \forall f \in services(e), \exists f' \in services(e') \,.\, c \in references(f) \Rightarrow c \in arguments(f') \wedge c \notin references(f').$$

The notion of *contracts* depends first on identifying all the clients of an entity:

$$clients : E \rightarrow \boldsymbol{P}(E) \qquad \forall e \in E \,.\, clients(e) = \{ c \in E \mid e \in collaborators(c) \};$$

Let there be a map from client-server pairs to the set of services invoked by the client in the server. We may relate this primitive function to proper notions of collaboration:

$$invokes : E \times E \rightarrow \boldsymbol{P}(F)$$
$$\forall c,e \in E, \exists s \in \boldsymbol{P}(F) \,.\, c \neq e \Rightarrow invokes(c,e) = s \wedge (s \neq \emptyset \Leftrightarrow c \in clients(e) \wedge e \in collaborators(c));$$

The contracts of an entity are defined as the disjoint partitioning of its services, such that each partition is invoked by a different subset of its clients. For this, we must use indexing to assert that service partitions are *disjoint* and that the corresponding subsets of clients are *distinct*:

$$contracts : E \rightarrow \boldsymbol{P}(\boldsymbol{P}(F))$$
$$\forall e \in E, \exists p_1..p_n \subseteq services(e) \,.\, p_i \in contracts(e) \Leftrightarrow \exists s_1..s_n \subseteq clients(e), \forall c \in s_i \,.\, invokes(c, e) = p_i \wedge s_i \neq s_k \wedge p_i \cap p_k = \emptyset \wedge \bigcup p_i = services(e), \qquad i, k = 1..n,\ i \neq k.$$

Minimisation of contracts may be judged later according to **card***(contracts(e))*, for each $e \in E$.

The basic logical machinery for transforming a subsystem by *aggregation* has already been described above, in the formulae for subcontracting and for transferring a permanent collaborator to a temporary argument. A rule for recognising mutually-joined entities and suggesting that a mediator-transformation is appropriate is encoded as:

$$\forall c,e \in E, \exists m,c',e' \in E, \forall f \in invokes(c, e), \forall g \in invokes(e, c), \exists f',g',f'',g'' \in F \,.$$
$$(e \in collaborators(c) \wedge e \in references(f) \wedge c \in collaborators(e) \wedge c \in references(g)) \Rightarrow$$
$$(c' \in collaborators(m) \wedge e' \in collaborators(m) \wedge f'' \in invokes(m, c') \wedge g'' \in invokes(m, e') \wedge$$
$$((f' \in invokes(c', e') \wedge e' \in arguments(f')) \vee e' \notin collaborators(c')) \wedge$$
$$((g' \in invokes(e', c') \wedge c' \in arguments(g')) \vee c' \notin collaborators(e')))$$

Transforming a system by generalisation depends on being able to introduce a *parent* for pairs of clients which share some contracts with a server. We can set the threshold at which generalisation is triggered by saying that the clients must share at least half of their contracts.

$$\forall c,d,e \in E, \exists p,c',d' \in E,$$
$$\quad \forall f \in invokes(c, e), \forall g \in invokes(d, e), \exists f',g',h \in F \ .$$
$$(e \in collaborators(c) \wedge e \in collaborators(d) \wedge$$
$$\quad \mathbf{card}(contracts(c) \cap contracts(d)) \geq \mathbf{card}(contracts(c) \cup contracts(d)) / 2) \Rightarrow$$
$$(p = superclass(c) \wedge p = superclass(d) \wedge e \in collaborators(p) \wedge$$
$$\quad ((f' \neq g' \wedge h \in invokes(p, e) \wedge f' \in invokes(c', e) \wedge g' \in invokes(d', e)) \vee$$
$$\quad (f' = g' = h \wedge h \in invokes(p, e) \wedge$$
$$\quad\quad e \notin collaborators(c') \wedge (e \notin collaborators(d'))))$$

In all our transformation rules, we have simply asserted that an alternative system structure exists. To be able to trace the evolution of the system, we should have to introduce further ∃-quantified state-variables, which considerably complicates the model.

References

[AsGo90] C Ashworth and M Goodland, SSADM: A Practical Approach (1990), McGraw-Hill.

[BeCu89] K Beck and W Cunningham, "A laboratory for teaching object-oriented thinking", Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl., pub. Sigplan Notices, 24(10), 1-6.

[Booc94] G Booch, Object-Oriented Analysis and Design with Applications, 2nd edn. (1994), Benjamin-Cummings.

[Budd91] T Budd, Introduction to Object-Oriented Programming (1991), Addison-Wesley.

[CABD94] D Coleman, P Arnold, S Bodoff, C Dollin, H Gilchrist, F Heyes and P Jeremaes, Object-Oriented Development: The Fusion Method (1994), Prentice Hall.

[CoYo91a] P Coad and E Yourdon, Object-Oriented Analysis (1991), Yourdon Press.

[CoYo91b] P Coad and E Yourdon, Object-Oriented Design (1991), Yourdon Press.

[DeSn95] G Dedene and M Snoeck, "Formal deadlock elimination in an object-oriented conceptual schema", Data and Knowledge Engineering, 15 (1995), 1-30.

[FHG97] D Firesmith, B Henderson-Sellers and I Graham, OPEN Modelling Language (OML) Reference Manual, March (1997), SIGS Books.

[GHJV95] E Gamma, R Helm, R Johnson and J Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (1995), Addison-Wesley.

[Gibs90] E A Gibson, "Objects born and bred", BYTE magazine, 15(10) (1990) 245-254.

[Grah95] I M Graham, *Migrating to Object Technology*, Addison-Wesley (1995).

[HeEd96] B Henderson-Sellers and J Edwards, Book Two of Object-Oriented Knowledge: The Working Object (1996), Prentice Hall.

[Hoar85] C A R Hoare, Communicating Sequential Processes (1985), Prentice-Hall.

[HoSi93] G M Høydalsvik and G Sindre, "On the purpose of object-oriented analysis", Proc. 8th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl., pub. Sigplan Notices, 28(10) (1993), 240-255.

[Hump95] W S Humphrey, A Discipline for Software Engineering (1995), Addison-Wesley.

[JCJO92] I Jacobson, M Christerson, P Jonsson and G Övergaard, Object-Oriented Software Engineering: a Use-Case Driven Approach (1992), Addison-Wesley.

[Meye88] B Meyer, Object-Oriented Software Construction (1988), 2nd. edn. rev. and enl. (1997), Prentice-Hall.

[Miln80] R Milner, A Calculus of Communicating Systems, LNCS (1980), Springer.

[Parn72] D Parnas, "On the criteria to be used in decomposing systems into modules", Comm. ACM, 15(12) (1972), 1053-1058; reprinted in: Classics in Software Engineering, ed. E Yourdon (1979), Yourdon Press.

[Rati93] Rational, C++ Booch Components Class Catalog, version 2.3 (1993), Rational.

[Rati97] Rational, UML 1.1 Reference Manual, September (1997), Rational; also available through: http://www.rational.com/uml/ .

[RBPE91] J Rumbaugh, M Blaha, W Premerlani, F Eddy and W Lorensen, Object-Oriented Modeling and Design (1991), Prentice-Hall.

[RuGo92] K Rubin and A Goldberg, "Object-behaviour analysis", Comm. ACM, 35(9) (1992).

[ShMe88] S Shlaer and S Mellor, Object-Oriented Analysis: Modelling the World in Data (1988), Yourdon Press.

[Simo98] A J H Simons, "Object Discovery: a systematic approach to developing object-oriented systems", *Workshop 8, BCS Object Technology*, Oxford (1998); see also: http://www.dcs.shef.ac.uk/~ajhs/discovery.

[SnDe96] M Snoeck and G Dedene, "Generalisation/specialisation and rôle in object-oriented conceptual modelling", Data and Knowledge Engineering, 19(2) (1996).

[SnDe98] M Snoeck and G Dedene, "Existence dependency: the key to semantic integrity between structural and behavioural aspects of object types", *IEEE Trans. Software Engineering, 24(4)* (1998).

[Snoe95] M Snoeck, "On a process algebra approach to the construction and analysis of MERODE-based conceptual models", PhD dissertation, Katholieke Universität Leuven (1995), 415pp.

[StLe94] A Stepanov and M Lee , The Standard Template Library, Technical Report, June (1994) Hewlett Packard.

[Strou91] B Stroustrup, The C++ Programming Language, 2nd edn, Addison-Wesley.

[WaNe95] K Waldén and J M Nerson, Seamless Object-Oriented Architecture (1995), Prentice-Hall.

[Wirf96] R Wirfs-Brock, Responsibility-Driven Design Tutorial Notes (1996).

[WiWi89] R Wirfs-Brock and L Wiener, "Responsibility-driven design: a responsibility-driven approach", Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl., pub. Sigplan Notices, 24(10) (1989), 71-76.

[WWW90] R Wirfs-Brock, B Wilkerson and L Wiener, Designing Object-Oriented Software (1990), Prentice Hall.