Design Patterns as Litmus Paper to Test the Strength of Object-Oriented Methods

Anthony J H Simons[1], Monique Snoeck[2] and Kitty S Y Hung[1]

[1] Department of Computer Science, University of Sheffield, Sheffield, United Kingdom
[2] Département d'Informatique, Université Libre de Bruxelles, Brussels, Belgium

Abstract

This paper shows how Design Patterns may be used to reveal properties of object-oriented development methods. The responsibility-driven and event-driven design methods are contrasted in the way they transform and layer systems. Each method elevates a different modularising principle: contract minimisation and existence dependency. Different design patterns, such as *Mediator*, *Chain of Responsibility*, *Template Method*, *Command* and *Composite* emerge for each method, illustrating the particular bias and the different design decisions each makes.

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

1 Introduction

The vast majority of object-oriented analysis and design methods are in agreement 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, some methods, such as Booch [3], p229, emphasise the continual need for creativity and intuition, believing that it is impossible to codify the design process. Other methods, such as OMT [20] choose to split systems up according to subjective criteria, such as *layers* (code substrates, virtual machines) and *partitions* (intuitively-determined subsystem modules). Instead, it would be better if subsystems were selected

according to measurable internal criteria, such as the degree of inter-module coupling [18], which corresponds to the number of inter-object references needed for message sending in the object-oriented model. A good system design method should minimise inter-object coupling across subsystem boundaries and thereby also foster *subsystem reuse* in new contexts.

More recently, *Design Patterns* have emerged as the "distilled products" of high-quality object-oriented designs [10]. 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. The application of Design Patterns is normally a *system design* activity, in the sense we are seeking above. But, Design Patterns are again applied *intuitively* to particular problem/solution spaces [10] by expert developers who recognise these situations. In the rest of this paper, we use Design Patterns in a quite different way, as the "litmus paper" to judge the quality or strength of particular object-oriented development methods.

Because we were interested in comparing the kinds of *systematic guidance* provided by object-oriented design methods to non-expert developers, we needed to select methods which were obviously directive in their modelling approach. We considered that Booch [3] and OOSE [15] rely over-much on expert developer intuition in the identification of object concepts and subsystems. OMT [20], Coad-Yourdon [6, 7] and Shlaer-Mellor [22] all have a data-driven foundation that is amenable to systematic entity-relationship modelling (ERM), which elevates *data dependency* as its system modularisation principle. The deliverable of ERM is a set of normalised data files (equiv. 3NF) which says nothing about the procedural structure of the system interrogating the data. We did eventually find two methods which satisfied our criteria for providing proper direction for object-oriented design. Section 2 reformulates the *Responsibility-Driven* Design method [28, 29, 27] from a 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 adapted from the work of the second author and her colleagues [24, 25], which elevates *existence dependency* as its guiding principle for modular decomposition. Both approaches are evaluated for their potential to identify properly-layered subsystems with loose external coupling. In our assessment of these two contrasting methods, we use Design Patterns in an unusual way: as indicators of the design decisions taken by the methods. We allow the *systematic application of the methods themselves* to generate the Patterns which they naturally tend to promote. 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 each method. This connection 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 [28]. Data attributes are assigned later, on a need-to-know basis [4]. The design method [29] operates in two phases: the first generative phase produces new object abstractions using the CRC-card modelling technique [2]; 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 [27].

Most second-hand treatments of RDD [4, 3, 13] 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 [3]. However, the generative phase of RDD is best applied *ab initio*, not after the prior construction of object models. 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 [11, 21, 12] which use scripts/scenarios/use-cases [15] 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 (an arbitrary number, but sufficient for our purposes), shown in table 1. The rules are an original semiformal characterisation of published informal descriptions of the RDD method [29, 4, 27]. We have made certain aspects of the RDD process more explicit (rules 1, 3), introduced a halting-condition (rule 4) and a novel decision function (rules 5, 6) for determining *how* an entity should be split when it is judged too large (by rule 1). A novel coupling weighting (rule 8), which we have found useful in the *Discovery* method [23] helps to show the degree of functional dependency expressed in a static client-server coupling. 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 [29]).

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 contracts.

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

The terms used in RDD are sometimes misunderstood, in particular: *responsibility, collaboration* and *contract.*

- A *responsibility* is not necessarily the same thing as a service, but may be (rules 1 and 4); it is a statement of purpose, not the name of a method; keeping this 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 [29], rather than the messages sent between them [19]; the coupling view is needed for rule 9 to operate correctly.

- The transformational stage depends crucially on identifying *contracts*, sets of services in a class interface *that are used by common sets of clients* [29]. Meyer's use of the term "contract" is different [16], 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 Meyer's "method contracts" from Wirfs-Brock's "class contracts", understood to be the set of method contracts used by each client [13]. Each client-server collaboration 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. In summary, RDD is 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, so reducing the degree of static inter-entity coupling required. This is consonant with Parnas' dictum on modularity [18].

2.2 Transformations in RDD

A version of the well-known ATM banking machine example is presented to illustrate the operation of the RDD process. Nouns from the original problem description, such as *Teller, Money, CheckingAccount* are selected as candidate object abstractions ("entities", hereafter). Sets of responsibilities are constructed for each of these entities, for example, according to the grammar:

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

where R is the set of atomic natural language statements and P are non-atomic statements of responsibility constructed from these. The initial entities are filtered and retained only if they can be conceived as bearing some kind of responsibility

(rule 1), so concepts like *Money* do not survive, except as the *balance* attribute of a *CheckingAccount* entity (rule 3). Collaborators are elicited (rule 2) where these server-entities are obviously involved in the fulfilling of client responsibilities; this information is entered on CRC cards. Figure 1 shows the initial collaboration pattern between these first-cut domain entities.

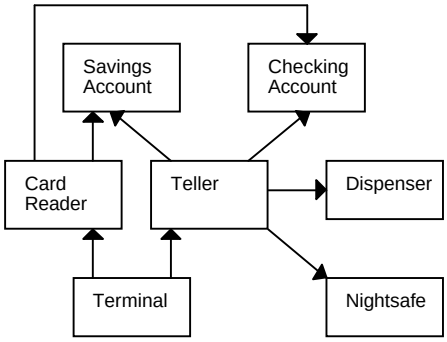


Figure 1: Pre-transformed RDD collaborations

Clearly, there is a degree of arbitrary interpretation in the early selection of object abstractions; nonetheless all entities selected must have the required behavioural properties. The elicitation rules (1-3) are perhaps less automatic than the later rules, but this is inevitable and not a fault. We have deliberately chosen the most obvious domain-influenced initial model, which fails to differentiate the activities of the *Teller* and fails to generalise on types of *Account*, although the RDD method would equally accept a more perceptive initial conceptualisation. The strength of RDD lies in its ability to reorganise the initial model according to modularising principles, forcing the invention of new abstractions.

In figure 2, the design process is more advanced, but not yet complete. An early and obvious generalisation on common responsibilities in the interfaces of *SavingsAccount* and *CheckingAccount* has generated the abstract *Account* parent class (rule 10). When all the responsibilities of the existing entities are listed, the two most overburdened entities are *Teller* and *CardReader*, both of which have over 7 responsibilities (rule 1), so these need to be split.

The *CardReader* must read, validate, encode and transmit account and PIN numbers, search for accounts and authorise connections to them. The choice of applying rule 5 over rule 6 to split *CardReader* is determined by the fact that its responsibilities are judged cohesive, since they all involve the same collaborators and attributes. According to rule 5, a new entity, *Verifier*, is spun off as a delegate of *CardReader* with the responsibility to handle and validate PINs. In retrospect, this is a good design decision, since *CardReader* has no need to retain

the PIN number (rule 3) once it has read the card and PIN number [4]. Notice how this is an instance of the *Chain of Responsibility* pattern [10], p223, in which the responsibility to *verify PIN number* is passed onto a delegate object. RDD will tend to generate a *Chain of Responsibility* pattern every time rule 5 is invoked.

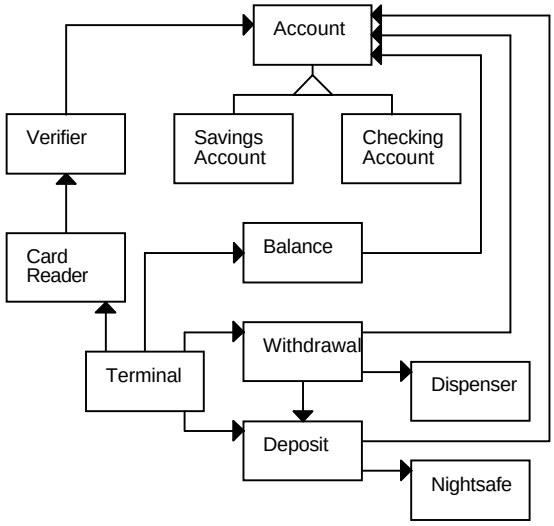


Figure 2: Partially-elaborated RDD collaborations

In contrast with this, the *Teller* entity must be partitioned into peers, because its many responsibilities are not cohesive (rule 6), even when restated. This is judged by observing how *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 distinct type of *Teller*-transaction. Note that the rule requires invention of new entities; and it is up to the developer to provide significant names, based on the partitioning of responsibilities. The elaborational rules 4-6 of RDD tend to generate manager entities to handle different system functions, by virtue of the constraint (rule 1) on the number of responsibilities assigned to each entity. We shall see later how this leads inevitably to instances of the *Command* behavioural pattern [10].

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 are coupled with their servers. At this time, areas of strong and weak coupling may be identified. In our example, one of the kinds of withdrawal to be supported is really a transfer of funds, which leads to the undesired cross-coupling highlighted in figure 3 (a): *Withdrawal* is the only manager-entity with a cross-linkage to one of its peers. This is strong evidence that rule 9 should be applied to remove the

cross-coupling. This rule mandates the introduction of a new entity to aggregate over the subsystem and manage the communication between the parts. Calling this new entity the *Transfer* manager, we encapsulate *Withdrawal* and *Deposit*, as shown in figure 3 (b). *Withdrawal* no longer needs a direct reference to *Deposit*. Notice how this is an instance of the *Mediator* pattern [10], p273: the *Transfer* entity coordinates the sequence of interactions between the *Deposit* and *Withdrawal* managers, such that these do not need to refer to each other; the anomalous *transfer money* contract is moved from *Withdrawal* to this new entity. RDD rule 9 always generates *Mediator* patterns, where other object-oriented methods might be content to let the cross-coupling remain.

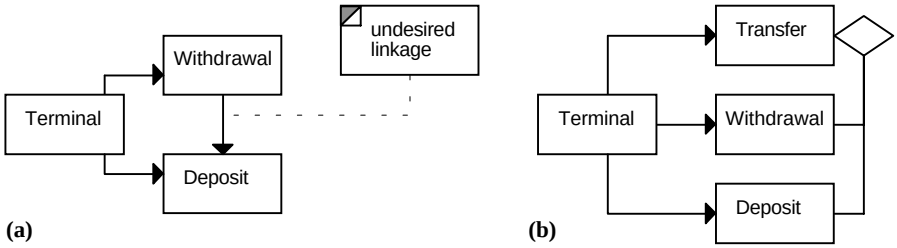


Figure 3: Aggregating over a closed subsystem

The last group of transformations involves considering how the contracts of *Account* are invoked by clients. Once *Account* responsibilities have been refined down to the level of individual services (by rule 4), these may be grouped into named and indexed contracts according to the distinct sets of clients which invoke them. According to rule 7, *Account* eventually offers five contracts, many of which only contain one service each: (1) *inspect balance* is used by *Balance*, *Deposit* and *Withdrawal*; (2) *make deposit* is used by *Deposit*; (3) *make withdrawal* is used by *Withdrawal* (grouping together the services *request withdrawal* and *withdraw amount*); (4) *commit changes* is used by *Deposit* and *Withdrawal*; and finally (5) *connect to account* is used by *Verifier*, (grouping together the services *valid a/c?*, *valid PIN?* and *a/c frozen?*).

Figure 4 (a) is a fragment of the system, showing how *Deposit* and *Withdrawal* invoke the *Account* contract (4) in common, but otherwise invoke apparently separate contracts (2) and (3) each. This is nonetheless suggestive, according to rule 10, that some generalisation of *Withdrawal* and *Deposit* should handle all communication with *Account*. Calling this new abstract entity a *Transaction* manager, the responsibility for invoking *Account* contracts migrates upwards to *Transaction*. In figure 4 (b), contract (4) *commit changes* is now invoked directly by *Transaction* (instead of separately by *Account* and *Withdrawal*). Contracts (2) *make deposit* and (3) *make withdrawal* are judged sufficiently similar, from the perspective of *performing a transaction*, that an abstract method *transact(int)* may

be provided in *Transaction*, which is subsequently redefined and dynamically bound in the descendants *Deposit* and *Withdrawal* to perform the appropriate deposit or withdrawal action. The effect of this transformation is to merge the communication paths leading from different manager-entities to *Account*. First, the duplicate paths to contract (4) are merged, then the paths to contracts (2) and (3) are merged (on the basis of polymorphism).

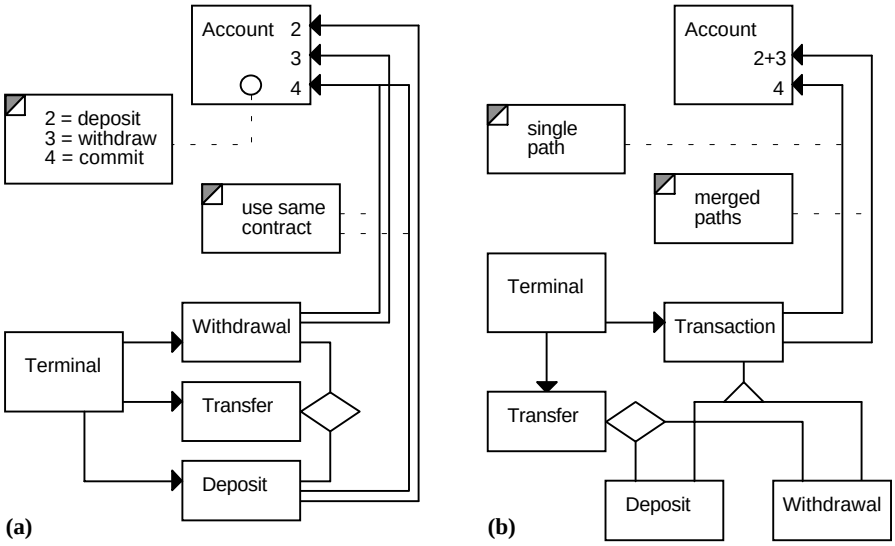


Figure 4: Generalising on commonly-invoked contracts

From figure 4 (b), it is clear that the revised contracts (2+3) and (4) are now only used by the client *Transaction*, so these may also be merged (by rule 7), making it possible to combine the *transact(int)* and *commit()* methods. Notice how these transformations lead systematically to an instance of the *Template Method* pattern [10], p325, in the form of *Transaction*'s *handleRequest(Account&)* method. This method is the template for all single transactions on an *Account*. First, it 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)*; c.f. [10], p327.

The continuing process of generalisation (rules 10, 7) eventually predicts an abstract superclass for *Balance*, *Transfer* and *Transaction*, which all communicate with *Account*. Since this entity will be the root of all managers handling banking requests, we reintroduce *Teller* as the abstract superclass in the final design in figure 5, having a single contract (1+2+3+4) with *Account*. We emphasise that it is the similarity in the way different manager-entities communicate with *Account*, judged according to contracts, which motivates the introduction of the *Teller*

entity; the fact that this corresponds to an existing concept in the analysis domain is serendipitous. Notice how *Teller* is an instance of the *Command* pattern [10], p233: *Teller* encapsulates different kinds of abstract banking requests, which are fielded by its more concrete subclasses. This could be represented by a polymorphic *handleRequest(Account&)* method. Further merging of *Teller* and *Verifier* is prevented by their too-different external interfaces.

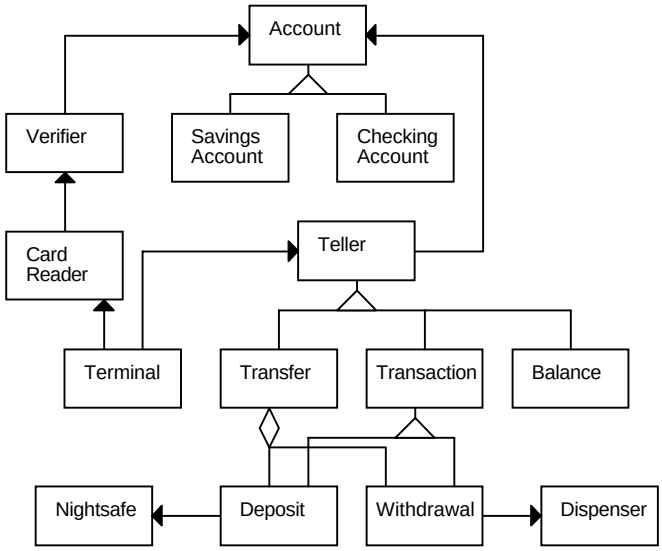


Figure 5: Fully-transformed RDD collaboration graph

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. The per-service weighting measure (rule 7) lets the designer see how many services each collaboration is carrying, in highly-coupled systems. It provides a rationale for placing subsystem boundaries: you aggregate over the most tightly-coupled parts of the system (with high per-service counts) and break the system at weakly-coupled points (with low per-service counts). RDD subsystems are eventually much better motivated than Coad-Yourdon *subjects* [6].

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 showed 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 [13, 19] 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 [25, 8] and a conceptual modelling approach [24]. 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 [14]; whereas traditional message-passing is more like CCS [17].

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 (see figure 6 for examples); 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 (see also figure 6). 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 necessarily cease to exist. 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* and *return* events, in which a *Copy* of a book and a *Borrower* participate. Since *Copy* and *Borrower* are so far unrelated in the EDG, a new associative entity, named *Loan*, is introduced. The *borrow* event marks the creation of the *Loan* entity, 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, polymorphic families of methods are devised corresponding to one method per system event handled in each entity. The flow of control is initiated from the dependent associative entity to the participating master entities, each of which must have a version of the method to react to the event. The polymorphic *borrow* method constructs a *Loan*, dispatching the same *borrow*-message to the participants, where it (variously) decrements a *Borrower*'s book allowance and marks a *Copy* as unavailable to other library users.

3.1 The Rules of EDD

Once more, we are interested in the potential of EDD as a systematic design process. In table 2, we have distilled 10 rules (coincidentally, the same number as for RDD) from the principal sources [8, 24, 25], by ignoring the more subjective aspects of the design processes described there. 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 (in ERM terms) 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.

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 associative entity is created for each distinct set 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.

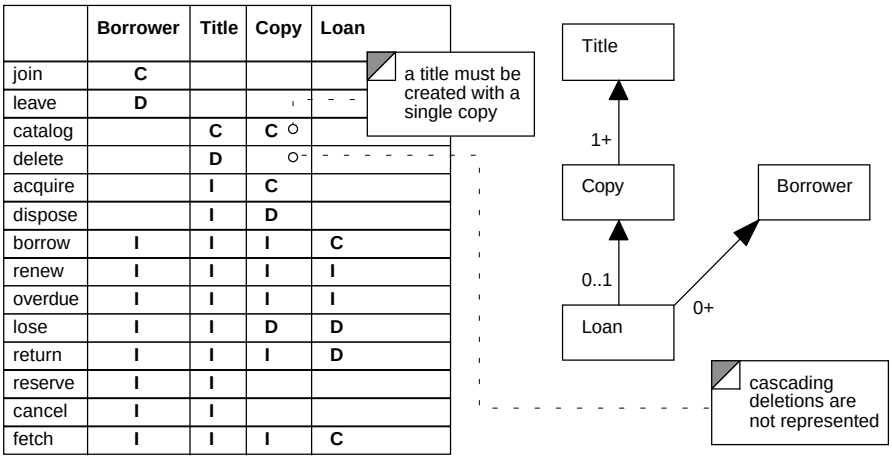
EDD rule 9: All events become methods invoked on the dependent entities, delegating to the participating master entities; dependents handle the intersection of their masters' events.

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

3.2 Transformations in EDD

Most of the system layering activity is performed during the elaboration phase (rules 5-8), 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.



	Borrower	Title	Copy	Loan
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		
cancel	I	I		
fetch	I	I	I	C

Figure 6: OET and EDG after addition of Loan

In figure 6, *Loan* is the latest associative entity introduced, according to rule 7, to manage the common events {*borrow, renew, overdue* and *return*}, in which the unique set of entities {*Copy, Borrower*} participate. *Loan* has also been attached as the latest child in the EDG, and made dependent on *Copy* and *Borrower*. The multiplicity figures state how many *Loans* may exist for each *Copy* or *Borrower*. Note how, in accordance with rule 7, the OET contains I-events for *all* the the master entities, viz. {*Borrower, Copy, Title*}, impacted by *Loan* C-, I- or D-events, such as *renew*. This allows *renew*'s consequences to propagate to all the 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 *renew* might have on *Title* - it is possible for an event to have a null effect; we show how this is handled below.

In figure 6, 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 separate creation of the *Reservation* associative entity (by rule 7). This is shown added to the OET and EDG in figure 7. Note how there are no longer any I-entries in the OET which are not covered by some

existing association, indicating that the elaboration phase is now complete. Every time a new entity is introduced, existing events are examined for their impact on this entity (rule 1). For example, the *fetch* event is identified as a (D)elete-event for a *Reservation* and a simultaneous (C)reate-event for a *Loan*. This is the only event to involve both a *Loan* and a *Reservation*. No new associative entity need be created (according to rule 8), since a pair of time-separated events is always necessary to (C)reate and (D)elete each new associative entity introduced. Furthermore, there are no unconnected entities in the EDG (rule 8).

	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**

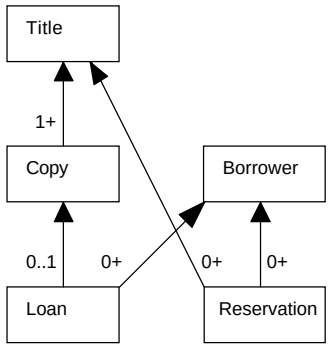


Figure 7: OET and EDG after addition of Reservation

Rule 9 is applied to convert *all* event-participations into methods (one method per entity-event). A large number of methods are generated. Every associative entity automatically becomes the root of a call-graph for each event it manages. For example, the *renew* event is translated into: *Loan::renew()* - update the due date of the loan, which dispatches to: *Copy::renew()* - reduce the time to inspection; and also to *Borrower::renew()* - restore borrowing privileges. *Copy::renew()* dispatches to *Title::renew(),* which eventually is a null operation, a degenerate method. Notice the similarities between the method interfaces of each associative entity and the participating entities it manages: every dependent entity manages the intersection of its masters' events. Just like a *Composite* pattern [10], p163, associations encapsulate part-whole hierarchies which respond to the same sets of messages. Just like the *Chain of Responsibility* pattern [10], p223, the events are handed on to the next component in the hierarchy, until components are reached which perform significant parts of the computation. EDD will always generate these two patterns in abundance. Finally, rule 10 is applied to eliminate degenerate methods, such as *Title::renew().* Groups of methods may be renamed, to increase their mnemonic salience, for example: *transact* may convert into *buy()* and *sell()* for objects playing these complementary rôles.

3.3 Subsystems and Coupling in EDD

Dependent entities in EDD have some of the characteristics of ERM's linker entities (they represent associations and store foreign keys) but also have characteristics of RDD's *Mediator* patterns (they are devised in response to a need to communicate events to their participants). However, data aggregations may be handled differently in EDD than in other object modelling approaches [22, 20, 5, 19, 9]. 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. EDD optimises the *construction order* of a system. It is easy to draw *entity life history* diagrams (ELHs) [1] 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. Consider that *Borrower* and *Copy* satisfy the interface of *Loan* (because they respond to all *Loan*'s events, and also to other events). It is tempting, but wrong, to think of *Loan* as a generalisation of *Borrower* and *Copy*, since *Loan* implements the common events differently from either class. Instead, all three classes should inherit from an abstract class which defines the *Loan* interface (but not the implementation). This abstract class is an instance of the *Composite* pattern base class [10], p163, whose concrete descendants respond to each message and then delegate these messages to their own components. EDD inevitably produces large numbers of composite patterns, because of the emphasis on shared participation in events. More important master 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* [10], p223, which allows events to be dispatched to one starting point, then forwarded down the line to some object which eventually executes the major part of the response.

4 Conclusions

This paper has examined two different approaches to object-oriented design, each of which elevates a different modularising principle: *contract minimisation* and

existence dependency. Different *Design Patterns* emerged during the application of the methods, showing how they take different design decisions when structuring a system. We showed above how these *Design Patterns* emerged naturally and are in fact an inevitable part of the layering and transformational rules of each method. We characterised each method in a semiformal way, so that the reader could see more easily the link between the "rules" of the methods and the particular *Design Patterns* generated.

4.1 Emergent Patterns and Coupling Characteristics

The kinds of subsystems and layering suggested by each approach are different. EDD promotes unidirectional data coupling in its modelling, so is unable to handle inverse effects, such as a cascading deletion (see note in Figure 6), which is formally forbidden. An *Observer* pattern [10], p293, could be used to register master entities with their dependents, although this would significantly worsen the coupling characteristics. RDD is most 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. It is no accident that RDD generates all *behavioural* patterns (*Mediator, Command, Template Method, Chain of Responsibility*), since its focus is on responsibilities and behaviour. EDD, on the other hand, is dominated by the *structural* pattern, *Composite*, determined by the EDG structure. The event-participation model leads directly from this to the emergent *Chain of Responsibility* pattern.

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 later stage to aggregate over the closed ring of collaborations involved in transferring money, goods and ownership; whereas in EDD, *Sale* will necessarily exist from the beginning, 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

There is far more to object-oriented system design than elaborating analysis models to the point where they can be implemented. This is not truly appreciated by seamless approaches [6, 7, 26, 13, 19]. System partitioning has only been

treated informally in many other presentations [20, 15, 3, 13]. Initially, we had set out to identify, codify and then compare two design approaches which offered some leverage in the system design stage. When we applied our semi-formal rules to example designs, we found again and again that recognisable Design Patterns emerged. In particular, we gave examples of instances of *Mediator*, *Command*, *Chain of Responsibility*, *Template Method* and *Composite* that were generated automatically. These five patterns all have the property that they reduce cross-coupling in system design. The *Façade* pattern [10], p185, also exhibits this property, but whereas the other five may be derived from the internal coupling characteristics of systems, *Façade* is always imposed externally, in situations where components are being bundled for convenience. Some patterns, such as *Adapter* [10], p139 and *Bridge* [10], p151, are neutral with respect to cross-coupling: they introduce an extra layer of composition to reduce the number of specialised variants of a class. Other patterns, such as *Proxy* [10], p207, *Flyweight* [10], p195 and especially *Observer* [10], p293, actually increase cross-coupling and mutual dependency. This reinforces our confidence in the five emergent patterns as indicators of high-quality system designs. We note that Design Patterns have not been used in this manner before - as litmus paper for testing the strengths, weaknesses and preferences of design methods.

References

1. Ashworth, C. and Goodland, M., *SSADM: A Practical Approach*, McGraw-Hill, 1990.

2. Beck, K. and Cunningham, W., "A laboratory for teaching object-oriented thinking", *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, pub. *Sigplan Notices, 25(10)*, 1989, 1-6.

3. Booch, G., *Object-Oriented Analysis and Design with Applications, 2nd edn.* Benjamin-Cummings, 1994.

4. Budd, T., *Introduction to Object-Oriented Programming* Addison-Wesley, Reading MA, 1991.

5. Coleman, D., Arnold, P., Bodoff, S., et al., *Object-Oriented Development: The Fusion Method*, Prentice Hall, 1994.

6. Coad, P. and Yourdon, E., *Object-Oriented Analysis*, Yourdon Press, 1991.

7. Coad, P. and Yourdon, E., *Object-Oriented Design*, Yourdon Press, 1991.

8. Dedene, G. and Snoeck, M., "Formal deadlock elimination in an object-oriented conceptual schema", *Data and Knowledge Engineering, 15*, 1995, 1-30.

9. Firesmith, D., Henderson-Sellers, B. and Graham, I., *OPEN Modelling Language (OML) Reference Manual*, SIGS Books, 1997.

10. Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

11. Gibson, E. A., "Objects born and bred", *BYTE magazine, 15(10)*, 1990, 255-264.

12. Graham, I. M., *Migrating to Object Technology*, Addison-Wesley, 1995.

13. Henderson-Sellers, B. and Edwards, J., *Book Two of Object-Oriented Knowledge: The Working Object*, Prentice Hall, 1996.

14. Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

15. Jacobson, I., Christerson, M., Jonsson P. and Övergaard, G., *Object-Oriented Software Engineering: a Use-Case Driven Approach*, Addison-Wesley, 1992.

16. Meyer, B., *Object-Oriented Software Construction, 2nd. edn. rev. and enl., Prentice-Hall*, 1997.

17. Milner, R., "A calculus of communicating systems", *Lecture Notes in Computer Science*, Springer, 1980.

18. Parnas, D., "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, Yourdon Press, 1979.

19. Rational, *UML 1.1 Reference Manual*, Rational Software Corp., September, 1997; also available through: *http://www.rational.com/uml/* .

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

21. Rubin, K. and Goldberg, A. "Object-behaviour analysis", *Comm. ACM, 35(9)* 1992.

22. Shlaer, S. and Mellor, S., *Object-Oriented Analysis: Modelling the World in Data*, Yourdon Press, 1988.

23. Simons, A. J. H., "Object Discovery: a process for developing medium-sized object-oriented applications", *Tutorial 14, European Conf. Object-Oriented Prog.*, Brussels (1998); see also: *http://www.dcs.shef.ac.uk/~ajhs/discovery*.

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

25. Snoeck, M., "On a process algebra approach to the construction and analysis of MERODE-based conceptual models", *PhD thesis, Katholieke Universität Leuven* 1995.

26. Waldén, K. and Nerson, J.-M., *Seamless Object-Oriented Architecture*, Prentice-Hall, 1995

27. Wirfs-Brock, R., "Responsibility-Driven Design" *Tutorial Notes, ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, 1996.

28. Wirfs-Brock, R. and Wiener, L., "Responsibility-driven design: a responsibility-driven approach", *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, pub. *Sigplan Notices, 25(10)*, 1989, 71-76.

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