-
- Book

The event modeler

New patterns and transformations for behavioral modeling

V7

SORRY NO UML YET, BUT THIS DRAFT IS REVIEWABLE

Contents

1. Introduction

This is a unique book on event modeling for enterprise applications. It says many new things and challenges prevailing orthodoxies.

The book presents a new knowledgebase of patterns for drawing event models. Event model patterns are analysis patterns first and design patterns second. The analysis goal is to find out what the business rules are and specify them correctly. The design goal is to design a process structure that meets performance requirements and can be readily maintained.

This book is perhaps the first to take a view of event modeling that applies equally to object-oriented software design and procedural program design in enterprise applications. It presents principles that apply to classes and objects as well as procedures.

Readers will range from analysts and designers working on enterprise applications, to students of computer science. For readers familiar with UML, an event model is a kind of object Interaction structure; you may choose to stereotype the control object with <<event>> and other objects with <<entity>>.

1.1 Professional analysts and designers

It has been reported that 70% of the world's programmers are writing code for enterprise client-server systems. An enterprise application supports a business by providing it with information about the real-world objects that the business seeks to monitor and perhaps control.

Enterprise applications are driven by events. Events maintain state by imposing and assuring business rules. Yet knowledge of how to build event models to capture business rules and ensure data integrity is thinly distributed. This book can help anybody who works on enterprise applications.

Your time is at a premium. Patterns save you time. You can apply many of the patterns immediately. They help you:

- adopt modern analysis techniques that work with new technologies
- extend database development methods with object-oriented analysis
- understand and address the limitations of object-oriented analysis and design methods.

It is clear that many, perhaps most, applications have been written with little or no methodology. But most of our legacy systems only worked properly after a protracted process of iterative development, they contain redundant code and they are difficult to maintain.

1.2 Academics

The beginning of wisdom for an analyst or designer is to realise that a one-dimensional methodology, be it object-oriented or relational, is only part of what is needed.

This book describes the specification of business rules and constraints in a style that can be reconciled with formal specification. In doing so, I hope to break the stranglehold that the 'object model' 'and relational theory' and have over university teaching. Both theories make good servants and poor masters. We need a broader theory that encompasses process structure as well as data structure, and event-orientation as well as object-orientation.

2. Preface

‘One must be careful to define an **event** clearly - in particular what the initial conditions and the final conditions are’. Richard Feynman writing on quantum electro dynamics

2.1 The event modeler

Models are tools we use to analyse and define the requirements for software systems. A comprehensive model defines both structural things/features and behavioral things/features. I discuss the modeling of structural things/features in the companion volume “The entity modeler”, and very briefly on this one. This book focuses on modeling behavioral things/features in the form of event models; see the How and When columns in the table below.

Models are drawn at various levels of abstraction, from models of code in a specific programming language, through specifications for such code, to specifications of an enterprise regardless of any software that might be written. This book focuses on the specification of an enterprise application; see the middle row in the table below. (This table is a kind of cut-down Zachman framework.)

Orientation / ***Level***	Structural model	Behavioral Model	
	Entity model	Event models	Entity state machine models
	What	How	When
Enterprise model			
Enterprise application model	“The Entity Modeler”	This book	This book
Technology model			

In ordinary conversation, event can mean an event instance (a message or process) or an event type (a class of events). Similarly, model can mean a live model (a running enterprise application models the real world) or a dead model (a specification for a live model). I started writing this book with careful attention to such distinctions. This pedantry made the text unreadable. I believe you will find it is easier to interpret the words event and model according to their context, as you do in conversation.

Events drive enterprise applications. An event model shows how an event affects entities. Event models are used by software engineers working on enterprise applications to:

- refine higher-level requirements and use case models
- facilitate discussions with business people and clarify requirements
- specify the business rules and processing constraints for developers
- specify control objects and Interaction structures

Trying to meet all of these goals in one model creates some tensions that are reviewed in this book.

I am interested the challenge of helping the Agile Modeler. The Modeler traditionally takes the view that specification and design up front are important. The Agilist tends to the view that design up front is a waste of time, that models are a distraction from real work, that success depends mostly coding, testing and verbal communication.

The Agile Modeler keeps event models simple, is aware of different modeling options, understands trade offs between them, and introduces complexity only when and where it is needed. The Agile Modeler knows a variety of approaches and embraces the philosophy of a well-know guru.

> “It is important not to be dogmatic. Every approach has its advantages and limitations. You must use a mixture of approaches in the real world and not mistake any one of them for truth”. James Rumbaugh

Getting event models "right" is not straightforward. Event modelers face awkward questions to be explored later. Model patterns and transformations shed light on these questions.

2.2 Analysis patterns

It is increasingly apparent that a software development process is not enough. There is more wisdom to be taught through patterns and rules of thumb than through the stages and steps of a process

The analysis patterns in this book are similar to object-oriented design patterns in some ways, and different in others. I will highlight a few points of correspondence, but the emphasis here is mostly on analysis and design questions for enterprise applications.

For more years than I care to remember, I have taught analysts and designers to recognise patterns in entity models (cf. class diagrams), state-transition diagrams (aka state charts) and Interaction structures and to be aware of possible transformations between related patterns. As long ago as 1994, Grady Booch pointed out the kinship between my ‘analysis patterns’ and Coplien’s work on ‘generative patterns’ for object-oriented design. This prompted me to document my patterns more thoroughly. I ended up with many more patterns than one book can accommodate, so I have to publish the entity model patterns and event model patterns separately.

Patterns raise productivity. They speed up thinking and help you to avoid mistakes. They apply equally to rapid and slow development, to engineering of new systems and reengineering of legacy systems.

Patterns raise quality. They help you to elicit requirements. They prompt you to ask business analysis questions and quality assurance questions. Look out for the bad patterns as well as the good ones.

Patterns connect things. They are recognisable structures or templates that capture expert knowledge about connecting the modules, classes and objects of a software system, via interfaces, relationships and events.

Patterns make wider connections. They enable you to link apparently distinct analysis and design techniques, coordinate different views of a system into one coherent specification, reconcile object-oriented and relational ways of thinking.

Analysis patterns help you to get things right, discover the relevant requirements and design so as to minimise redundancy. If one or two of the patterns and questions save you a few days effort, then this book will have paid its way.

3. Manifesto

I propose that the OMG (Object Management Group) host an EMG (Event Management Group) whose beliefs and ideals are encapsulated in the following manifesto:

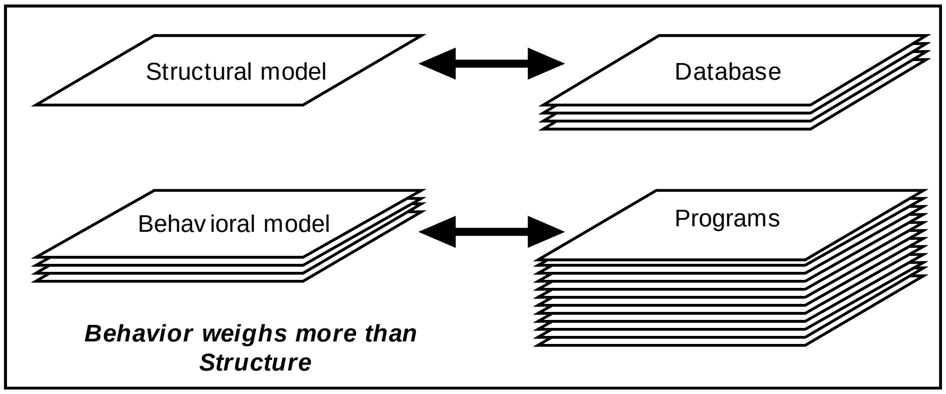
EMG 1: A software system does nothing but validate input data, store persistent state data, and derive output data from inputs and state.
EMG 2: The first challenge in application design is to maintain data integrity by defining business rules and controlling the many users and many clients that may update shared state; the second is to synchronise duplicated state; the third is to synchronise distributed state.
EMG 3: The ideal is to build applications on top of a service-oriented architecture.
EMG 4: Three levels of process specification can and should be distinguished: long-running business processes, shorter running use cases and atomic business services.
EMG 5: Entity-oriented and event-oriented views of a system are complementary, equally important, and united by event effects (each the effect of a transient event on a persistent entity).
EMG 6: Analysts must posit transaction management of events, and the key specification artefact is the event effect (not the operation(s) that implement an event effect).
EMG 7: Business rules are invariants of state and preconditions and post conditions of event effects; volatile rules should be stored as user-updatable attribute values.
EMG 8: Caching persistent data outside the persistent data store is an optimisation technique, not a design principle.
EMG 9: Persistence and flexibility undermine class hierarchies and aggregates; the more numerous and long-lived the entities, the less fitting the class hierarchy; the more multi-dimensional the entity model, the less fitting is the aggregate entity
EMG 10: The art, the skill, the goal of the architect and the analyst is to keep the design as simple as possible.

4. The need for behavior modeling

This chapter discusses the fragility of structural rules where the entities are numbered in thousands or millions, and they persist for months or years. It proposes reasons why structural analysis must be supplemented by behavioral analysis.

4.1 The weight of behavioral rules

In most Enterprise Applications, the database schema definition is far outweighed by the code of the programs that access that database. In the days when the database schema and program listings were stored in hanging files, it was highly visible that the programs outweighed the database. Nowadays, much of the program code may be written in the form of operations attached to classes or objects in a structural model, but it remains true that the operations weigh more than the attributes, that the behavior weighs more than the structure.



Given our models are abstractions from the code, we must expect a similar imbalance between structural and behavioral models. We need both data/structural and process/behavioral views of a system, but the latter will outweigh the former.

4.2 The equivalence of structural and behavioral rules

The difference between structural and behavioral rules is more subtle that it might seem from the introductory RAP group papers.

Principle: “You can replace an invariant rule (constraint or derivation) of a persistent entity by a behavioral rule (a precondition or post condition) of every event that might threaten the truth of the rule.”

E.g. one might replace an invariant constraint on an Account entity (or class):

ENTITY: Account	**Invariant**
AccountBalance	> 0

by a precondition applied on a Withdrawal event (or operation):

EVENT: Withdrawal (AccNum, Amount)		
Entities affected	**Precondition**	**Post condition**
Account	WithdrawalAmount < AccountBalance	AccountBalance = that - Withdrawal

You don't want to specify or code any rule more than once if you can help it. The goal must be both Single Point of Declaration and Single Point of Deployment.

At first sight, a behavioral constraint seems less unsatisfactory, because if two operations can both reduce the AccountBalance then both operations would have to contain the same rule, same behavioral constraint. An invariant Constraint in an entity model seems preferable. The rule is declared at one point. Every operation on the AccountBalance attribute will be constrained in the same way.

But Single Point of Declaration may not mean Single Point of Deployment, since to implement an invariant rule you might have to code it in each operation anyway.

And there is a more general problem with invariant rules where the entities are numbered in thousands or millions, and they persist for months or years. What seems an invariant rule today may turn out over a longer time to be

- contradicted by updates, or
- have exceptions, or
- be true only now and then, or
- be changed.

These points are considered in turn below.

4.3 Rules that are contradicted by updates

Time changes everything: persistence undermines invariance. If you maintain only a partial history of past events, then many rules are sooner or later contradicted by events that overwrite historical data. E.g. consider an order processing system where the derivation is:

- Item.Value = Item.Quantity * Product,Price.

Is this an invariant rule? The condition must be true when an order item is placed. But (unless you maintain a full history of product prices) as soon as the product price is updated the condition is no longer true. So, if you write a program to test the integrity of the invariant rules in the database, it will report perfectly valid orders as being in error.

4.4 Rules that have exceptions

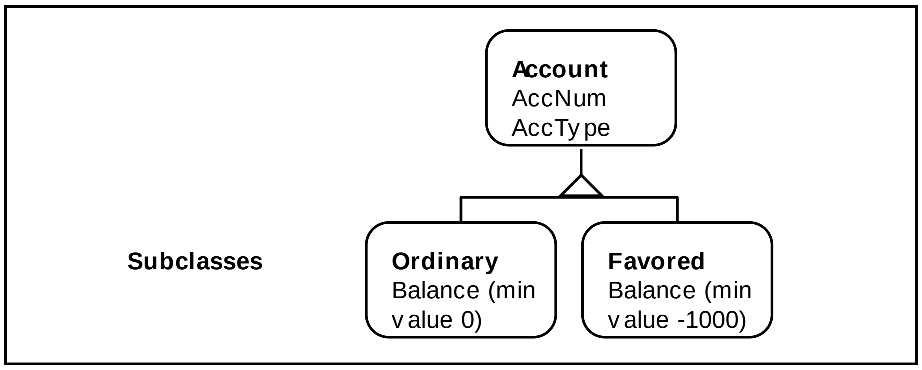
Time changes everything: persistence undermines invariance. The longer a system lasts, the more likely that an exception to the rule will be discovered. What seems at first to be invariant rule of a class may turn out to be true only for some members of that class.

Consider the invariant constraint on the Account entity that 'Account Balance > 0'. What if a variation is introduced such that favoured accounts are allowed to go overdrawn by $1000? Five options are considered below.

4.4.1 Option 1: Constraints on ‘subclass’ attributes

An OO designer's first thought might be to model variations by drawing a class hierarchy.

- Subtype accounts into 'Ordinary' and 'Favoured'.
- Specify different Invariant Constraints for the balance attribute of the two subclasses.



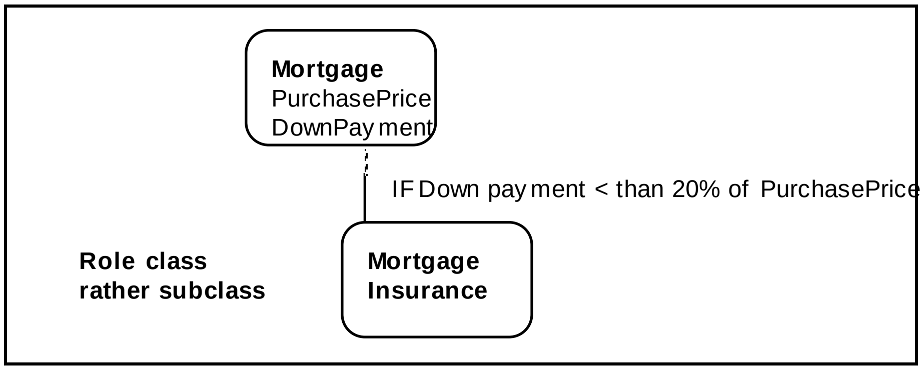
But drawing a class hierarchy should, I dare to suggest, be the last approach one considers. The danger is that the number of subclasses will grow very large, and the structure of super and subclasses will prove volatile as rules are changed.

I offer a rule of thumb: Do not try to express the options of a case statement as a class hierarchy in an entity model until or unless you know this case statement would otherwise be replicated in several operations for a considerable period of time.

4.4.2 Option 2: Constraints on ‘role class’ attributes

Generally, a flexible alternative to defining subclasses is to define 'aspect' or 'role' classes connected by association relationships as children of the basic class. Designers may then use forwarding or delegation rather than inheritance as a means to invoke operations of the classes.

Let me mention an illustration given to me by Haim Kilov. The rule is that if the down payment on a property is less than 20% of the purchase price, then you must connect a Mortgage Insurance to the basic Mortgage object.

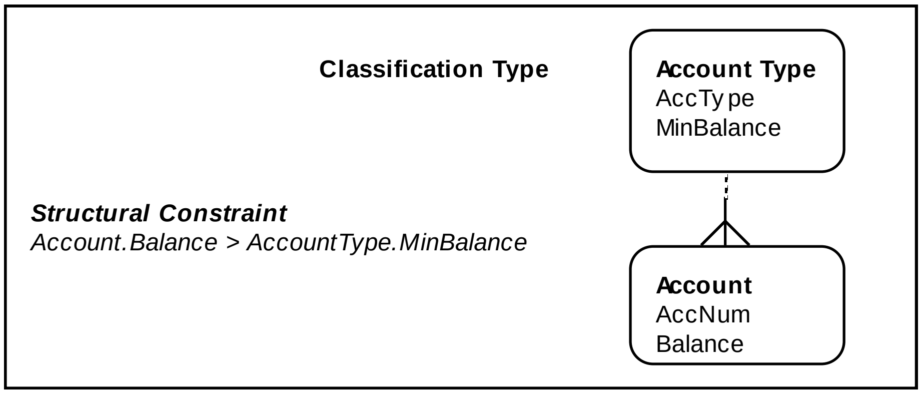


For the given example of Ordinary and Favoured Accounts, role or aspect classes do not seem any better than subclasses.

4.4.3 Option 3: Constraints on 'classification type' attributes

A more flexible approach is to specify a classification type (that is, a parent of the basic entity) with attributes that hold values used in the business rule. So any transaction that invokes a relevant operation on the basic class must first retrieve the relevant rule element from the classification type.

- Specify the class AccountType with the attribute MinBalance.
- Specify the rule 'AccountBalance > AccountTypeMinBalance' as an invariant Constraint.
- Store AccountType instances 'Ordinary' and 'Favoured' with different MinBalance values.



Note that this Invariant Constraint rule refers to data attributes owned by objects of different classes. So which class does the rule belong to?

'Attaching an invariant to a particular class is not a specification decision for analysts; it is, if anything, an implementation decision for designers.' Haim Kilov

Haim's suggestion still leaves somebody with the task of placing the rule. You do have to specify the rule somewhere.

4.4.4 Option 4: Constraints on an event

What is it that brings the objects together? It is the discrete event that triggers a transaction. So it is at least a possibility that the constraint is better specified as a behavioral rule.

EVENT: Withdrawal (AccNum, WithdrawalAmount)			
Entities affected		**Precondition**	**Post condition**
Account	o-- (ordinary)	WithdrawalAmount < AccountBalance	AccountBalance - Withdrawal
	o-- (favored)	WithdrawalAmount < AccountBalance +1000	AccountBalance - Withdrawal

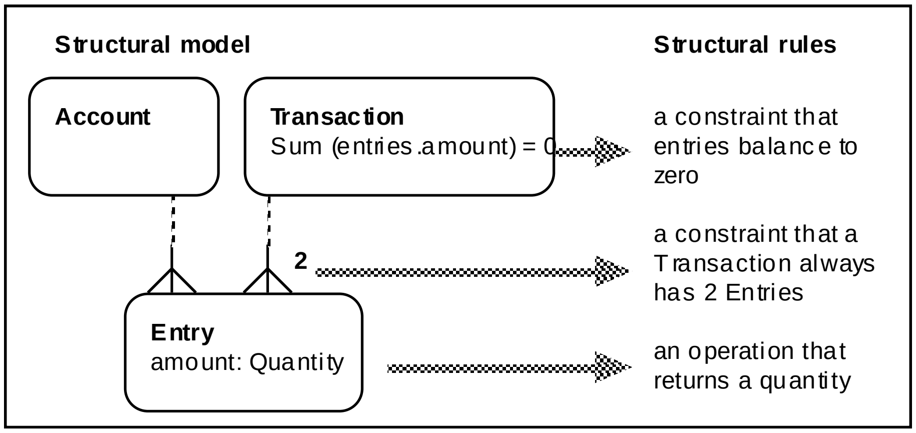
4.4.5 Option 5: Control flow in a procedure

The last approach is to specify rule variations under options of a case statement within a procedure. Even this old-fashioned procedural approach can give us both Single Point of Declaration and Single Point of Coding, provided that the case statement appears in only one operation.

4.5 Rules that are transient - true only now and then

Time changes everything: persistence undermines invariance. A condition that is true immediately after a special correction process has been run, but not true at any other time, is not best regarded or documented as an invariant vonstraint in an entity model.

E.g. Consider a double-entry bookkeeping example presented in 'Analysis Patterns' by Martin Fowler. He specifies two apparently invariant constraints on the structural model thus:


Structural model
Structural rules
Account
Transaction
Sum (entries.amount) = 0
a constraint that entries balance to zero
2
a constraint that a Transaction always has 2 Entries
Entry
amount: Quantity
an operation that returns a quantity

As Martin says below, he is only using this simple example to introduce more complex accounting patterns. My aim here is to explore the meaning of structural and behavioral constraints.

4.5.1 Dialogue between the editor and Martin Fowler

Graham: The rule that a transaction has 2 entries appears in an entity model. It is the basis of double-entry book keeping. However, the rule 'Sum (entries.amount) = 0' surely cannot be an invariant rule, because that negates the whole point of double-entry book keeping. The only reason to record both positive and negative entries is to test one against the other. They may be recorded separately, perhaps by different people. The idea is to check for errors by running a reconciliation process at a later date.

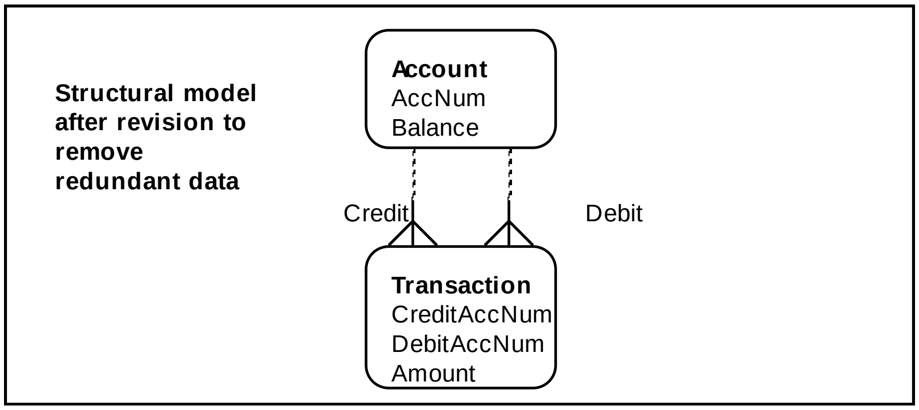
Martin: It depends. I have seen cases where the rule is part of the structure and you can only create balanced transactions. Other systems use a reconciliation process and I should have discussed that in the pattern. Another pattern that applies to reconciliation, is that of corresponding accounts. Most reconciliations that I have come across use something like this. That's what happens when an individual reconciles a bank account.

Graham: If 'Sum (entries.amount) = 0' is a transient post condition that holds true only at the end of a reconciliation process, then the rule should be documented with this event in a behavioral model, not with a class the structural model. At a meeting of the RAP group, Mike Burrows discussed a share-trading system in which entries do not balance at all points during the processing cycle, while individual credit and debit transactions are processed. The interesting part of the design was the post condition of the discrete end-of-cycle process that reconciled persistent data on distinct databases.

Martin: Yes, that will work. Another way is to rephrase the rule along the lines of "if reconciled then sum of amounts must be zero"

Graham: If 'Sum (entries.amount) = 0' truly is an invariant constraint, then there is no need to run a reconciliation process. You might code it for system testing, but you should switch it off afterwards. Or you might tell the designers that a reconciliation process will be run, threaten that they will be fined by the amount of any discrepancy, then not actually bother to code it at all!

Then your model can be improved. There is no need to store two entries for the transaction, since they carry the same data. You should specify the amount as an attribute of the transaction class and revise the structural model thus:



Martin: Yes that is true. I considered showing this. But multi-legged transactions are more useful, and not much more complex, so I used the two-legged case as an entry into the multi-legged case.

We certainly need to explore this ground further, especially the patterns around letting incomplete and inaccurate data into the system and doing later reconciliation. Too many people want to bar errors at the gates, when often it is more effective to let them in and hunt them down once they are safely inside.

4.6 Rules that change

The longer data persists, the more likely a seemingly invariant rule is modified during the life of an entity whose state is recorded in the system. Even the laws of the land change from time to time. There are two ways to meet the challenge of volatile rules. The first is to store the rule itself as an attribute value that can be updated by end users. This approach has a limited application.

The second and more general approach is to define the rule as a transient pre or post condition of one or more processes (rather than an invariant of a data structure). It is often better, easier or safer to specify rules in a behavioral model, that is:

- in a process model rather than a data model
- with the events rather than the entities.

This helps us to minimise evolution problems, since changing a process structure is normally easier than changing a data structure. See also the later chapter <Preconditions and control flow conditions>.

4.7 Conclusions

Invariant rules are essential. We do need an analysis and design method that helps us to specify invariant rules, especially those that define the data type of an attribute or the multiplicity of a relationship. Most analysis and design methods already focus on specifying invariant rules; they offer various means of declaring such rules on a data model or class diagram.

But the passage of time undermines structural models. Persistence undermines invariance:.

- the substance of a thing grows, changes or decays
- apparently fixed aggregations turn into loose associations
- apparently fixed types turn into temporary states
- rules are changed.

What seems an invariant rule today may turn out over a longer time to have exceptions, or be contradicted by updates, or be true only now and then, or be revised. The longer the view you take, the more that persistent types turn into transient states, and the more that invariant rules turn into behavioral rules. And changing a process structure is easier than changing a data structure.

So, it would seem better to specify rules in a behavior model rather than a structural model, as transient rules rather than invariant rules. For specifying enterprise applications (where the stored data persists for years, where the data may be distributed across several databases, where the rules evolve) behavior modeling is essential.

5. PART ONE: EVENT MODELS

This chapter discusses the insubstantial nature of things and the fuzziness of the real world. It proposes reasons why event-orientation is as important object-orientation, and why phenomenology is as important as ontology.

Let me tell you a story. This is story about the passage of time and the insubstantial nature of things.

The ship of Theseus: episode one

Kero the boat builder sold a ship to Theseus the trader. Theseus bought a brand new ship's log and set off on his first voyage, trading between Greece and Persia.

Theseus had exacting standards and was rich enough to maintain them. After his first trip, Theseus paid for Kero to renew part of the deck that had been scratched when a load was dragged across it. After his next trip, Theseus paid for Kero to replace the main sail, which had been torn a little. After his third trip, Theseus paid for Kero to replace the rudder, which had become worn and loose.

Eventually, after many more trips, Kero had renewed every part of the original ship, every single molecule of its substance. Theseus noted each repair event in the ship's log.

One day when Theseus was away sailing and trading, Kero looked around his boatyard. He noticed that all the parts of Theseus' original ship were lying there. With new nails, some canvas, twine and a little spit and polish, his slaves were able to rebuild the original ship.

Meanwhile, out at sea there was a wild storm. Wave after wave swept over Theseus as he stood at the wheel. Eventually, Theseus gave the order 'abandon ship'. He swam for the shore carrying his log book in a leather pouch.

When he got back home, he wrote 'sunk in storm' in the log book and sought out Kero. Could he please have another ship like the first? Kero couldn't resist a little deception 'I salvaged your ship after you abandoned it. It is sitting in my boatyard. I'll be happy to sell it to you, at the full price of course.'

When Theseus overcame his surprise and his reluctance to pay twice for the same ship, he wrote 'salvaged' in the log and set out on another voyage.

The objects we think about are not concrete things: Any way you look at it, the ship of Theseus is not simply a tangible object; it is more abstract; it is behavior; it is a memory. The ship exists in Theseus' mind; it is his experience of a thing that carries him around the Mediterranean. The ship is given a continuity of existence not only by his memory but by also by his written record. An object is something that persists for a while and is remembered. Objects are only memories of things.

The ship of Theseus: episode two

Another surprise awaited Theseus when he next returned home. A handful of his crew, poor swimmers, had been forced to cling to the apparently sinking ship. After the storm abated, they were able to bale out the sea water and bring the ship home. The laws of salvage meant that they were now entitled to claim ownership. They bought a new ship's log and set up in business, competing with Theseus.

Kero felt obliged to tell Theseus the truth. To avoid any confusion between the two almost-identical ships in his boatyard, he nailed nameplates to their prows to distinguish them. Actually, this didn't resolve the confusion at first, because he called them 'New Ship' and 'Old Ship'. Nobody else was clear which was which, so he copied the names into the corresponding log books to make things clear.

There are different models of reality: If objects are distinguishable entities, you ought to be able to enumerate them. If a ship is an object, you ought to be able to count ships. So, how many ships are involved in the story?

There are two named ships; Kero has labelled them in his boatyard. There are two recorded ships, documented in log books. But the named and recorded ships do not correspond. The recorded ship in Theseus' log book has been at different times both of the named ships. And you might say there are three paid-for ships in Kero's sales ledger. Theseus has paid Kero for two whole ships plus a ship's worth of parts.

Sometimes the model takes over: The ship's log is a kind of Enterprise Application model. In the end, you often go with the log book version of reality. The real world is just too fuzzy and complex to deal with. In abstract businesses like banking, the model is the business.

5.1 The fuzziness of the real world

The real world is a lot fuzzier than you might think from looking at the things around you.

5.1.1 Objects are not discrete in nature

Classes are not discrete in nature. The boundaries between so-called types are not at all clear. In biology, the apparently firm boundaries between biological species cannot be firm, otherwise evolution would be impossible.

Similarly, the class hierarchy above species (genus, phylum, etc.) is a highly subjective notion, with no firm basis in reality. It certainly does not correspond to the cladogram, the hierarchical structure that shows the forking paths of evolutionary history.

Nor are object instances discrete in nature. We believe ourselves to be individual members of the human race, but the discreteness we cling to is a kind of egoism, mainly to do with the continuity of our memory. Consider:

- A psychological curiosity: It has been shown that after a surgeon cuts the corpus callosum that connects the two halves of the brain, to relieve the symptoms of epilepsy, both sides of the brain think carry on thinking independently (though only one side may speak).
- An entymological curiosity: The queen of an insect colony gives birth to clones of herself.

Where is the individual in these cases?

The edge of an object in space is disputable. In cosmology, where is the boundary of the planet earth? At its surface? Or at the top of its atmosphere? Or at the end of the light travelling away from the earth since it was created?

5.1.2 Events are not discrete in nature

Events are as fuzzy as objects. The moment when an event happens is disputable. Consider

- A medical dilemma: Does a person die with the cessation of breathing? Or heart beat? Or brain activity?
- A cosmological dilemma: Was the earth born with the division of a large gas cloud into smaller ones? Or when this smaller gas cloud started to condense? Or when the mass of the earth stopped increasing?

5.1.3 Entity and events only become discrete in our models

To build a business rules model, we crystallise entities and events out of a world that is much more fluid, fuzzy and formless than our models imply. Where an object starts and ends in space and time is something we decide and define in building systems.

A doctor might distinguish between the classes disease and drug. A biologist: species and gene. A cosmologist: star and planet. A nuclear physicist: electron and photon.

These are not just different levels or partial views of the same model. They are entirely different perspectives.

Even in physics, the hardest of sciences, Einstein’s model of cosmological forces is to date irreconcilable with the model of quantum mechanics, though both are tested and accepted in their field. We always separate out the entities and events that best suit the model we are building.

5.2 The appearance of things

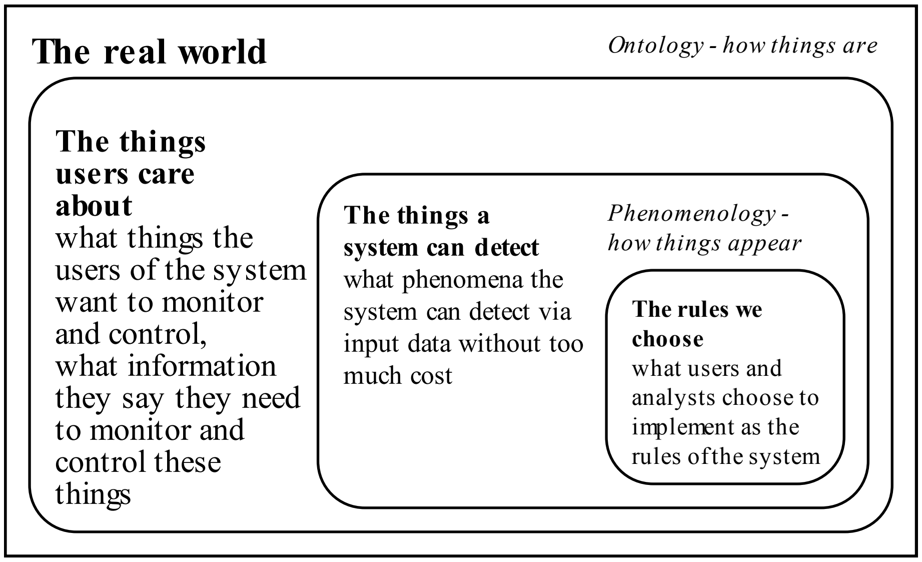
The problem of software engineering might be described as: How to discover, describe and connect the components of a system? In recent years, authors have stressed that the objects in a software system should somehow model or represent the things in the real world that the system seeks to monitor or control.

Many data modeling courses begin with the notion that entities are concrete things you can touch. One object-oriented author quotes Aristotle’s ideas on studying the substance of things. Some authors propose we should base our modeling work on ontology, the study of how things are.

This does not feel right. The entities or objects in our systems are not concrete things. They are only memories of things. And the things users want to remember are often highly abstract concepts such as dates, promises, and contracts.

In software engineering, we never build a model of how things are. Our models are highly subjective. Subjectivity enters at many levels. Our models reflect only the narrow business perspective of the system's owners and users. We model how things appear to these people. Then we must constrain our models even further to the appearance of things that can be detected by our systems.

The figure below shows the substance of the real world is filtered through several gauzes.


The real world
Ontology - how things are
The things users care about
what things the users of the system want to monitor and control, what information they say they need to monitor and control these things
The things a system can detect
what phenomena the system can detect via input data without too much cost
Phenomenology - how things appear
The rules we choose
what users and analysts choose to implement as the rules of the system

Michael Jackson suggested at one of our RAP group meetings that phenomenology (the study of how things appear) is probably more relevant to software engineering than ontology (the study of how things are). How things appear to a software system is limited to what input data it can read. What does this input data represent? It represents things happening in real world. It represents events. Events represent changes in that tiny, tiny portion of the world monitored by the system.

An event is a phenomenon by which a software system recognises a change in the real world (or a change in the user's perception of the world, which amounts to the same thing). Events are the phenomena by which a system recognises time passing.

5.3 Conclusions

We can take both entity-oriented and event-oriented views of system behavior. Event-orientation matters as much as object-orientation.

6. Behavioral terms, facts, constraints and derivations

The Business Rules Group have outlined a business rules classification scheme composed of terms, facts, constraints and derivations. This chapter introduces the behavioral variety of these rules.

6.1 Terminology

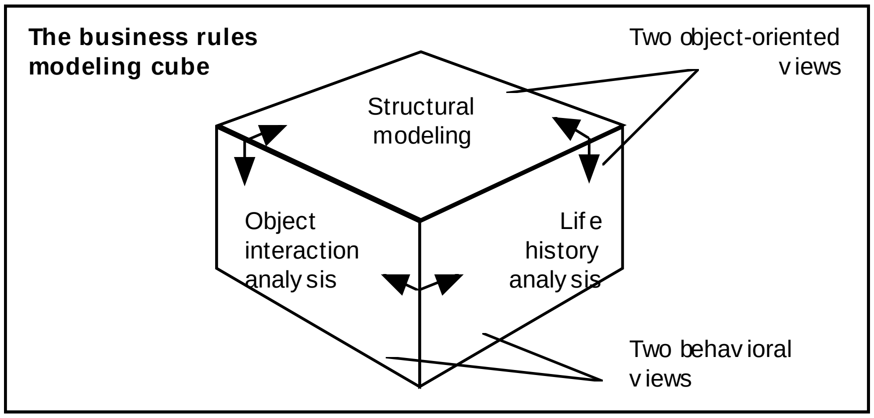
An event is a discrete, atomic, all-or-nothing happening. It updates one or more objects, and perhaps refers to the state of other objects. You might call it an atomic transaction. It is conceptual commit unit's worth of event effects.

An event carries data that represents something happening, or a decision being made. It happens in an instant and leaves its mark on, changes the state of, persistent objects. It triggers a discrete all-or-nothing process in the system being designed. It is a minimum unit of consistent change.

An event rules table or Interaction structure shows a short-running process, the pattern of objects affected by one event. Database readers: think of events as database transactions. J2EE readers: think of events as session beans.

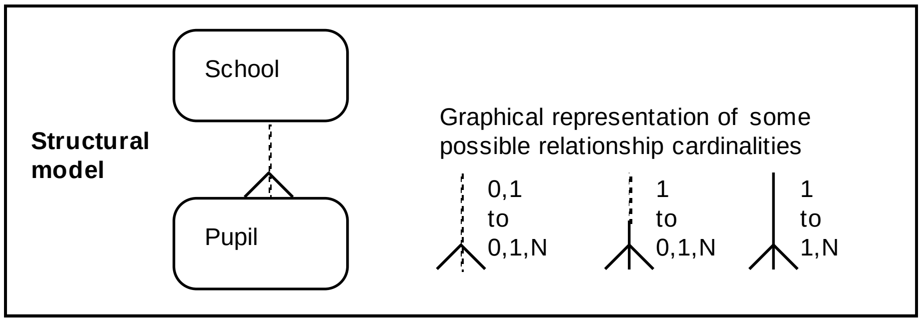
term	**definition**
behavioral term	the name of an event or operation. E.g. Wedding.
behavioral fact	relates behavioral terms and structural terms in a behavioral statement. E.g. Wedding events join Brides and Grooms.
behavioral constraint	a precondition that prevents an event from being accepted and processed. E.g. A Wedding event joins a Bride and Groom. Both Bride and Groom must be over 18 years old.
behavioral derivation	usually declares a side effect or post condition that an event leaves in its wake. E.g. MaritalStatus = married.

The RAP group uses a cube the show the three dimensions of business rules modeling.


The business rules modeling cube
Two object-oriented views
Structural modeling
Object interaction analysis
Life history analysis
Two behavioral views

6.2 The structural dimension

In structural analysis you define the classes of persistent information objects, and the persistent relationships between them. The figure below illustrates one style of structural model.


Structural model
School
Pupil
Graphical representation of some possible relationship cardinalities
0,1 to 0,1,N
1 to 0,1,N
1 to 1,N

Database designers draw entity relationship models. OO programmers draw class diagrams. I use the hybrid term 'entity model' in talking to mixed audiences about this view of a system.

All methodologies (from Information Engineering, through SSADM, to RUP) are 'data-driven' to the extent that structural modeling is the main graphical specification tool. But while the business rules modeling cube has one data-oriented view, it has two process-oriented views.

6.3 The two behavioral dimensions

In life history analysis you draw a entity state machine to specify the dynamic or behavioral view of a persistent information object. A entity state machine shows the events that may affect an object and the states it may pass through. An event may need to test the state variable in a precondition that can fail the event, or a control flow condition that selects between valid event effects.

In theory, you can model every class in the form of one or entity state machines. (Methods such as

Shlaer-Mellor, JSD and SSADM encourage this). In practice, people often find the step from entity model to entity state machines is difficult and obscure. There is a missing link.

Some would say the primary purpose of object interaction analysis is to define communication between objects. I do not. I say the primary purpose is to define the behavioral fact that an event appears concurrently in several entity state machines. An event's Interaction structure separates the modeling of concurrency (a feature of the problem domain) from the modeling of communication (a feature designed into the solution).

The matrix below is a picture that helps to show the wonderful symmetry between the columns (the entity-oriented view of behavior - the entity state machine diagrams of entities) and the rows (the event-oriented view of behavior - the Interaction structures of events).

6.3.1 An object event matrix

Persistent object ***Transient event***	**Pupil**	**School**
Pupil Registration	Create Pupil	
Pupil Enrolment	Tie Pupil to School	Gain Pupil
Pupil Transfer	Swap Schools	Lose Pupil / Gain Pupil
School Closure	Cut Pupil from School	Lose Pupil

You can complete the cells in the matrix with entries like create, update and delete, or as shown above, with more specific details. But for non-trivial systems, the entity-event matrix is inadequate. One event can have multiple effects on objects of one class, and trigger multiple operations within one effect. And the matrix is unmanageable in enterprise-scale systems.

6.4 How do we discover events?

You can build an entity-event matrix from the entity model by asking the following analysis questions:

- What events create and destroy objects in the system?
- What events connect and disconnect relationships between objects?
- What events update the attributes of objects?
- What constraints prevent each event from being processed?

Users must be confident that a system will perform correctly in terms of what preconditions apply to the processing of the events.

6.5 How do we model events?

You can document any non-trivial column in some kind of entity state machine diagram, and any non-trivial row in some kind of Interaction structure.

If you don't have a CASE tool to help you draw Interaction structures, and even if you do, you can document an event's business rules in a table. The table below shows the main terms and facts, constraints and derivations relevant to the Wedding event. In simple cases, the sequence of object access can be shown top to bottom.

EVENT: Wedding (Person [bride], Person [groom])		
Entities affected	**Preconditions: Fail unless…**	**Post conditions**

Person [bride]	Person exists
	Age > 18
	SexAtBirth = Female
	MaritalStatus = unmarried
		MaritalStatus = married

Person [groom]	Person exists
	Age > 18
	SexAtBirth = Male
	MaritalStatus = unmarried
		MaritalStatus = married

Marriage		Wife = Person [bride]
		Husband = Person [groom]
		MarriageDate = Today
		MarriageStatus = active

The table records constraints under the heading of preconditions, and derivations under the heading of post conditions.

If you model all the events in event rules tables, or all the entities in entity rules tables, then the entity-event matrix can be deduced or derived from that documentation. A CASE tool should be able to navigate from column to row by selecting an event name, and from row to column by selecting an entity name.

6.6 Conclusions

It is helpful and relatively straightforward to document the behavioral terms, facts, constraints and derivations using a mix of event rules tables and Interaction structures

Do I have to model every row of the entity event matrix, and every column? No. Each view does help to validate the other. It is very satisfying to fully specify both entity and event-oriented views of a system, and bring these views into perfect harmony. But in practice, you do not have to model the more trivial rows and columns. Concentrate on events that affect more than one entity, change a relationship or update a state variable. Concentrate on entities with more than one state (these are surprisingly common in Enterprise Applications, since many business entities 'die' some time before they are deleted).

7. Discrete event modeling

Using Interaction structures to model *association facts* rather than message passing.

The main point of this chapter is to separate the modeling of concurrency (a natural feature of the problem) from the modeling of communication (a designed feature of the solution).

This chapter is about drawing Interaction structures to specify how objects act in a coordinated way when an event occurs. It shows how to specify the route by which an information event discovers objects, without considering message passing. It introduces the concept of transient association facts.

The chapter goes some way to explain why and how business rules specification differs a little from object-oriented design. Interaction structures are not only platform independent, but also OO and procedural programming independent. Yet they make good specifications for both kinds of programming.

The object-oriented paradigm focuses attention on message passing, and it would seem that the primary purpose of drawing interaction structures is to define communication between objects. The business rules paradigm takes a different view. You should draw Interaction structures to specify the synchronisation of concurrent information objects, *before* making an decision about message passing.

This chapter presents interaction structures as a variety of interaction diagram that are useful as a formal problem-oriented modeling tool. Readers coming from a formal background may regard the chapter as philosophical rather than mathematical in spirit. However, the basic concept is mathematical, it is the notion of one-to-one association between the things and sets of things synchronously affected by an event.

7.1 Modeling the concurrency of objects

Almost every design notation or method is now based on what might grandly be called discrete object modeling. But while the business rules modeling cube has two entity-oriented dimensions, it also has an event-oriented dimension.

7.1.1 The need to model events

The objects in a system (be they graphical objects, information objects in a database, tasks in work flow modeling, or entity state machines in a process control system) have to be co-ordinated.

‘Our understanding of message routing tends toward the magical. Message routing problems are resolved often in a haphazard way at coding time.’ Palmer 1993.

‘The one-object-at-a-time view of system specification has its limitations.’ ‘No object stands alone; every object collaborates with other objects to achieve some behavior’ Booch 1994.

‘A near-universal short-coming’ of work flow modeling products lies in ‘managing rendezvous (or synchronisation) conditions in processes with parallel task threads.’ Ovum 1995.

7.1.2 What is missing?

OO analysis techniques involve ‘use cases’. But a use case is normally at a higher level of granularity; it is a package of events and enquiries designed to support a user in carrying out a task, or even a sequence of tasks. Use cases are rarely precisely or completely specified.

At the lower level, OO programming techniques include interaction diagrams for showing the messages passed between objects. Imagine how complex these diagrams become when you have to specify the assembly of a complex output data structure from many third normal form relations, or the two-phase commit in a database transaction.

The problem is not just one of complexity. Some of the object-oriented notations look friendly enough, but they are still based on the idea of message passing, which implies implementation choices have already been made to do with the programming environment and message-passing strategy. What is on offer are really coding notations, not problem modeling notations.

What is missing is what the late Keith Robinson used to call “discrete event modeling”.

7.2 Modeling real-world events

A real-world event may affect several objects, and so initiate collective behavior. The effects of the event are contemporaneous. This is true first in the real world, and then in the systems we build to model the real world.

‘If a pupil enrols in a school that is an event shared by the school and the pupil… participation is not sequential… the two aspects of the event are contemporaneous’ Michael Jackson.

In general, you cannot assume one real-world object becomes one information object (or one information object becomes one technology-level database table), but I do so in figures below, for the sake of simplicity.

An event requires associations (albeit transient) between information objects. These transient associations are facts. They may look like constraints, but to me they are inevitable facts of life.

7.3 One to one transient association

The tables below use 1:1 associations to show how the possible effects of the event on different real-world objects are related. The first table shows two objects are in 1:1 transient association with respect to the event.

EVENT: Pupil Enrolment

Entities affected		
Pupil	**<-->**	School

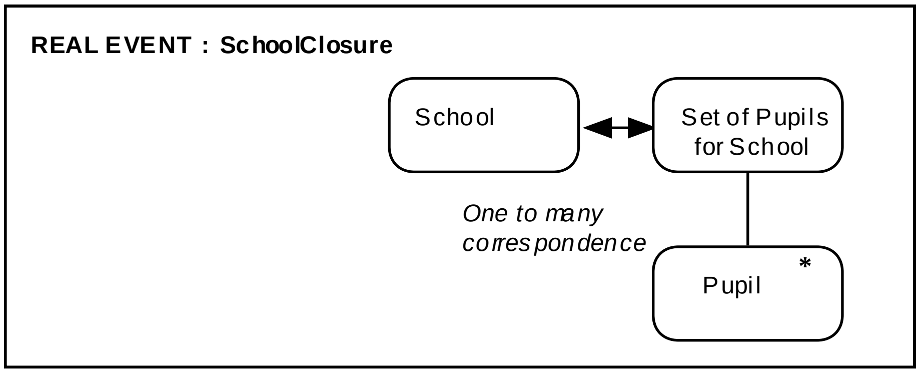
The second table shows three objects are in 1:1 transient association with respect to the event.

EVENT: Pupil Transfer				
Entities affected				
School [old]	**<-->**	Pupil	**<-->**	School [new]

Of course, not all objects will be in 1:1 association with respect to an event. However, you can always connect the objects using 1:1 associations by introducing selection and iteration components into the structure, as shown below.

7.3.1 One to many transient association

The diagram below (drawn using the old SSADMv4 notations for an effect correspondence diagram) shows two objects in one to many association with respect to the event. Note how the 1:1 transient association (here a two-headed arrow) is drawn to the set.

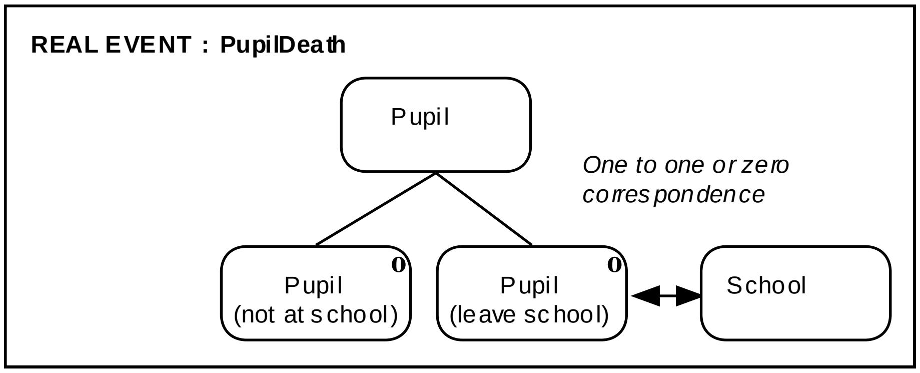


The diagram structure can be collapsed into a tabular form, using an asterisk to show the manyness of Pupils thus:

EVENT: School Closure		
Entities affected		
School	**<-->***	Pupil

7.3.2 One to one or zero transient association

The diagram below shows two objects in one to one-or-zero association with respect to the event. Note how the 1:1 transient association is drawn to one option of the selection.



A fact or control flow condition is needed to determine whether a pupil is currently enrolled in a school. This is condition is not constraint on, or precondition of, the Pupil Death event. Again, the diagram structure can be collapsed into a tabular form, using an **o** to show the optionality.

EVENT: Pupil Death (PupilSerialNum)			
Entities affected			
Pupil	**o--** (not at school)		
	o-- (leave school)	**<-->**	School

In practice, we do not build business rules models of real-world entities and events, we model information entities and events. These are a highly attenuated model of the real world. The attenuation from real world object to information object may not be so clear in embedded systems, where real-world objects are directly under the control of the system, but the attenuation is obvious in specifying Enterprise Applications. “I am more than a number!”

7.4 Modeling information events

Information events are the means by which a software system recognises objects in a real-world system, and detects changes in those objects. The job at the system level is to specify how persistent information objects are maintained by information events.

An information event initiates collective behavior. It advances the information objects in a system from one mutually consistent state to the next. An event is a minimum unit of consistent change to the information in a system. An event is a ‘logical commit unit’, meaning that if it fails in its effect on any one object, then it must fail in all objects.

Once you have accepted that you are specifying information event, not real-world events, it is natural to draw directed transient association arrows.

Given that the business rules model specifies information objects rather than real-world objects, you can and should specify transient association arrows as pointing in one direction.

What does the direction mean? It describes how the affected information objects are identified. See next section for more details. This direction is still ‘logical’; it does not imply any choice between technologies or decisions about physical design.

The table below separate transient one-way association arrows from higher and lower level concepts.

Level	**Interaction concepts**
Enterprise Model - Real-world events	Multi-way associations between entities
Enterprise Application Model - Information events	One-way association arrows between entities
Technology Model	Messages and/or foreign keys and indexes

A one-way directed arrow does not say that an event's effects in the real world are sequential. Nor does it prescribe choices in the machine domain to do with programming language, or message-passing strategy, or sequence of update processing.

Our systems should give the appearance that all the effects of an event are contemporaneous and co-ordinated. Whatever happens down at the technology level, the system users should believe that an event has a coherent and indivisible effect on the system. I'll talk later about different message-passing strategies you may employ at the technology-level, after looking more closely at the arrows.

7.4.1 How to draw the arrows in an event's Interaction structure

An arrow shows 1:1 association between the effects of the event at either end. The direction does not specify message passing; it specifies how the affected objects are identified. Sometimes, the system can identify all the persistent objects affected from the parameters supplied with an event. But generally, the system has to locate objects one after another, regardless of how you design the message passing.

You draw an arrow to an object from either the entry point, or from another object. An entry point arrow says the system can identify the affected object from the event parameters alone. An arrow from one object to another says the object at the tail of the arrow has to remember the identity of the object(s) at the head of the arrow.

7.4.2 The need to ask one object for the identifiers of other objects

A system must remember relationships as well as objects. You rely on one object remembering the identifiers of other objects, or somehow being able to find a memory of them.

7.4.2.1 An event will expect the current parent of a child to be remembered

The words 'parent' and 'child' often used by systems analysts to distinguish one end of a relationship from the other. Given a one-to-many relationship, the one end is the parent and the many end is the child.

Consider the Pupil Transfer event that swaps a Pupil from one School to another. The event identifies both the Pupil and the School [new]. These two objects may receive the event in parallel. But the event parameters do not identify the School [old]. This existing parent object is remembered by the system. The figure below shows this by a directed arrow.

EVENT: Pupil Transfer (PupilSerialNum, SchoolName [new])				
Entities affected			**Preconditions: Fail unless…**	**Post conditions**
Pupil				Pupil swapped from old school to new school
	-->	School [old]		Pupils = Pupils - 1
School [new]			School not full	Pupils = Pupils + 1

The main point is: the School [old] object cannot, in any reasonable implementation, receive the event until after the Pupil object. This precedence is neatly documented by drawing the transient association arrow as pointing in that direction.

The precise mechanism by which the system remembers the School [old] makes no difference to the event's Interaction structure, whether it is via a pointer chain, or a foreign key inside the persistent Pupil object, or some other mechanism.

By the way, does an event necessarily affect all the objects whose keys are present in its parameters?

No. Suppose the key of a Pupil is a hierarchical composite of School and Pupil, this does not mean that an event carrying the key of a Pupil will access the School first (though this might later be forced on a deeper level of programming by an implementation decision of the database designer). Some events hit only School or only Pupil, some will go from School to Pupil, and some (like the one above) will go from Pupil to School.

7.4.2.2 An event will expect the current children of a parent to be remembered

Suppose the event 'School Closure' affects all the Pupils in the School.

It is common in Enterprise Applications to broadcast an event to all the objects of a given type, all the child objects belonging to a given parent object. The event does not identify all the child objects. It expects the system to remember and locate all the children, given only the identity of the parent.

The child objects cannot, in any reasonable implementation, receive the event until after the parent object. The figure below neatly documents this sequence by drawing the transient association arrow as pointing from the parent to the set of child objects affected by the event, and showing the set as an iterated element.

EVENT: School Closure (SchoolName)		
Entities affected		
School	**-->***	Pupil

7.4.3 Conditions

There are two kinds of condition that might be annotated on an event's Interaction structure.

- A fact condition - a logic or guard condition that controls the entry to a selected option or iterated component in the control structure of the event's Interaction structure.

- A constraint condition - a precondition that stops the event from being processed or prevents the process from completing.

In the UML, these are easily confused, because discrete events are not distinguished from operations. The allocation of fact and constraint conditions is discussed in later chapters in this series.

7.5 Implementing an event's Interaction structure

The choice of coding style or language is not important during business rules analysis. Business rules modeling must be entirely separable from OO programming. But nobody wants to spend their time building models that are no use. It is important that people can transform an event's Interaction structure into program code. They key decision missing from an event's Interaction structure is the choice of message-passing strategy.

Consider in the Pupil Transfer event example, how does the Pupil Transfer operation in Pupil communicate to the Pupil Transfer operation in 'old' School, and what data is passed back and forth?

An event's Interaction structure documents the fact of an interface between objects, but not its data contents. You might assume each object passes on all the event data, and a copy the state of every object the event has passed through so far, but this is way over the top.

Three different ways to implement the interactions between objects are described below. The arrows in Interaction structures are not meant to be messages, but they do turn into messages under the second of these three strategies.

7.5.1 Chain or Staircase pattern: hand-to-hand message passing

In what might seem the most 'object-oriented' implementation, the objects pass the event from one to another, as though following the arrows in an event's Interaction structure. This is called the chain or staircase pattern.

Any event (in a process control system perhaps) that simply triggers objects into action is easily implemented following the staircase paradigm. Difficulties arise where you need to get data back from the objects and assemble this into a report. The staircase solution is not so good in Enterprise Applications where the event may have to build up a complex output data structure from the many concurrent information objects it affects.

7.5.2 Fork pattern: centrally-controlled message passing

In one possible object-oriented implementation, an event manager controls the whole event's Interaction structure. This solution is called the comb or fork because that's what it looks like in an Interaction structure that records the messages going back and forth from the event manager object.

The event manager controls something like a two-phase commit. First it calls each object with the event, then it reads all the objects' replies to check they are in the correct state, then it invokes each object again, telling it to process the event, update itself and reply with any required output.

Actually, it's more complex than this because the event manager will have to request some objects to provide it with the identities of others.

The difficulty with this approach is that by the time you've put all the control logic into the event manager, there is so little left for the individual objects to do that it seems barely worth invoking them to do it.

7.5.3 Procedure: combine the relevant parts of the objects into one

You can get around the need to define the message passing by extracting the relevant operations from each object, bringing them together into one procedure, and making them communicate via the local memory or working storage of that procedure.

This may seem strange to an OO programmer, but it is what procedural programmers do naturally. They code the event's Interaction structure as a single procedure, and implement it within one commit unit controlled by the database management system. If an event finds one object is in the wrong state, it tells the database management system, which rolls back any effects of the event on other objects that have been processed so far.

You might code the procedural solution in an OO programming language or another kind of language. Procedural languages like COBOL and declarative languages like SQL remain an effective means of implementing Interaction structures in Enterprise Applications.

7.6 Automated forward engineering

You can develop an entity model to the point where a CASE tool can generate most of the detail in the event models. There are CASE tools that can generate the boxes in an Interaction structures from the information recorded in entity state machine diagrams. The analyst is left to add the transient association arrows and the conditions on selections.

There is at least one CASE tool that can list and allocate actions to nodes of the event's Interaction structure. It takes some actions from the information recorded in entity state machines, and it invents and allocates 'read' and 'write' actions if the objects have to be stored in and retrieved from a database.

Given an event's Interaction structure you can code the event processing in either procedural or OO-style. A CASE tool can convert the event's Interaction structure into the form of a structured procedure or 'action diagram'. To design the object-oriented version involves choosing the staircase or fork message passing strategy and adding the message passing invocations to the operations of each class.

7.7 Conclusions and remarks

An event's Interaction structure is a graphical representation of behavioral terms and facts

- It specifies business rules without design detail; it stands firm in the face of different message passing strategies, object identifiers and implementation languages,
- It primarily shows the synchronisation of objects, but may be annotated with preconditions, post conditions and implementation details.

The style of Interaction structures in this chapter has several interesting characteristics and some advantages over OO-style message-passing diagrams.

7.7.1 What without how

Interaction structures are simple, friendly, and technology-independent. They say what without how. They provide specification without implementation. They are not affected by the designer's choice of programming environment or message-passing strategy, because the arrows represent 1:1 associations rather than messages.

An event's Interaction structure does not commit you to any statement about communication. Messages are an implementation device. You may select between a number of viable message-passing strategies. You can choose to send messages along the paths specified in the event's Interaction structure, or another route. You might eventually code the arrows in an event's Interaction structure as message passing in C++ or a sequence of read actions in COBOL, but this is irrelevant at the stage of systems analysis.

Interaction structures provide a better problem-modeling tool than OO-style sequence diagrams. The more you use Interaction structures the more you realise that event-orientation is just as important in systems analysis and design as object-orientation.

7.7.2 Formal modeling of the problem domain

An event reflects a natural phenomenon. An event's Interaction structure is a good place to document the behavioral facts of an event. It specifies the interactions between concurrent objects in a formal way. It is directed graph that tells you which objects are affected, the order they can be discovered in, and how one object naturally governs the route of the event to other objects.

7.7.3 Explicit and implicit relationships

An implicit relationship is implied by two or more explicit relationships. E.g. If a mother is explicitly connected by relationships to two children, a brother and sister, then the two children are implicitly related by a sibling relationship.

An event's Interaction structure is a directed graph; the event travels along a relationship in a one-way direction. But an arrow in an event's Interaction structure may follow an implicit relationship. On the other hand, physical message passing will have to follow explicit relationships if those are the only ones remembered by stored identifiers in the implemented system.

7.7.4 Concurrency without communication

An OO designer may consider the purpose of interaction analysis is to define the message passing between objects. I say the primary purpose is to define the behavioral fact that an event appears concurrently in several entity state machines. I believe the concurrency of interacting objects is more fundamental, more objective, than the communication that makes it work.

There is a body of theory about concurrency and communication (Hoare's Communicating Sequential Processes and Milner's Calculus of Concurrent Systems, among others). but this is seen as difficult and obscure. People are frightened by the abstract mathematical calculi that are used to explore the nature of communication and concurrency.

An event in the concurrency theories of Hoare and Milner is an abstract construct describing a condition that occurs as the result of concurrency, such as deadlock and non-determinism. An event in this chapter is *a cause* of concurrency rather than a result; it carries the semantics and business rules of an application.

An event's Interaction structure is not at all frightening. It helps us to separate the modeling of concurrency (a feature of the problem domain) from the modeling of communication (a feature designed into the solution).

Later chapters show how life history analysis helps you to specify classes as concurrent entity state machines and how object interaction analysis helps you to specify how these entity state machines interact when a discrete event, that is a minimum unit of consistent change occurs. Drawing an event's Interaction structure helps you to see and define the relationships between classes that are used in object interaction analysis.

7.7.5 Better CASE tools

You can develop entity state machine models to the point where a CASE tool can generate most of the detail in the event models. Three such CASE tools have been built. The only things you have to add by hand are the transient association arrows and the conditions governing selections and iterations.

This is a boon. Its value is quality assurance and configuration management. It also has a productivity benefit. But I do not advise anybody to develop a complete entity model then attempt to generate the event models. It is much better to develop the entity and event models in parallel. So in practice you will probably run the automated generator several times.

Automatic code generation from Interaction structures is an exciting area for CASE tool development. Again, the three message-passing strategies provide three different ways for the tool do this.

7.7.6 Modeling distributed systems

The style of Interaction structures show here could prove valuable to those who wish to combine federated systems or partition a single one into distributed business components. See the book "The Enterprise Modeler"

8. The seven basic event interaction patterns

You can specify an event using one or both of two tools: event rules table and event Interaction structure. In simple cases (and many cases are simple) it is possible to sketch the Interaction structure within the event rules table. This short chapter introduces seven patterns that form the basic building blocks of event specification. The figure below shows generic patterns drawn in the form of event rules tables with directed transient association arrows. Each pattern is a shape that you can reuse over and over in specifying different events in different business rules models.

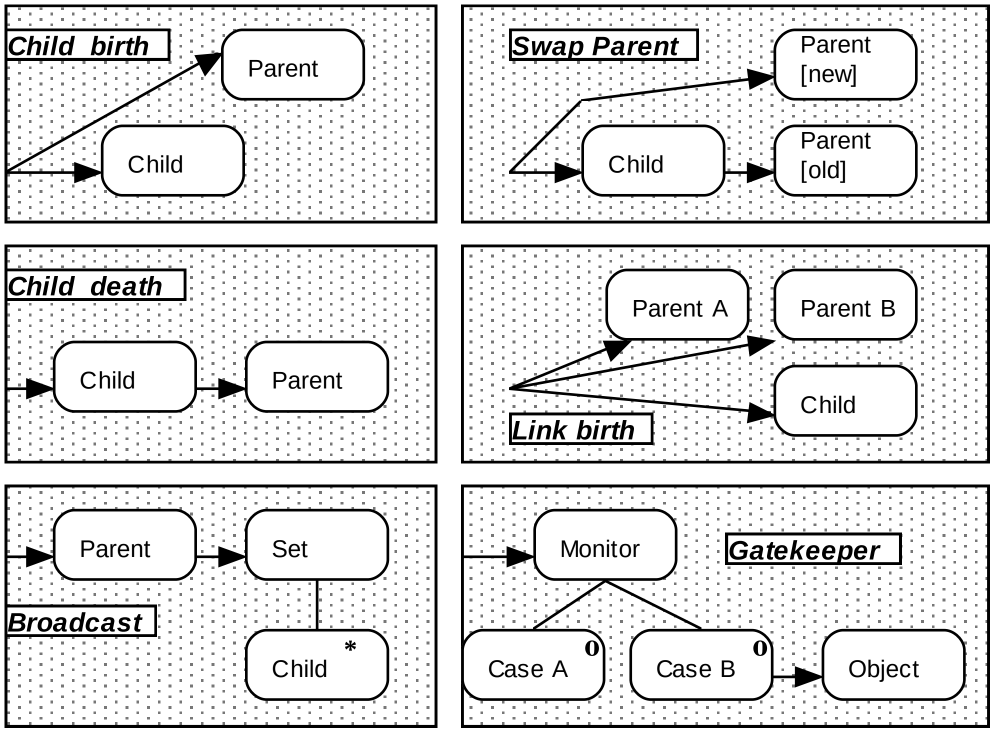
8.1 Patterns in tabular form

The table below shows the basic patterns.

SEVEN EVENT INTERACTION STRUCTURE PATTERNS		
EVENT: <<Child Birth >>		
Pupil		
Child		
EVENT: <<Child Death >>		
Child	--->	Parent
EVENT: <<Link Birth >>		
Parent A		
Parent B		
Child		
EVENT: <<Link Death >>		
Child	--->	Parent A
	--->	Parent B
EVENT: <<Swap Parent >>		
Pupil	--->	Parent [old]
Parent [new]		
EVENT: <<Broadcast >>		
Parent	--->*	Child
EVENT: <<Gatekeeper>>		
Monitor	o-->	Object

8.2 Patterns in diagram form

The same Interaction structures can be drawn using the SSADMv4.2 notation for an effect correspondence diagram.



8.3 Some examples

The three events below exactly fit the <<pattern name>> shown.

EVENT: OrderItemCreate <<link birth>>		
Entities affected		
Order		
Product		
OrderItem		
EVENT: Divorce <<link death>>		
Entities affected		
Marriage	--->	Person (wife)
	--->	Person (husband)
EVENT: Product Withdrawal <<broadcast>>		

Entities affected		
Product	--->*	OrderItem

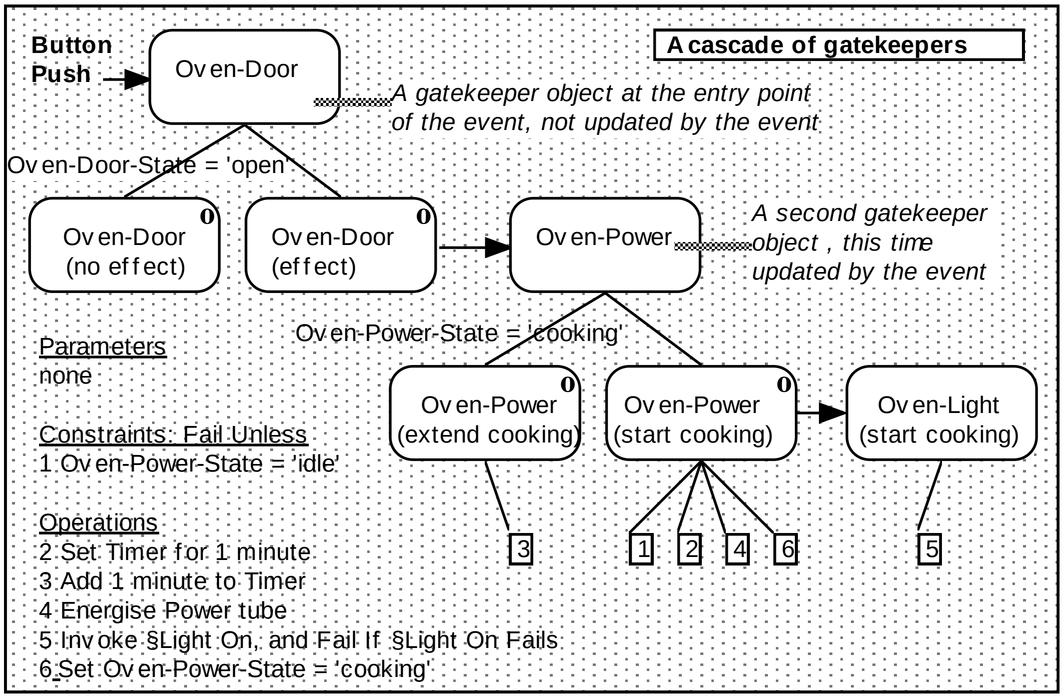
The Order Closure event is more complex, but you can see it includes the broadcast pattern.

EVENT: Order Closure				
Entities affected				
Order	--->	Customer		
	--->*	Order Item	--->	Product

8.4 A more complex pattern – gatekeeper cascade

A gatekeeper prevents an event instance from reaching all the object types or instances in the event's Interaction structure. You might say a gatekeeper filters an event. (I believe a gatekeeper is called a "context filter" in Jackson System Design).

e.g. The figure below shows an event's Interaction structure from my reworking of an old Shlaer-Mellor case study. It shows the possible effects of a Button Push event on the Oven-Power and the Oven-Light objects.



Button Push
Oven-Door
A cascade of gatekeepers
A gatekeeper object at the entry point of the event, not updated by the event
Oven-Door-State = 'open'
Oven-Door (no effect)
o
Oven-Door (effect)
o
Oven-Power
A second gatekeeper object , this time updated by the event
Oven-Power-State = 'cooking'
Oven-Power (extend cooking)
o
Oven-Power (start cooking)
o
Oven-Light (start cooking)
3
1
2
4
6
5
Parameters
none
Constraints: Fail Unless
1 Oven-Power-State = 'idle'
Operations
2 Set Timer for 1 minute
3 Add 1 minute to Timer
4 Energise Power tube
5 Invoke §Light On, and Fail If §Light On Fails
6 Set Oven-Power-State = 'cooking'

Notice, the Oven-Power object does not hear of the Button Push if the door is open. Similarly the Oven-Light object does not hear of the Button Push if the door is open, or if the power is already on.

The Oven-Power knows, by inspecting its state variable (cooking or idle), which of two optional effects the event will have (start cooking or extend cooking). The Oven-Light is only affected in one case (start cooking).

8.5 Patterns as a tool for analysis and design

Patterns make the work of the teacher easier; they shorten the learning curve. The teacher can illustrate the patterns via case studies, and teach how to use them as an analysis and design tool. I am especially interested in how patterns prompt you to ask important business analysis questions and so refine the specification. I call these refinements "generative pattern transformations".

8.6 Occam's razor in the gatekeeper pattern

Occam's razor tells us to cut out needless dross, to prefer the simpler of two possible explanations. This is useful as a general principle of system design. The gatekeeper pattern gives us chances to apply Occam's razor in the form of a more specific principle.

Guideline: "Do not allow two objects to duplicate the role of gatekeeper"

In other words, two objects should not maintain what is in effect the same state variable. (Or in the terms of Jackson's structured programming method, you should resolve 'boundary clashes' wherever possible.)

e.g. It would be crazy to specify the Oven-Light as receiving the event even if the power is on, making it repeat the same test (cooking or idle) to choose between effects, and then ignoring one of the cases. This would mean the Oven-Light object has to maintain what is in effect the same state variable as that of Oven-Power.

So, Oven-Power *has* to act as gatekeeper for Oven-Light. Again, this precedence is neatly documented by drawing the 1:1 transient association arrow as pointing from an optional effect under Oven-Power to Oven-Light.

Using this principle, I have reworked old case studies by Jackson and by Shlaer. I find that introducing gatekeeper objects and applying the principle above helps us to produce more elegant solutions in which objects filter events for each other. The entity state machines reduce to what intuition suggests is the correct and minimal specification. And so, the resulting code is shorter, smaller.

You should not duplicate event control flow in different entities. Where an object chooses (by testing its state) which of two or more effects an event may have, no other object should have to make the same test. If it needs to know, it should learn from the first object.

8.6.1 Gatekeeper as a generative pattern

The gatekeeper is a generative pattern. It prompts the following analysis question.

- Q) Given a gatekeeper at the entry point of an event: Does the gatekeeper object choose between effects by testing the event's parameters or the object's state?

If the former, then you should divide the event into two different classes of event, drawing a distinct Interaction structure for each. If the latter, keep the selection in the Interaction structure.

8.7 Conclusions and remarks

This short chapter has introduced six patterns that form the basic building blocks of events' Interaction structures. Extremely complex Interaction structures can be constructed by assembling the basic building blocks into the shape that meets the requirements of the business at hand.

Even the simplest Interaction structure pattern can prompt you to ask questions in life history analysis, and enable you to uncover more exactly what the end-users' requirements are.

I have not looked yet for more complex patterns in Interaction structures. But I have found scores more 'analysis patterns' in entity models and in entity state machines.

Another of our projects has compared and contrasted analysis patterns with design patterns (after Gamman et al.). It turns out that the differences are as instructive as the similarities.

9. Behavioral constraints

Specifying constraints during interaction analysis.

Terms and facts are fundamental. But you can't do much without the constraints; this is where all the useful stuff is. Analysts often neglect the constraints under which the enterprise operates. Frequently, required constraints are not articulated until it is time for programmers to code them.

'One must be careful to define an event clearly - in particular what the **initial conditions** and the final conditions are'. Richard Feynman writing on quantum electro dynamics

This chapter discusses constraints and illustrates the specification of behavioral constraints in Interaction structures.

9.1 Constraints as event preconditions

A behavioral constraint is a precondition that prevents an event from being accepted and processed. E.g. A Wedding event joins a Bride and Groom. Both Bride and Groom must be over 18 years old.

A constraint is a precondition that prevents a system from accepting or containing information that breaks the business rules. Most constraints take the form: Fail event E unless object O is in a valid state for event E. I record constraints under the heading of preconditions. Consider for example the constraints on a Divorce event.

EVENT: Divorce (MarriageNum)			
Entities affected			**Preconditions: Fail unless…**
Marriage			MarriageStatus = active
	--->	Person [husband]	MaritalStatus = married
	--->	Person [wife]	MaritalStatus = married

An interesting question arises here. Given that the Marriage event has been specified to set the husband and wife's sate variables to 'married', do we need to test this on the Divorce event? I would say no, unless this is a safety-critical system and detecting every possible bug is important.

9.2 Does a constraint belong to an object or an event?

Logically, it belongs to both, to the effect of an event on an object. Physically, it may be coded with either. A constraint applies at the intersection of a transient event with a persistent object. A constraint belongs to the effect of an event on an object. Each operation on an object is only meaningful as part of an event, and you need to think about the preconditions and post-conditions of the whole event. You should analyse and specify constraints from both Entity and event-oriented points of view.

Using an object-oriented programming language, you might code a constraint in an operation of a class. Using a client-server programming language, you might code a constraint in a transaction procedure. This choice is more to do with implementation technology than with the logic of the business rules being implemented.

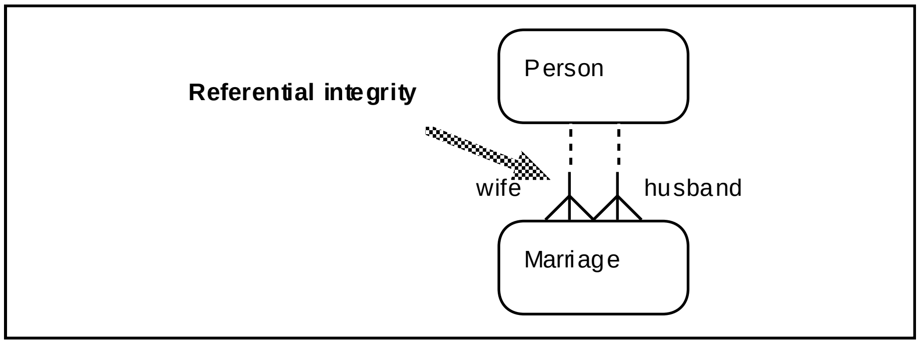
There are many and various ways to specify and code constraints; illustrated below.

9.3 Referential integrity constraints

A referential integrity constraint is a rule about the existence of relationships between objects before or after an event happens. You can always specify a referential integrity constraint as a precondition of one or more events. E.g. a Wedding event will:

- Fail unless Wife exists
- Fail unless Husband exists

Alternatively, you can ask a database management system to automatically impose referential integrity constraints by checking for the existence of records in the database. The figure below shows you might define such constraints on an entity model.



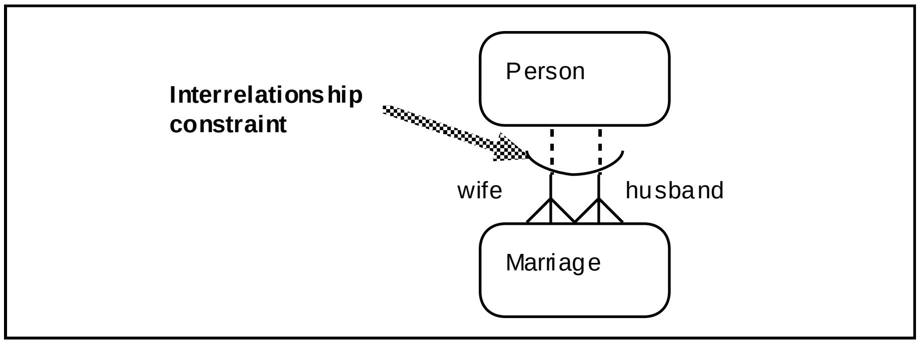
Most people would expect to code such constraints on relationships in the form dictated by their database management system, so that it will automatically impose referential integrity tests. But this approach has the severe limitations discussed in the RAP group papers on <Architecture definition>.

9.4 Inter-relationship constraints

An inter-relationship constraint specifies a mutual exclusion between relationships. You can always specify an inter-relationship constraint as a precondition of one or more events. E.g. a Wedding event will:

- Fail unless Wife SexAtBirth = Female
- Fail unless Husband SexAtBirth = Male

Alternatively, you can specify such a constraint in terms of mutual exclusion between relationships. The figure below shows a Person can relate to a Marriage as either husband or wife, but not both.



Most database management systems do not understand the exclusion arc, so you will have to code this constraint in another way, probably as attribute value constraints, which are discussed below.

9.5 Relationship multiplicity constraints

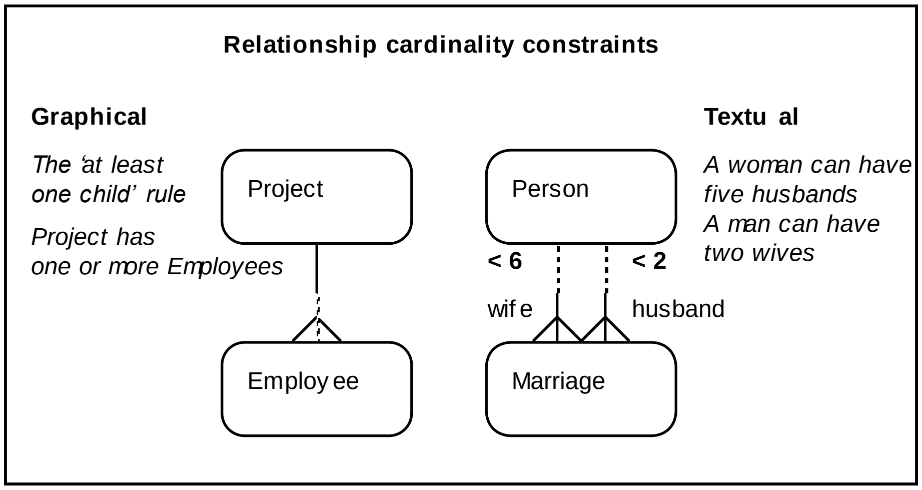
A relationship multiplicity constraint specifies the number of relationship instances that an object is allowed. You can always specify a relationship multiplicity constraint as a precondition of one or more events. E.g.

EVENT: Project Start	
Entities affected	**Preconditions: Fail unless…**
Project	Project has at least one Employee

Suppose you adapt the marriage registration system for a bizarre polygamous society:

- A man can have only one wife
- A woman can have five husbands.

You might specify such multiplicity constraints on an entity model.


Relationship cardinality constraints
Graphical
The 'at least one child' rule
Project has one or more Employees
Project
Employee
Textual
A woman can have five husbands
A man can have two wives
Person
< 6
< 2
wife
husband
Marriage

Since most database management systems understand neither the ‘at least one child rule’ nor numbers written on relationships, you will have to code these multiplicity constraints in some other way, probably as attribute value constraints. Often you will test the value of a total held in the parent class.

However, it can be difficult to specify the ‘on event’ nature of some constraints in an entity model. Constraints defined as relationship multiplicity constraints apply to every event, whether you want them to or not.

9.6 Attribute value constraints

An attribute value constraint specifies a restriction on data values. You can always specify an attribute value constraint as a precondition of one or more events.

EVENT: Order Closure	
Entities affected	**Preconditions: Fail unless…**
Order	OrderValue > $100
Customer	OrderValue + CustomerDebt < CustomerCreditLimit TotalUnpaidOrders < 5

Alternatively, you can specify such a constraint in a data dictionary by declaring the valid range of an attribute. However, it can be difficult in a data dictionary to specify the ‘on event’ nature of some constraints.

9.7 Inter-attribute constraints

An inter-attribute constraint checks two or more attributes are compatible. A class attribute may be compared with an event parameter. E.g.

EVENT: Promotion	
Entities affected	**Preconditions: Fail unless…**
Employee	PromotionGrade [input] > EmployeeGrade

An attribute of one object may be compared with an attribute of another object. E.g. the Order Closure event above will fail unless OrderValue + CustomerDebt < CustomerCreditLimit.

You can always specify an inter-attribute constraint as a precondition of one or more events. You do sometimes have to make a relatively arbitrary decision about which object will apply the constraint in the course of the event’s interaction. Further research may reveal useful heuristics in this area.

9.8 State variable constraints

A state variable constraint checks value of a state variable. E.g. a Wedding event will:

- Fail unless bride’s MaritalStatus = ‘unmarried’
- Fail unless groom’s MaritalStatus = ‘unmarried’

You can always specify a state constraint as a precondition of one or more events. Any constraint you think of in the form ‘Fail event A unless event B has already happened’ is naturally coded by testing the value of a state variable.

There is little alternative here. It is normal to detect an out-of-sequence event by testing a state variable value. They will ensure that the previous event in the entity state machine of the husband and wife must have been an event that set the state variable to ‘unmarried’ (presumably Birth or Divorce).

In fact, most constraints can be turned into tests on state variable values if you think hard enough, especially referential integrity and other constraints on relationships between objects.

9.9 Date constraints

A date constraint specifies when something should happen, before or after a specific date or time. You can always specify a date constraint as a precondition of one or more events. E.g. a Wedding event will:

- Fail unless Wife Age > 18 years
- Fail unless Husband Age > 18 years

Alternatively, you might code a date constraint as an inter-attribute constraint by comparing the value of an input date or stored date with today’s date. Or you might code a date constraint as a state variable value constraint. There are two events involved: the first event, a date, puts the object into the state where the second event, the constrained event, is allowed.

9.10 Conclusion

This chapter has discussed constraints and illustrates the specification of behavioral constraints in event rules tables.

10. Behavioral derivations

Specifying derivations during interaction analysis

Derivations define how knowledge in one form may be transformed by calculation into other knowledge, possibly in a different form.

‘One must be careful to define an event clearly - in particular what the initial conditions and the **final conditions** are’. Richard Feynman writing on quantum electro dynamics

This chapter illustrates the specification of behavioral derivations in event rules tables under the heading of post conditions.

10.1 Structural derivations

A structural derivation describes how one piece of data derives from other data. A structural derivation must be true at all times.

The Marriage Registration case study in the previous chapter contains a simple derivation.

- PersonAge equals the difference between DateToday and DateOfBirth.

And more elaborate rules describe how PersonCondition is derived for display on a marriage certificate.

- PersonCondition = Bachelor provided that Marital Status = Unmarried and SexAtBirth = Male.
- PersonCondition = Spinster provided that Marital Status = Unmarried and SexAtBirth = Female.

The Marriage Registration case study is no use for illustrating calculations. So I turn below to an order processing case study used in other RAP chapters

10.2 Behavioral derivations

A behavioral derivation usually declares a side effect or post condition that an event leaves in its wake. E.g. MaritalStatus = married.

A behavioral derivation is not true at all times, only just after a specific event has happened. In our order processing case study, an Order Closure event fires several derivations. These are only guaranteed to be true just after that event has been completely processed.

The figure below shows how these derivations can be specified as actions in the event rules table for Order Closure.

EVENT: Order Closure (Order num)					
Entities affected					**Post conditions**
Order					OrderValue = SumValue – CustomerDiscount AmountDue = OrderValue OrderClosure Date = Today OrderState = ‘Closed’ *Note 3*
	--->	**Customer**			CustomerDebt = that + OrderValue CustomerUnpaidOrders = that + 1 *Note 2*
	--->*	**Order Item**	o-- ItemQuantity = 0		No action
			o-- else		ItemValue = ItemQuantity * ProductPrice OrderItemState = Closed *Note 1*
			--->	**Product**	StockOnHand = that - ItemQuantity
Transient working data derivation					Sum Value = that + ItemValue *Note 4*

There is a minor and deliberate mistake in the example event rules table, discussed when I look at the behavioral facts of the event in a later chapter.

The illustration shows that derivations are readily specified with a behavioral event rather than an entity. You can and should support a structural model with behavioral models.

10.3 Allocating derivations to event effects

I aim to allocate each derivation to the effect of an event on an object. The general principle is to allocate a derivation to the effect on the object that owns the derived attribute. However, there are complications.

10.3.1 (note 1) Derivation of a child’s attribute from its parent’s data

ItemValue, an attribute of Order Item, is derived using data in its parent object, Product. The relevant action is placed in an operation of Order Item, the class that owns the derived attribute.

The model does not say how ProductPrice gets from Product to Order Item. This is a matter to be decided during design/coding rather than analysis/specification. In an object-oriented design, it would be via message passing.

10.3.2 (note 2) Derivation of a parent's attribute from its children's data

CustomerDebt and CustomerUpaidOrders are both derived using data gathered from a set of child objects. The relevant actions are placed in an operation of Customer, the class that owns the derived attributes.

10.3.3 (note 3) Derivation of an attribute from both parent and children

The derived attribute OrderValue in Order is derived from data from its children Order Items and its parent Customer. The relevant actions are placed in an operation of Order, the class that owns the derived attribute.

Notice again, the event rules table does not say how CustomerDiscount gets from Customer to Order. Again, this is a matter to be decided during design/coding rather than analysis/specification.

10.3.4 (note 4) Derivation of transient working data

Processing involves the maintenance of a transient attribute called SumValue. This is created and consumed within the process of the event. Who owns SumValue? You could argue it is a phantom (never stored) attribute of Customer and maintain it there, but it is surely best to regard it is an attribute of the event process itself – part of the session state if you like.

10.4 Post conditions are not all "derivations"

Some side effects of an event are derivations; some aren't. Consider for example the post conditions of a Pupil Transfer event.

EVENT: Pupil Transfer (PupilSerialNum, SchoolName [new])				
Entities affected			**Preconditions: Fail unless…**	**Post conditions**
Pupil				Pupil swapped from old school to new school
	--->	School [old]		Pupils = Pupils - 1
School [new]			School not full	Pupils = Pupils - 1

The rule "Pupil swapped from old school to new school" is an abstraction from implementation-specific detail. In a relational database it would mean replacing the value of a foreign key. In some other kind of implementation, it might mean updating indexes.

10.5 Conclusions

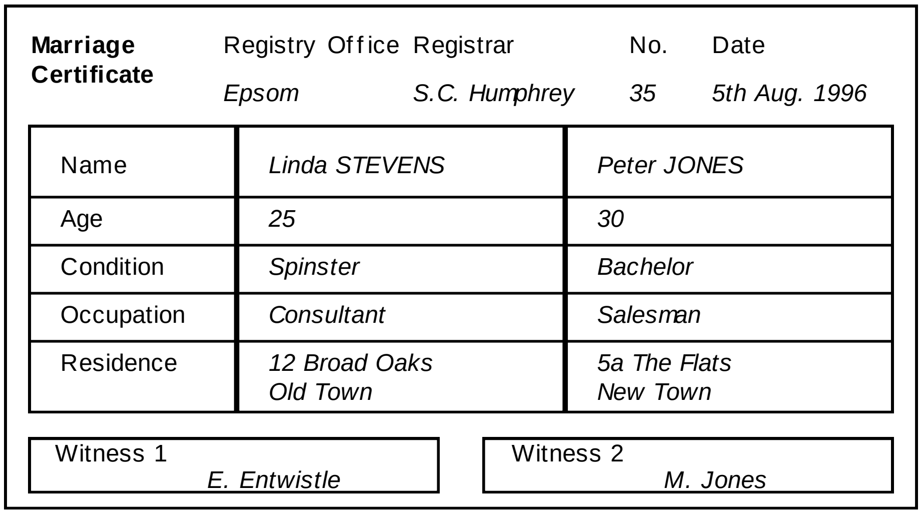
An event’s rule table or Interaction structure is a good place to document the behavioral facts of an event. The Interaction structure tells you the objects that are affected, the order they can be discovered in, and how one object naturally governs the route of the event to other objects.

This a good time also to document behavioral derivations. Most are readily specified in event rules tables. Later chapters develop the Order Closure example further, through examination of the constraints and control flow that governs the processing.

11. Marriage registration system case study

This chapter uses a small case study to illustrate the modeling of business rules, especially behavioral constraints.

Suppose the British government wants a system to register all marriages conducted under their administration. The system must also register everybody who is eligible for marriage. Let us start by prototyping a key part of the UI - the window for displaying the details of a marriage.



Marriage Certificate	Registry Office	Registrar	No.	Date
	Epsom	*S.C. Humphrey*	*35*	*5th Aug. 1996*

Name	*Linda STEVENS*	*Peter JONES*
Age	*25*	*30*
Condition	*Spinster*	*Bachelor*
Occupation	*Consultant*	*Salesman*
Residence	*12 Broad Oaks* *Old Town*	*5a The Flats* *New Town*

Witness 1 *E. Entwistle*	Witness 2 *M. Jones*

Reference and update modes? In operation of course, you don't want anybody who looks at the marriage record being able to overtype the details. So you might have two versions of the above screen, one for data entry and one for display only.

Primary keys? Oops, it looks from the example above that one real-world entity has been recorded as two information objects in the system. This is a common problem, and businesses usually try to minimise it by introducing some kind of business identifier; they uniquely label each real-world entity with a key or code. Note that businesses were assigning keys to real-world entities long before computers were invented. These keys are business identifiers, not the storage addresses of database records. You should not try to make business people use identifiers that are designed to help programmers locate data. Rather you should make programmers store the identifiers that business people use.

11.1 Terms and Facts

Structural terms in the case study include: Person, PersonNumber, PersonName, DateOfBirth, SexAtBirth, Marriage, MarriageDate, etc. Behavioral terms include: Birth, Death, Wedding and Divorce.

Structural facts in the case study include: Marriages relate Husbands and Wives. Behavioral facts in the case study include: Wedding events join Husbands to Wives. Divorce events separate Husbands from Wives. Behavioral facts are to do with how objects are co-ordinated by an event or enquiry; they are independent of how any program code works to achieve this.

Systems analysts have long been able to describe the structural and behavioral facts of an enterprise. They usually document terms and facts as classes, attributes, relationships, events and operations in diagrams and/or some kind of a CASE tool repository.

11.2 Constraints

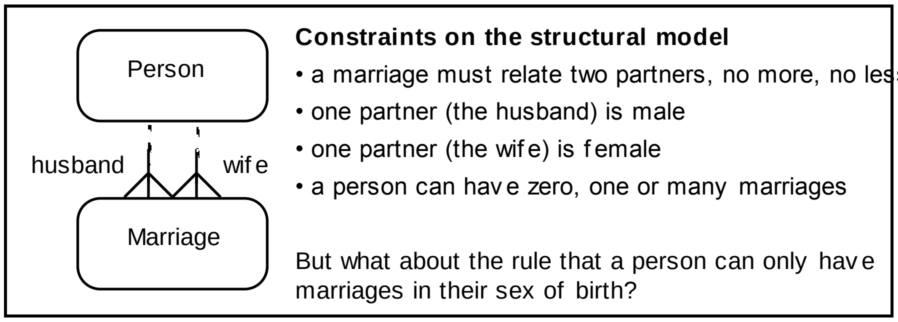
E.g. English law lays down a number of constraints governing a marriage. A marriage must relate two partners, no more, no less. One partner (the groom) is male. One partner (the bride) is female. Both partners must be over 18 years of age. A person can have zero, one or many marriages. A person can have only one marriage at a time. A person can only have marriages in their sex of birth.

Note the last of the points listed above. In the UK, a person can change sex to become a transsexual, but cannot contract a marriage in their new sex. This was established by the April Ashley case in 1970, and is currently under review. The fact that the rules may be changed in the future is a matter worth exploring, and such 'scheman evolution' is discussed in a later chapter in this series.

In reality, there are other preconditions to do with the notice period, the number of witnesses, the residential addresses of the partners, the location of the marriage, and so on, but we'll have to put them aside.

11.3 Constraints poorly specified in the structural model

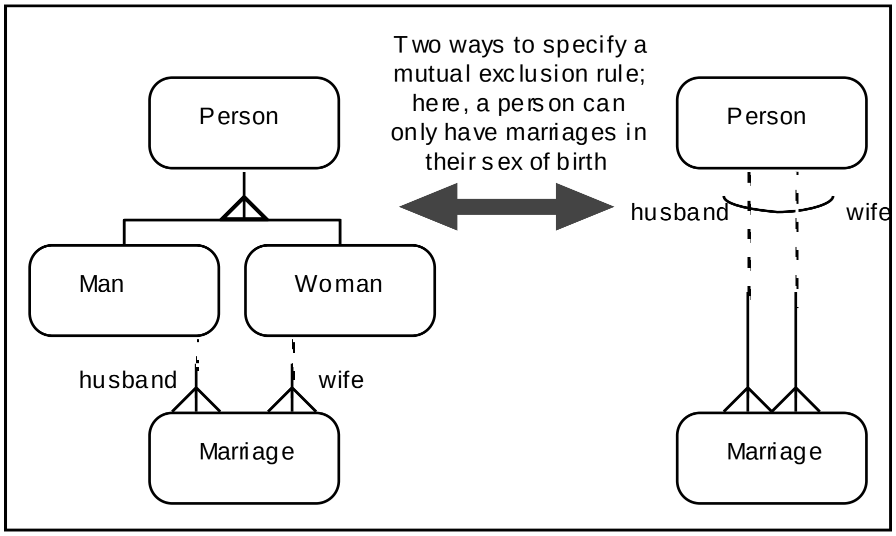
How to specify the constraints on a Wedding? . The figure below shows you may readily specify some rules in the form of a structural model.



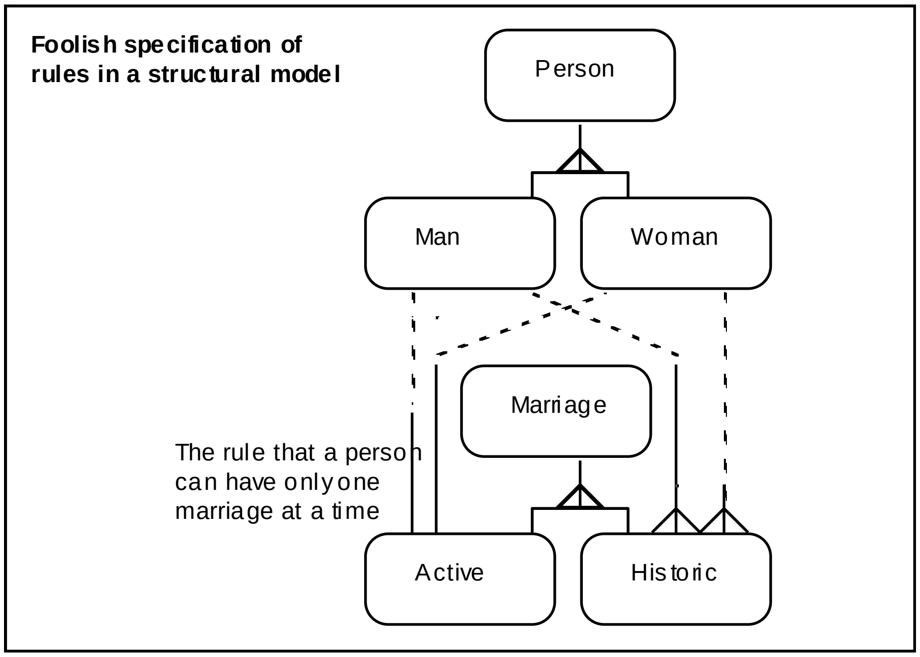
Constraints on the structural model
- a marriage must relate two partners, no more, no les
- one partner (the husband) is male
- one partner (the wife) is female
- a person can have zero, one or many marriages

But what about the rule that a person can only have marriages in their sex of birth?

How to specify the constraint that one Person cannot be related to Marriages in both husband and wife roles? The figure below shows two ways.


Two ways to specify a mutual exclusion rule; here, a person can only have marriages in their sex of birth
Person
Man
Woman
husband
wife
Marriage
Person
husband
wife
Marriage

This structural model still does not capture all the rules. What about the constraint that a Person can only have one Marriage at a time? The figure below shows you can extend the structural model to capture the rule of monogamy.


Foolish specification of rules in a structural model
Person
Man
Woman
Marriage
The rule that a person can have only one marriage at a time
Active
Historic

What about the constraint that both partners must be over 18 years of age? Stop! This way lies madness. It is a mistake to keep on extending the structural model until it shows all constraints.

"Although we could invent new graphic notations for further constraints, this could make the graphical language hard to learn, and lead to cluttered diagrams. Constraints which cannot be expressed on the diagram may be specified as textual constraints." 'Conceptual Schema' Halpin, page 231

This is true even of constraints that can be expressed (with some ingenuity) by adding subclasses into the diagram, such as the monogamy constraint on a Person's Marriages.

Many people try to treat all constraints as Invariant Constraint in an entity models. But many constraints are transient, dynamic or volatile, so it is not appropriate to build them routinely into structural models. You need ways to make all constraints explicit, not just Invariant Constraints.

11.3.1 Constraints well specified in the behavioral model

In general, constraints are assertions about the actions that are possible. You prevent a data item from being entered, or a relationship from being established, by preventing an event from taking place. So constraints can be expressed as event preconditions.

"While state-transition diagrams are useful for visualising constraints, formal rules for enforcing these are best specified in an Event Condition Action language." 'Conceptual Schema' Halpin, page 232

Our 'Event Condition Action language' is a variation of the event modeling language that was developed through extensive research by the UK government during the 1980s.

Every constraint is fired by an event. Most constraints apply to the intersection of a transient event object with a persistent information object. The figure below shows you can record each constraint in an operation fired by the event and acting on an object.

EVENT: Wedding (Person [bride], Person [groom])		
Entities affected	**Preconditions: Fail unless…**	**Post conditions**

Person [bride]	Person exists
	Age > 18
	SexAtBirth = Female
	MaritalStatus = unmarried
		MaritalStatus = married

Person [groom]	Person exists
	Age > 18
	SexAtBirth = Male
	MaritalStatus = unmarried
		MaritalStatus = married

Marriage		Wife = Person [bride]
		Husband = Person [groom]
		MarriageDate = Today
		MarriageStatus = active

Clearly, some constraints are readily specified with a behavioral event rather than with structural entities. You can and should support a structural model with behavioral models. Instead of the annotating the constraints on the diagram, you can record them in a table along with the diagram.

11.4 Derivations

Notice that the sex of a person is not shown directly as an attribute on the marriage record; it is implied by the conditions ‘bachelor’ and ‘spinster’. Assuming the system is to maintain historical information about people’s past marriages, you will need further windows for entering personal details and displaying people’s records. The figure below shows a list of people.

Person List

Name	Sex of birth	Date of birth	Occupation	Transsexual
Linda Stevens	F	8/10/71	Consultant	no
Peter Jones	M	1/1/66	Salesman	no
Peter Janes	M	1/1/66	Car Salesman	no

11.5 Conclusions

An Interaction structure is a good place to document the behavioral facts of an event. The diagram tells you the objects that are affected, the order they can be discovered in, and how one object naturally governs the route of the event to other objects. At the same time, you should document behavioral constraints. Most are readily specified in event rules tables.

12. More about constraints

This chapter says a little more about the specification of constraints, the preconditions that can cause an event to fail.

12.1 Constraints in event rules tables

I like to specify an event using one or both of two tools: event rules table and event Interaction structure. In simple cases (and many cases are simple) it is possible to squash the Interaction structure into the event rules table.

During object interaction analysis, you can specify every constraint in an event's Interaction structure. The figure below illustrates how you can write a constraint as a 'Fail unless' statement, and allocate it to an operation on the relevant entity in the event rule table.

EVENT: Divorce (MarriageNum)				
Entities affected			**Preconditions: Fail unless…**	**Post conditions**
Marriage			MarriageStatus = active	MarriageEndDate = Today
	--->	Person [bride]	MaritalStatus = married	MaritalStatus = unmarried
	--->	Person [groom]	MaritalStatus = married	MaritalStatus = unmarried

An event's Interaction structure specifies one event's effects on several objects. This kind of event rules table is a specification rather than an implementation; it says what is to be done rather than how it is done.

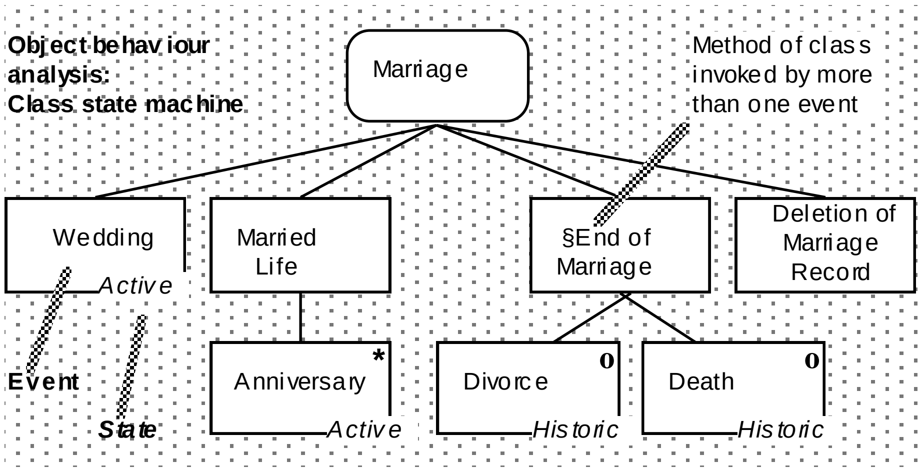
Chapter two discussed three ways to implement an event's Interaction structure. For object-oriented programming, you would extend the event's Interaction structure with implementation-specific detail to do with message passing. For event-oriented programming, you would base the programming on a read/write access path derived from the diagram.

Remember the difference between 'events' and 'operations'. A Divorce is an *event* that succeeds or fails as a whole. A Divorce event will trigger a number of lower-level elementary operations. If the event finds a necessary precondition is untrue, it will fail, backing out any operations done so far.

12.2 Constraints in state models

You may analyse the dynamic behavior of objects and describe the behavior of a class in the form of a state-transition diagram.

The figure below illustrates a structured form of state-transition diagram that imposes a regular expression (a hierarchical structure composed of sequence, selection and iteration components) over the event effects.



Drawing a hierarchical structure has some advantages; it makes it easier to tidy up the diagram and recognise standard patterns; and it naturally leaves space at the bottom of the diagram for annotation. By annotating the event effects with processing detail, you can specify the implementation details, all the processing operations, all the state transitions and all the constraints on event processing.

Some have assumed that *every* constraint can be defined graphically, in the shape of a entity state machine model, in terms of the permitted sequence of events. If this were true a CASE tool could detect all the constraints from the shape of the diagram alone, and generate all the relevant preconditions in code.

I have found through extensive research that it is possible to specify *most* constraints as sequences of events in large entity state machines. But without going into the detailed research, I have found that it is clumsy to specify certain kinds of constraints in the shape of a entity state machine diagram.

Given the various kinds of constraints listed in earlier, I find the following kinds are best listed as numbered constraints and allocated underneath event effects in entity state machine diagrams:

- Relationship multiplicity constraints
- Attribute value constraints
- Date constraints
- Inter-attribute constraints

This leaves us with several kinds of constraint that are readily specified in the *shape* of a entity state machine diagram:

- Referential integrity constraints
- Inter-relationship constraints (exclusion arcs)
- State variable value constraints

Broadly-speaking you can specify referential integrity rules in a entity state machine by showing the valid points at which the birth and death events of a child object can occur in the life histories of its parents. The main advantage is that you can get away from the restrictions of automated referential integrity. You can bend the rules so that full referential integrity is maintained on some events, but disregarded on others.

You may also specify ‘cascade’, ‘restrict’ and ‘no effect’ rules by showing how and where the death event of a parent appears in the life histories of its children. The main advantage is that you can apply these rules to logical death events as well as physical deletion events.

You can specify an inter-relationship constraint (an exclusion arc over relationships) as a high-level selection in a entity state machine. This often involves drawing a parallel aspect entity state machine for the purpose, as discussed in the chapters on ‘OO and business data’.

Last but not least, you can naturally specify all state variable value constraints in the shape of the entity state machine diagram.

12.3 Conclusions

The question arises: So what? Yes, I can specify some constraints in the shape of entity state machines, but I know I can specify all constraints as numbered statements allocated to the event effects in an event’s Interaction structure, so why bother with the entity state machines?

The answer is threefold.

- There are some problems that are difficult to grasp from only the event-oriented perspective. Looking at things event-by-event it is hard to visualise the state-transitions of an object and to validate that these state-transitions are sensible. This difficulty implies you need to formally analysis only the most complex of classes.
- The evidence suggests that people who make at least an attempt to analyse object behavior uncover more of the business rules than people who don’t. The earlier a rule is discovered the cheaper the costs of that discovering that rule. This benefit may be gained by carrying out a relatively informal analysis, concentrating on core business entities.
- If you have a good CASE tool (admittedly a big if), you can generate most of the details in the event rules tables automatically from the entity state machines (and perhaps vice-versa), and then generate code from the event rules tables. Analysing from both perspectives gives you a better chance to ‘get it right first time’ when it comes to implementation. This benefit can only be gained by a thorough and formal analysis.

In our opinion, training in life history analysis currently falls short of that required for people to achieve the last of these three benefits.

12.4 Bending the rules

The big advantage of specifying a constraint in the business rules layer or data storage layer is that the constraint is specified and coded only once. If the constraint changes, you only have to change one piece of code. You can design as many different on-line dialogues and off-line functions in the user interface layer as you like; you don’t have to specify any constraints within them, simply invoke events in the business rules layer. The constraint will always be applied, wherever or however the event is input.

There is however a disadvantage to implementing a constraint thus. The constraint is invariant. What the analyst at first thinks is a mandatory constraint to be applied to every case, may turn out to be optional in exceptional cases.

For example: Is it really true that every Project must have at least one Employee to be set up on the system? What if the users say that every now and then they do set up a Project without any Employees? There are a number of possible design strategies here. You could:

- say the odd Project must be handled outside of the system being designed
- insist the odd Project is registered as having a 'dummy' Employee it doesn't really have
- drop the 'at-least-one-child' constraint to handle exceptional cases like this.

The problem with the last approach is that the constraint is useful for the majority of cases. It seems a shame to throw away the constraint altogether. What you need is a way of implementing this as a business policy rather than a constraint.

There are no easy answers. You might transfer the constraint from the business rules layer into the user interface layer, and apply it there on some routes into the system, but not all. You might be able to develop an expert system for this more flexible kind of constraint.

I am not concerned with user interface layer constraints from now on. The remaining chapters show how you can specify event rules for the business rules layer.

12.5 Error handling

When an event fails a constraint test, the system must tell the user. You might begin designing a system under the assumption that all data is input correctly and constraints are automatically maintained. But sooner or later you must design how the system will detect contravention of these constraints, and respond to errors.

Error handling comes in three parts: error detection (see earlier chapters in this series), error reporting (see RAP group papers on Architecture definition), and error correction (see below).

12.5.1 Error correction

Despite all the best efforts of designers and users, some invalid events will be processed (for example, you might mistakenly identify the wrong person for input on a wedding event). The effects of processing mistaken events must be investigated and handled.

The effects of an error event are the same as effects of a valid event, except that they are mistaken, so the system will get out of step with the real world. The problems are:

- output data will be produced that is incorrect
- the stored data will be updated, but 'corrupted'
- future input data will be accepted or rejected, wrongly

Whenever an error report is produced, someone must investigate and do whatever is necessary to put things right. A mistaken event may trigger processing which is:

- beneficial: later proves to have been useful
- neutral: later proves to have been unimportant
- intolerable: has to be undone or handled by remedial action.

There are three things to do in handling intolerable side-effects:

12.5.2 Erasing the effect of mistaken outputs

It is hard to generalise about this. You must find out whether users:

- do not care about small errors in the output they receive
- will find the errors for themselves and handle them without further help
- will require some kind of ‘amendment notice’
- will require the output to be redone from scratch
- can be mollified by advance warning of possible error.

As an example of the last, consider the message often printed on reminder letters you receive, ‘if you have already paid this bill, please ignore it’.

12.5.3 Restoring stored data to the correct state

There are four ways to fix up the database:

- Reversal events
- Deliberate abuse of proper events
- Data fixing system
- Special fix-up transactions.

Robinson K. & Berrisford G. [1994] say a little more about these.

12.5.4 Reinput events/input records which have been rejected

Again Robinson K. & Berrisford G. [1994] say a little more about these.

13. Preconditions and control flow conditions

This chapter illustrates how an event's Interaction structure evolves as business rules change. The chapter distinguishes two kinds of condition that might be annotated on an event's Interaction structure.

- A fact condition - a logic or guard condition that controls the entry to a selected option or iterated component in the control structure of the event's Interaction structure.
- A constraint condition - a precondition that stops the event from being processed or prevents the process from completing.

The chapter illustrates how the balance between facts and constraints may shift as business rules change, and how discrete event modeling helps us to establish the right balance.

13.1 Rules that change

There are two ways to address the challenge of volatile rules. The first is to store the rule itself as an attribute value that can be updated by end users. E.g.

An entity model used to record invariant rules	
Entities and attributes	**Invariant rules:** these conditions always hold
Account	
Account Id	a system generated key
Account Balance	= a number > 0
Transfer	
Account Id [giver]	= Foreign key of a known Account,
Account Id [receiver]	= Foreign key of a known Account, not = giver
Transfer date	Not = Transfer the same day as any other Transfer
Amount given	= a number > 0
Amount received	= Transfer rule (see below)
Rule identifier	= Foreign key of a rule where Start date < Transfer date < End date
Transfer rule	
Rule identifier	A system generated key
Commission percentage	< 50
Transfer rule	= "Amount given * (100 – Commission percentage) / 100"
Commission receiver	= Foreign key of a known Account (not the giver or receiver)
Start date	= a date
End date	= a date > Start date

This first approach has its limitations. The second and more general approach is to define the rule as a transient pre or post condition of one or more events.

e.g. Another chapter features a marriage registration case study in which the laws regarding marriage are specified as Transient Constraints of the Wedding event rather than as invariant rules on the relationships between Person and Marriage entities in the structural model. The assumption is that when the marriage laws change it will be easier to recode and recompile the Wedding event process than to restructure the entity model, with all that implies for changes to the database schema and/or the data abstraction layer.

13.2 The process granularity issue

If you look at the various ways people code systems, a curious fact emerges.

- All constraint conditions can coded as control conditions inside procedures

It is probably obvious that you can specify every condition in a system as control flow conditions. You can specify the system entirely using procedural flowcharts. Every error/validation test can be specified and coded as a control flow condition within a procedure.

- All constraint conditions can be coded as preconditions of procedures

It may not be so obvious that you can specify every condition in a system as a precondition of a procedure. You can specify a system entirely in terms of atomic condition-less processes that only work under certain preconditions.

You do this by decomposing high-level procedures into smaller and smaller modules until there is no control structure left, until all algorithms have been broken into their elementary component processes, and all conditions are expressed as constraints.

So the question arises: How to strike the right balance between control flow conditions and preconditions in a system specification? What is the right level of granularity for specifying business rules?

13.3 Example version 1

The figure below is a fragment of the specification for the Order Closure event process in a simple order processing system.

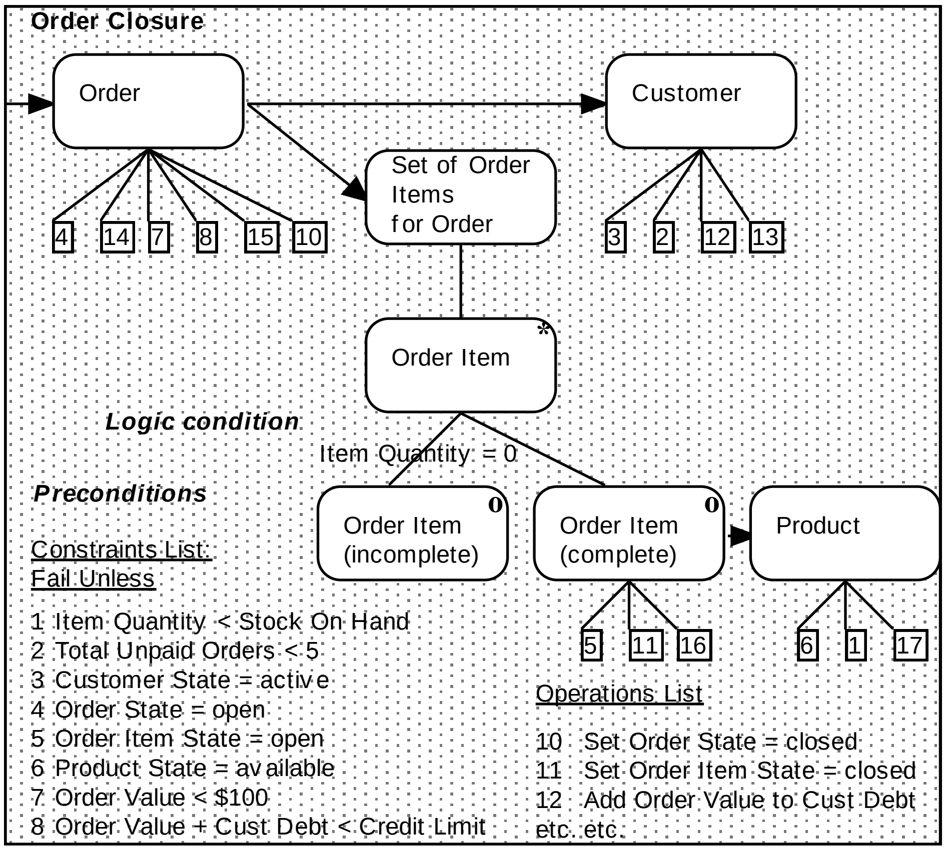
EVENT: Order Closure (Order num)		
Entities affected		**Post conditions**
Order		OrderValue = SumValue – CustomerDiscount
		AmountDue = OrderValue
		OrderClosure Date = Today
		OrderState = 'Closed'
	---> Customer	CustomerDebt = that + OrderValue
		CustomerUnpaidOrders = that + 1

	--->* Order Item	o-- ItemQuantity = 0	No action

		o-- else	ItemValue = ItemQuantity * ProductPrice
			OrderItemState = Closed

		---> Product	StockOnHand = that - ItemQuantity
Transient working data derivation			Sum Value = that + ItemValue

The informal specification above is reasonable, but not very precise, and you could not generate code from it. The event's Interaction structure below is more formal; it documents the algorithmic control structure that governs the control flow of an Order Closure event's effects on the objects in an order processing system. It also documents the constraints. If any one of the constraints is not satisfied, then the whole Order Closure event (not just an operation on one object) must be rolled back as though it never happened.


Order Closure
Order
Customer
Set of Order Items for Order
4
14
7
8
15
10
3
2
12
13
Order Item *
Logic condition
Item Quantity = 0
Preconditions
Order Item (incomplete) o
Order Item (complete) o
Product
Constraints List:
Fail Unless
5
11
16
6
1
17
1 Item Quantity < Stock On Hand
2 Total Unpaid Orders < 5
3 Customer State = active
4 Order State = open
5 Order Item State = open
6 Product State = available
7 Order Value < $100
8 Order Value + Cust Debt < Credit Limit
Operations List
10 Set Order State = closed
11 Set Order Item State = closed
12 Add Order Value to Cust Debt
etc. etc.

Other chapters in this series answer the questions. What do the arrows mean? How does the CustomerDiscount get from the Customer object to the Order object, where it is needed for a calculation operation?

13.3.1 A generative pattern in version 1

There is a generative pattern in the first version of the event's Interaction structure, an iterated selection where one of the options has no processing beneath it. A generative pattern is a shape to look out for because it prompts an analysis question and a possible transformation.

- Ask of an iterated selection where one of the options has no actions beneath it: Does the option belong in the processing?

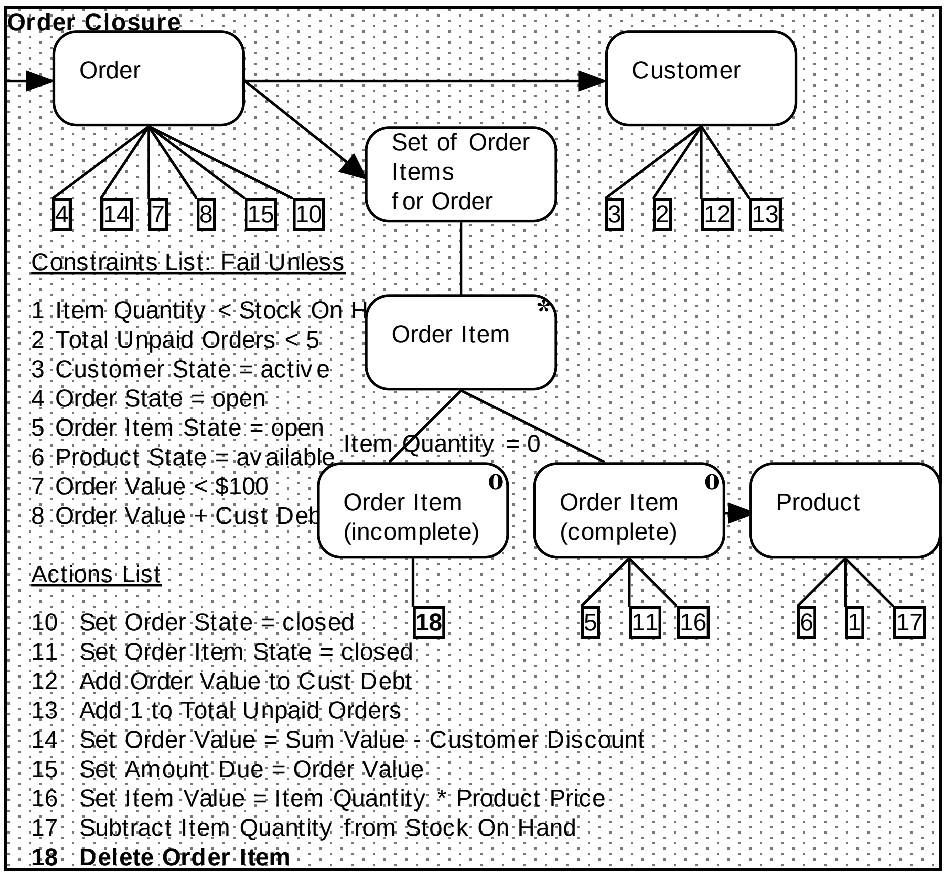
In this case the generative pattern prompts the question: What happens to an Order Item with an item quantity of zero? What should we do with Order Items that are incomplete when the Order is closed?

Of course, one might choose to prevent any Order Item from being entered with a zero quantity. But I am going to pursue the evolution of the Order Closure event's Interaction structure through three business rule changes.

13.4 Example version 2

Rule change: delete any incomplete Order Items on an Order Closure event.

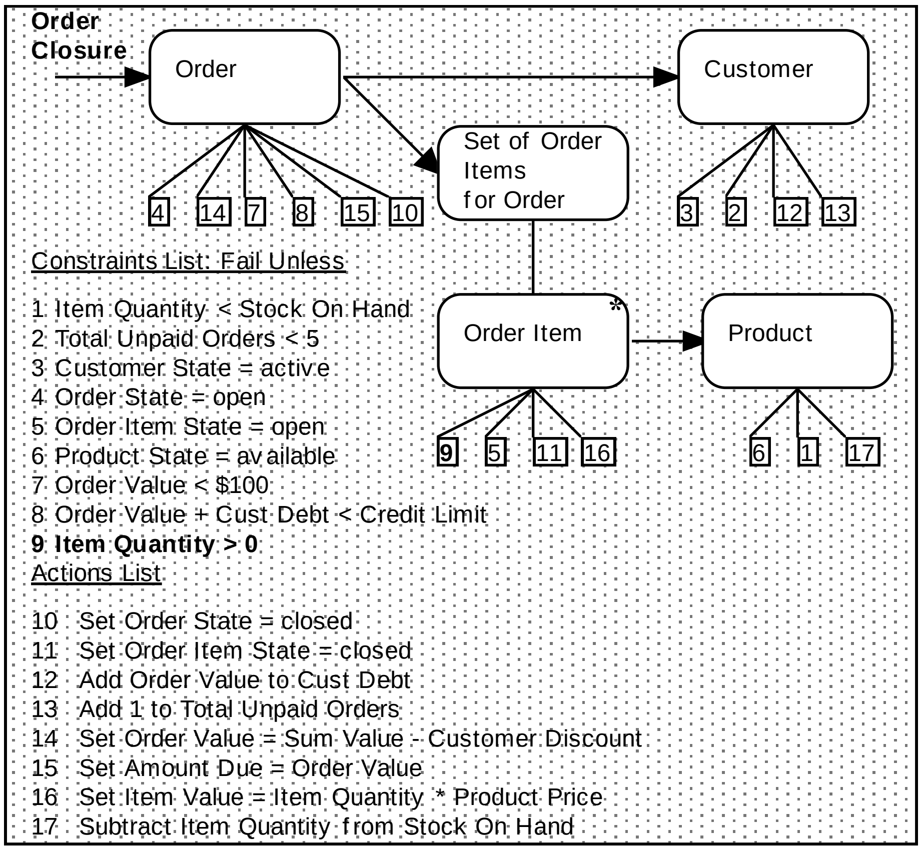
The figure below shows you can easily extend the event's Interaction structure with an extra action (18).


Order Closure
Order
Customer
Set of Order Items for Order
4
14
7
8
15
10
3
2
12
13
Constraints List: Fail Unless
1 Item Quantity < Stock On H
2 Total Unpaid Orders < 5
3 Customer State = active
4 Order State = open
5 Order Item State = open
6 Product State = available
7 Order Value < $100
8 Order Value + Cust Deb
Order Item
*
Item Quantity = 0
Order Item (incomplete)
0
Order Item (complete)
0
Product
Actions List
10 Set Order State = closed
11 Set Order Item State = closed
12 Add Order Value to Cust Debt
13 Add 1 to Total Unpaid Orders
14 Set Order Value = Sum Value - Customer Discount
15 Set Amount Due = Order Value
16 Set Item Value = Item Quantity * Product Price
17 Subtract Item Quantity from Stock On Hand
18 Delete Order Item
18
5
11
16
6
1
17

13.5 Example version 3

Rule change: reject the Order Closure event if there is any incomplete Order Item

The figure below shows how you can redraw the event's Interaction structure to capture this rule.



Constraints List: Fail Unless

1 Item Quantity < Stock On Hand
2 Total Unpaid Orders < 5
3 Customer State = active
4 Order State = open
5 Order Item State = open
6 Product State = available
7 Order Value < $100
8 Order Value + Cust Debt < Credit Limit
9 Item Quantity > 0

Actions List

10 Set Order State = closed
11 Set Order Item State = closed
12 Add Order Value to Cust Debt
13 Add 1 to Total Unpaid Orders
14 Set Order Value = Sum Value - Customer Discount
15 Set Amount Due = Order Value
16 Set Item Value = Item Quantity * Product Price
17 Subtract Item Quantity from Stock On Hand

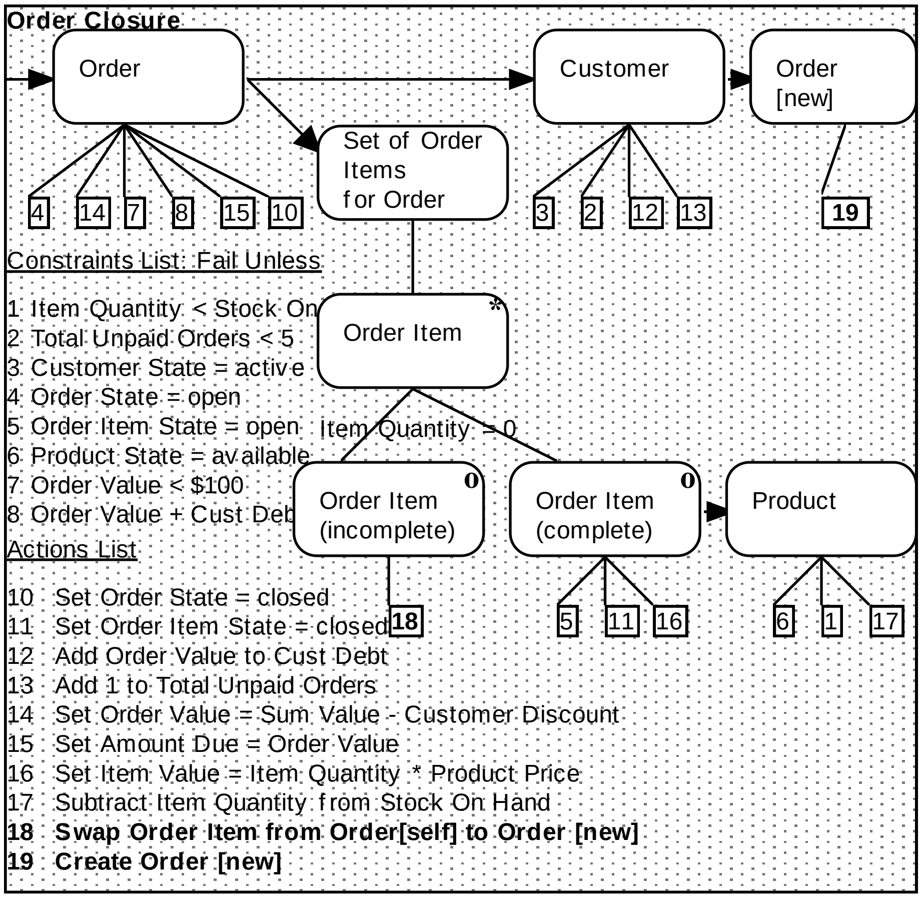
This variation of the business rule is a constraint condition rather than a business control flow condition; it appears as constraint number 9.

Notice that in this version of the specification, the processing logic of a successful event does not include incomplete Order Items.

13.6 Example version 4

Rule change: if there are incomplete Order Items, create a new Order and transfer all the incomplete Order Items to it.

The figure below extends the event's Interaction structure with an extra component (the new Order) to specify the rule: create a new Order object for the same Customer and transfer all the incomplete Order Items to it.



Constraints List: Fail Unless

1 Item Quantity < Stock On
2 Total Unpaid Orders < 5
3 Customer State = active
4 Order State = open
5 Order Item State = open
6 Product State = available
7 Order Value < $100
8 Order Value + Cust Deb

Actions List

10 Set Order State = closed
11 Set Order Item State = closed
12 Add Order Value to Cust Debt
13 Add 1 to Total Unpaid Orders
14 Set Order Value = Sum Value - Customer Discount
15 Set Amount Due = Order Value
16 Set Item Value = Item Quantity * Product Price
17 Subtract Item Quantity from Stock On Hand
18 Swap Order Item from Order[self] to Order [new]
19 Create Order [new]

Remember from the earlier chapters in this series that an arrow specifies first of all one-to-one association, and second the direction from which the object is identified. In this case the arrow suggests that the primary key of the new Order object is calculated from a value stored in the Customer object. If the primary key was input with the event parameters, then the arrow could go directly from the event to the Customer object.

13.7 Conclusions

Changes to business rules cause changes to event rules tables and/or Interaction structures. This is not such a bad thing; changing a behavioral model takes less effort than changing a structural model. On the other hand, if the rules are likely to change you might well look to create business rules classes in which the rules can be declared as data attributes and changed dynamically.

Discrete event modeling gives us a natural level of granularity to define processing. It naturally distinguishes facts (control flow conditions) from constraints (preconditions). It helps us to establish the right balance between them.

14. Generic events: reuse between event models

Even where there are no generic super classes in the entity model, you may find considerable potential for reuse of processing between event models. This chapter describes a rational way to discover reuse between events and specify an event class network.

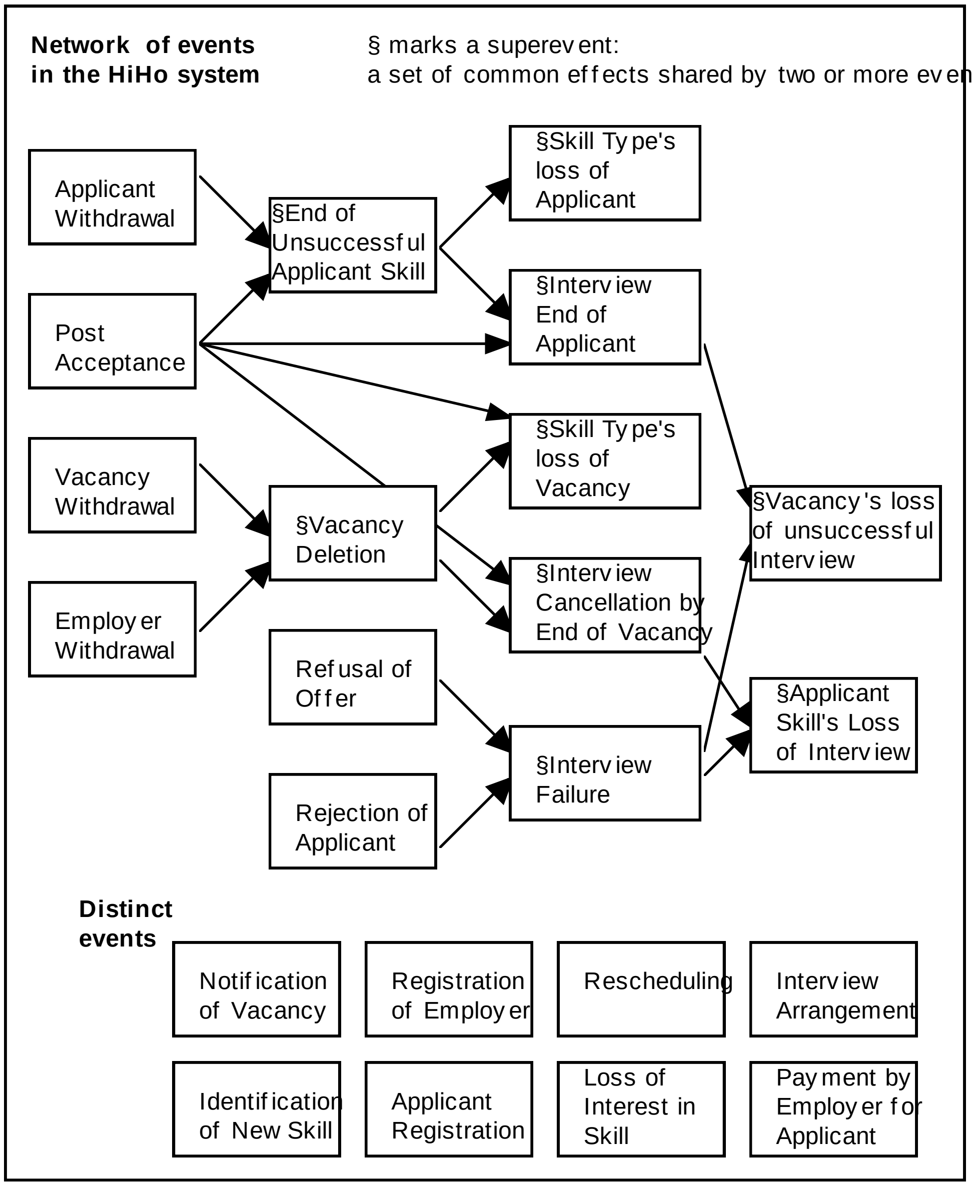
SSADM includes a formal event-oriented technique for defining reuse between business services. In this technique, the business service is called a discrete "event" which has an “effect” on each of one or more “entities”. Two discrete events can share a common process, known as a "super event". The OO concept of a responsibility is akin to an effect, or more interestingly, to a super event.

In short, you:

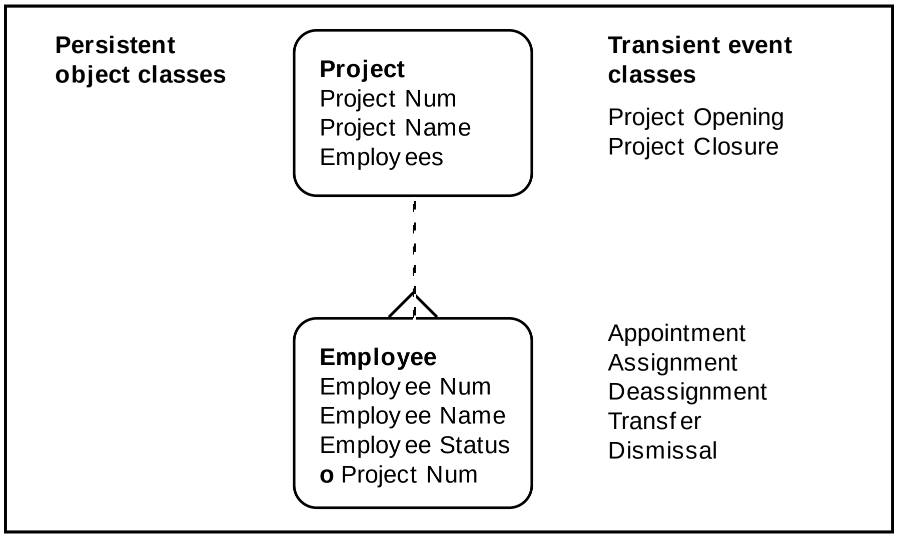
- identify events.
- identify where two or more events have same pre and post conditions wrt an entity (that is, the several events appear at the same point in the entity's state machine and have the same effect).
- name the shared effect as a super event.
- analyse to see if the super event goes on from that entity (where the events’ access paths come together) to have a shared effect on one or more other entities, and if so, you adopt the super event name in specifying those other entities’ state machines.

I don't mean to persuade you to use this exact “super event” analysis and design technique. I only want to indicate that reuse via event-oriented analysis and design has a respectable and successful history, since many object-oriented designers are unaware of this history.

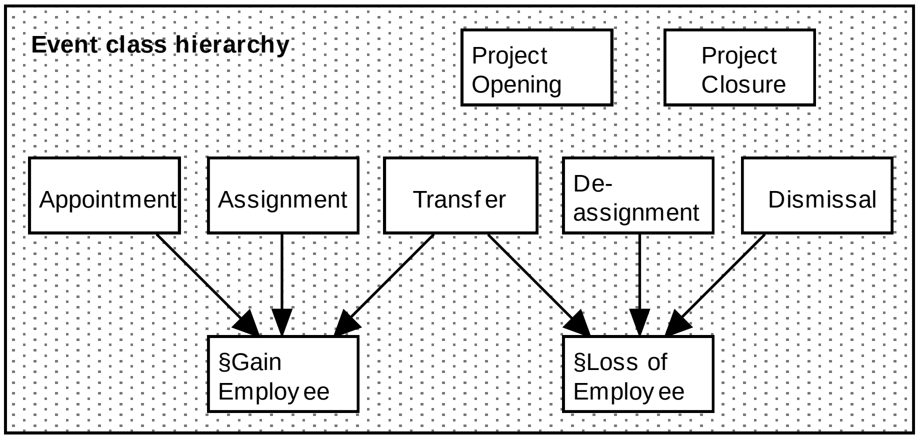
In the book ‘Object-Oriented SSADM’ Keith Robinson documented the entity and event model of recruitment agency case study. Look at the network of reuse between events documented in the figure below!


Network of events in the HiHo system
§ marks a superevent: a set of common effects shared by two or more even
Applicant Withdrawal
Post Acceptance
Vacancy Withdrawal
Employer Withdrawal
§End of Unsuccessful Applicant Skill
§Vacancy Deletion
Refusal of Offer
Rejection of Applicant
§Skill Type's loss of Applicant
§Interview End of Applicant
§Skill Type's loss of Vacancy
§Interview Cancellation by End of Vacancy
§Interview Failure
§Vacancy's loss of unsuccessful Interview
§Applicant Skill's Loss of Interview
Distinct events
Notification of Vacancy
Registration of Employer
Rescheduling
Interview Arrangement
Identification of New Skill
Applicant Registration
Loss of Interest in Skill
Payment by Employer for Applicant

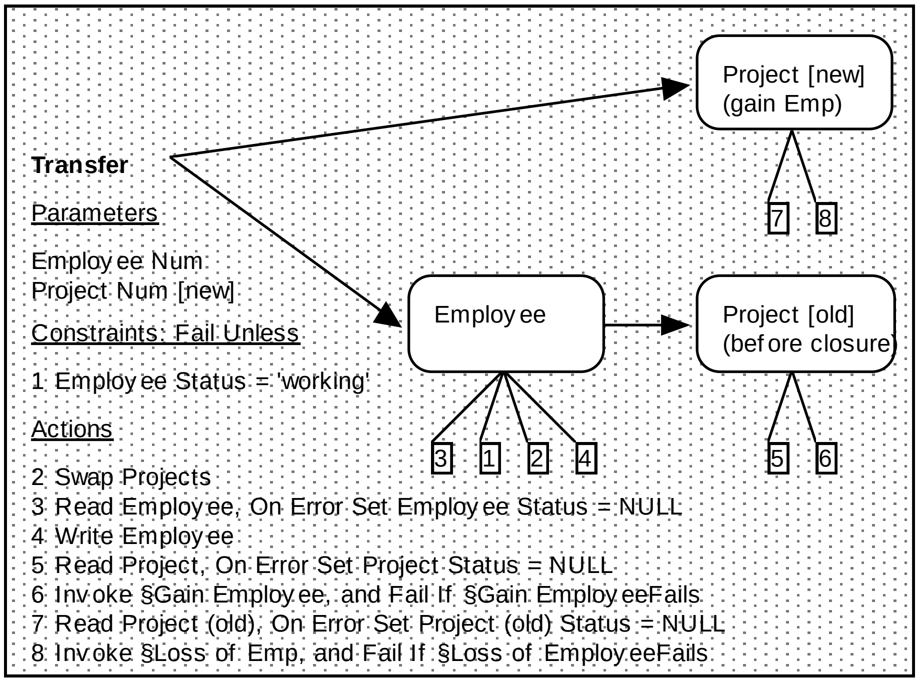
Keith's recruitment agency case study is too large a case study to illustrate how reuse between events works here. The figure below introduces the structural model of very much smaller case study.



The figure below shows the network of invocations that can be discovered in this trivial two-class system; there are two superevents, both called by three ordinary events.



The figure below shows how the Transfer event calls both of the superevents (in actions 7 and 9).



Transfer

Parameters

Employee Num
Project Num [new]

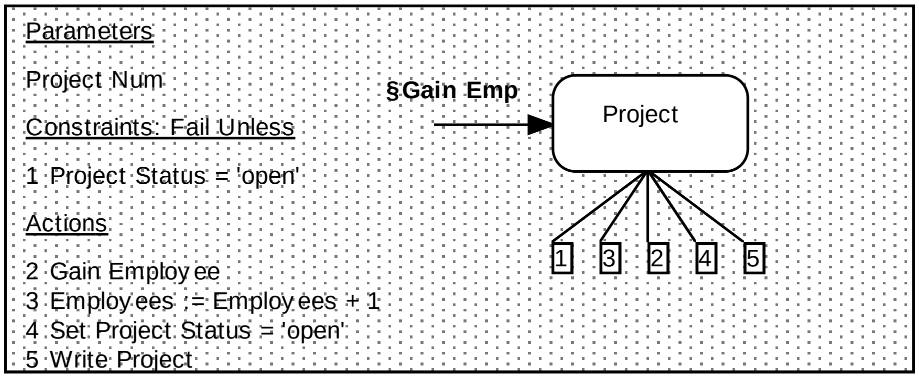
Constraints: Fail Unless

1 Employee Status = 'working'

Actions

2 Swap Projects
3 Read Employee, On Error Set Employee Status = NULL
4 Write Employee
5 Read Project, On Error Set Project Status = NULL
6 Invoke §Gain Employee, and Fail If §Gain EmployeeFails
7 Read Project (old), On Error Set Project (old) Status = NULL
8 Invoke §Loss of Emp, and Fail If §Loss of EmployeeFails

The figures below are the Interaction structures for the superevents called by Transfer (and two other ordinary events).



Parameters

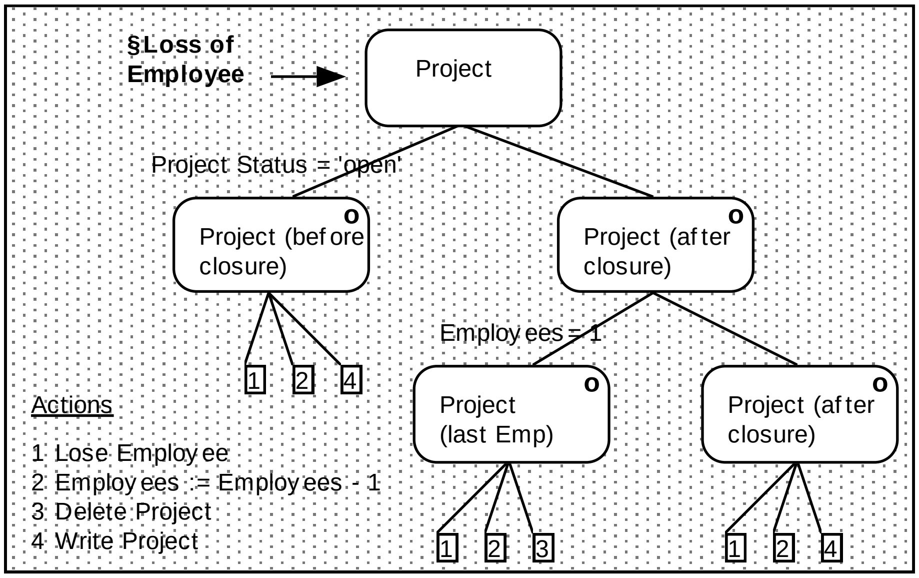
Project Num

Constraints: Fail Unless

1 Project Status = 'open'

Actions

2 Gain Employee
3 Employees := Employees + 1
4 Set Project Status = 'open'
5 Write Project

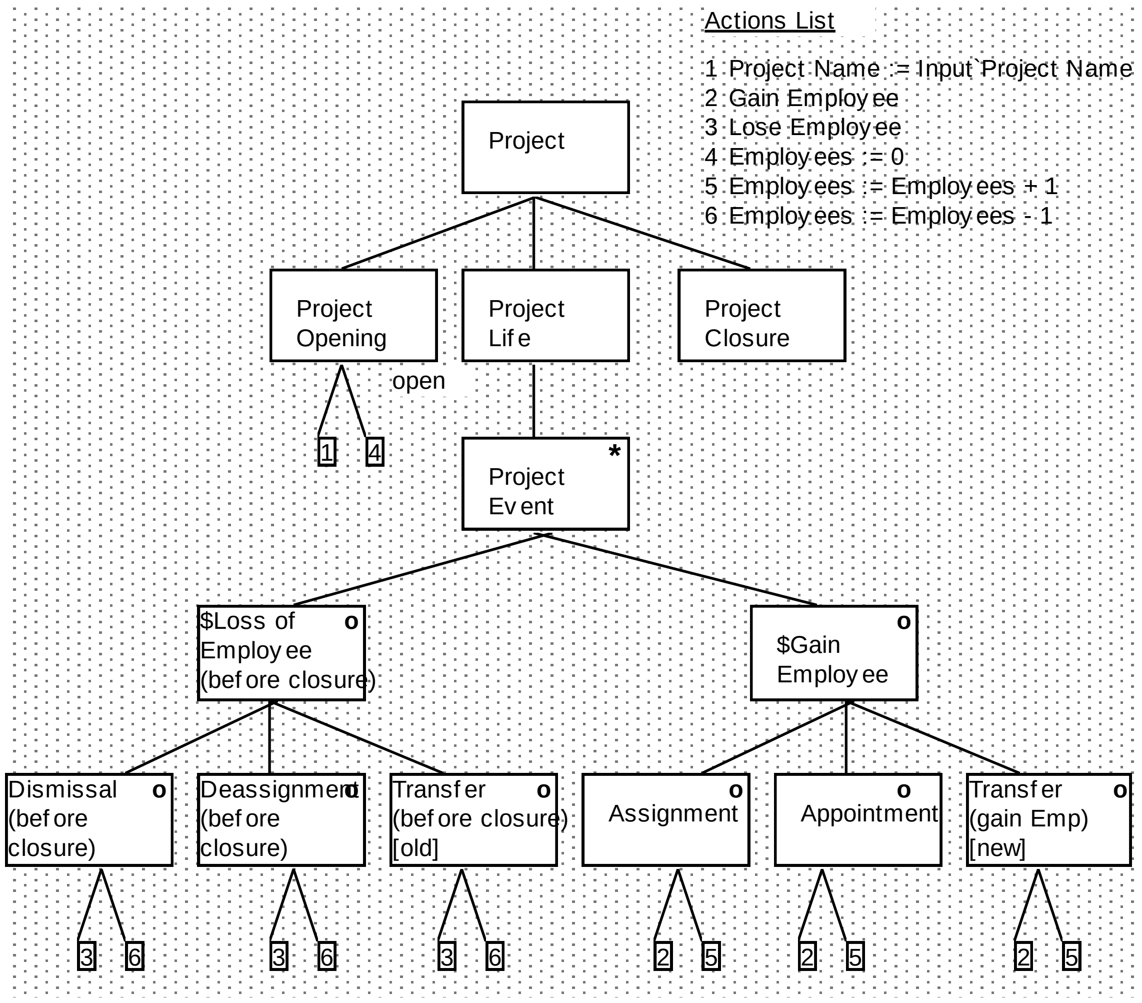


A CASE tool can generate most of the detail in these Interaction structures from the entity state machines. A CASE tool would generate a flat three-way selection in the last figure; but I have chosen to reshape this into two binary selections by separating the two different conditions that must be tested.

14.1 Discovering reuse in life history analysis

The three different events that cause a Project to gain or lose an Employee come together in the entity state machine of the Project class as options of a selection.

Where events under different options of a selection have the same effect (trigger the same actions), it is possible (though not always advisable) to declare the selection of options as a superevent. I use the symbol § to mark a superevent in a entity state machine.


Actions List
1 Project Name := Input`Project Name
2 Gain Employee
3 Lose Employee
4 Employees := 0
5 Employees := Employees + 1
6 Employees := Employees - 1
Project
Project Opening
Project Life
Project Closure
open
1
4
Project Event
*
$Loss of Employee (before closure)
o
$Gain Employee
o
Dismissal (before closure)
o
Deassignment (before closure)
o
Transfer (before closure) [old]
o
Assignment
o
Appointment
o
Transfer (gain Emp) [new]
o
3
6
3
6
3
6
2
5
2
5
2
5

Think of each superevent as a common module, invoked by each one of the events shown as options beneath it.

Once a superevent has been declared like this in a entity state machine, you may use the superevent in other entity state machines, instead of duplicating the same selection of three events.

In other words, wherever it appears in other entity state machines, the §Loss of Employee superevent is a common effect of the three different events that remove an Employee from a Project shown in the figure above.

14.1.1 The effects of events

An **effect** is the appearance of an event inside a entity state machine. One event instance may trigger one of several effects within one entity state machine. Different effects of one event can be distinguished by adding an effect name in brackets.

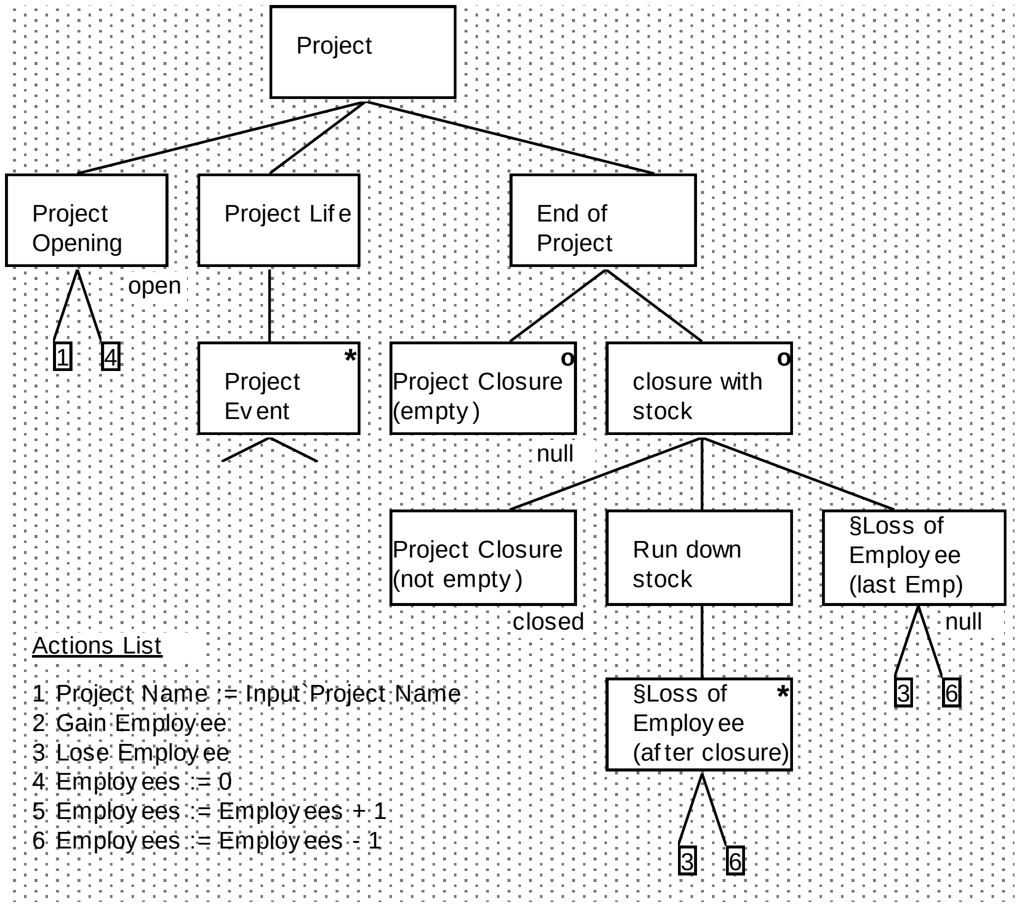
An effect name tells us briefly about the difference between effects. It may summarise the effect of actions (‘actual deletion’ or ‘intended deletion’). It may describe the state the object instance must be in for that event effect to occur, either in terms of different positions in the life (‘active’ or ‘dead’), or in terms of different values of an attribute (‘last’ or ‘not last’).

14.1.1.1 One event with optional effects on a class

The figure below shows that the Project Closure event has two different effects on a Project marked:

- Project Closure (empty)
- Project Closure (not empty)

The Project Closure event will delete the Project if it is empty (has no Employees remaining) or else act as a state-change on the way to deletion when later the last Employee is removed from the Project.

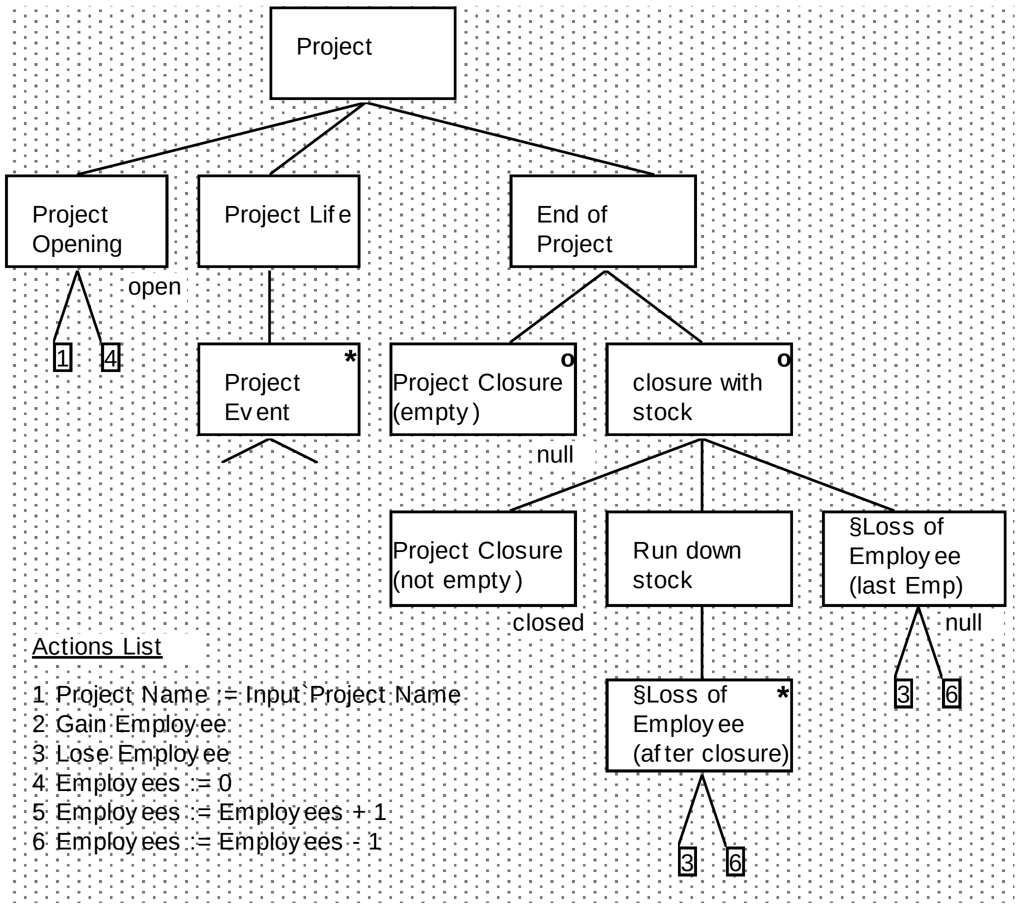


The figure above also shows that no new Employees may be added after a Project has been closed, since the §Gain of Employee superevent does not appear at this point in the entity state machine. And it shows that existing Employees may be removed after a Project has been closed, since the §Loss of Employee superevent name does appear at this point in the entity state machine.

14.1.1.2 A superevent with optional effects on a class

The figures above show that §Loss of Employee superevent itself has three different effects on a Project, marked:

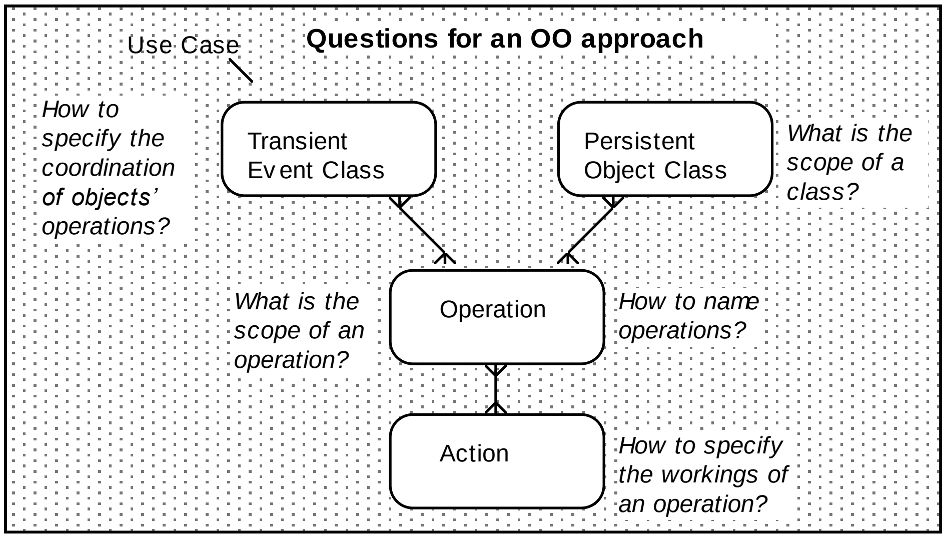
- §Loss of Employee (before closure)
- §Loss of Employee (after closure)
- §Loss of Employee (last Employee)


Project
Project Opening
Project Life
End of Project
open
1
4
Project Event *
Project Closure (empty) o
closure with stock o
null
Project Closure (not empty)
Run down stock
§Loss of Employee (last Emp)
closed
null
Actions List
1 Project Name := Input Project Name
2 Gain Employee
3 Lose Employee
4 Employees := 0
5 Employees := Employees + 1
6 Employees := Employees - 1
§Loss of Employee (after closure) *
3
6
3
6

Following a typical object-oriented analysis and design method you specify each class using some kind of template. You may also draw state-transition diagram for it. A entity state machine diagram like the one above combines both the class specification and state-transition diagram.

14.2 Entity and event orientation

The figure below shows a few questions raised by OO methods, annotated on a crude metamodel of system specification concepts


Questions for an OO approach
Use Case
How to specify the coordination of objects' operations?
Transient Event Class
Persistent Object Class
What is the scope of a class?
What is the scope of an operation?
Operation
How to name operations?
Action
How to specify the workings of an operation?

The figure above is very much oversimplified, but it captures something of the orthogonality between persistent objects and transient events - the many-to-many relationship between them. You can address the questions in the diagram by taking an event-oriented approach.

14.3 What is the scope of a class?

There are various ways to partition a system into classes. For example, relational data analysis and life history analysis can give different answers. You can define the size and scope of a class intuitively to begin with, then use event-oriented analysis techniques to refine the answers your intuition comes up with.

The notion of splitting one class into parallel entity state machines, one for each aspect of the class, turns out to be an important analysis and design technique, used in 'OO and business data'.

14.4 What is the scope of an operation?

It is relatively easy to list the elementary data attributes of each class, especially where a business already maintains some persistent data that you can inspect. You might think it will be just as easy to recognise and list the operations, but this is not so in our experience.

Some object-oriented approaches simply list one enquiry and one update operation for each attribute. This is the wrong level of abstraction. These are elementary *actions* rather than *operations*. You need to work at a higher level, the level of an event effect composed of several (perhaps one, perhaps ten) elementary actions.

14.4.1 How to discover the 'right' set of operations?

You don't want to clutter up your system with operations that are irrelevant, which fall into state of neglect and disrepair, which are never used by anyone. You do want to specify operations that are meaningful and useful.

To be meaningful and useful, an operation must be invoked by at least one event. So meaningful and useful operations naturally emerge from an event-oriented analysis. You can address the question of where the operations come from by taking an event-oriented approach to requirements capture and knowledge acquisition.

You should define each possible effect of a transient event on a persistent object, in an operation of that class. This approach encourages you to define exactly those operations that must be invoked to meet your system requirements, and only those.

14.4.2 How to name operations?

You should name operations after the classes that own them and the events that invoke them. To begin with, you can assume that each event fires a unique operation in an object. The initial list of events can be taken to be the initial list of operations. This remains true for the majority of events and operations.

However, the behavior analysis can reveal operations fired by more than one event. You can define these as reusable 'superevents' and give them a name that reflects their effect on the object, rather than their invoking event. For example, §Light On is a superevent invoked by the Button Push event in chapter <>, as well as the Door Opening event not shown in that chapter

Superevent analysis is a significant advance on current object-oriented techniques; it helps us to define useful and reusable operations via a rational analysis and design process.

14.4.3 How to specify the implementation of an operation?

One event effect in a entity state machine is close to the object-oriented idea of an operation, but there are two variations on this simple picture.

- One event may have more than one effect on a class. In an OOP implementation, you can join these effects within one operation under a selection of different cases. The selection is made by testing the state of the object when an event arrives.
- One effect on a class may be triggered by more than one kind of event. You can show this common effect as a 'superevent' and in an OOP implementation it becomes one operation.

14.4.4 How to specify the co-ordination of objects' operations?

The more you divide a system into self-contained modules, the more you have to work on the communication between modules, the interfaces and the message passing routes. Consider in the Pupil Transfer event example, how does the Pupil Transfer operation in Pupil communicate to Pupil Transfer operation in the School [old], and what data is passed back and forth?

Event modeling helps you sort out the way that operations in objects of different classes are co-ordinated when an event happens. It helps you not only to specify the right or best set of operations, but also to design the message routing between objects.

> 'To ensure proper modularity… wherever two modules A and B communicate, this must be obvious from the text of A or B or both.'
>
> Meyer

Event modeling encourages you to name communicating operations with the same name, but following the third of the three implementation strategies in chapter 2, you don't have to make every interface fully explicit.

You can declare the data passing between objects in one place as a shared resource. You could create a Pupil Transfer event module/class that encloses the objects Pupil, School [old] and School [new] and enables them to communicate via the 'working storage' of the event module/class, rather than explicitly sending data to each other.

14.5 Conclusions

Object-oriented techniques suffer from not making the events that invoke the operations explicit. The techniques described here extend object-oriented theory in this direction. They provide a rational way to discover reuse between events and specifying an event class network.

15. The granularity of events

This short chapter rounds up a few points that may be helpful to the event modeler.

15.1 The importance of achieving a shared understanding of events

Some analysts confuse events with ‘use cases’ or ‘functions’, or confuse an event with the ‘operation’ it fires in just one class. And some use the term ‘event’ in different contexts: business event, GUI event, message, etc.. Any methodology will come unstuck if such confusions are allowed to prevail. The concept and the level of granularity must be sharply defined.

People can find it helpful to think of an event as: a real-world event, or a business event, or a data group input when an end-user presses a ‘send’ button, or a user interface transaction, or a database transaction. But these ideas are too subjective. Designers on either side of the application-presentation interface must share the same idea of what an event/enquiry is, and what level of granularity it is defined at.

An event is a minimum unit of consistent change to the stored data within the scope of the system being engineered. It is a short-term process that affects one or more objects in the system; it moves a system as a whole from one consistent state to the next; it either happens or it doesn’t; it must succeed or fail as a whole.

Defined thus, an event fits nicely into a three-tier software architecture. It gives us reusable processing components in the business rules layer, ones that can be invoked from many different places in the user interface layer. It matches the idea of a database transaction or commit unit in the data storage layer.

An event tends to be small. The size of an event is not arbitrary. It is the *smallest* process that moves the system’s persistent data from one consistent state to the next. A large database update program, even though implemented as one physical database commit unit, may implement many conceptual events.

An event is sudden, happens in an instant, is transient, but leaves its mark on persistent objects.

An event moves a system from one state to the next. The state of an Enterprise Application is recorded in its memory or database. The database must be consistent. That is, the facts in it must not contradict each other. An event is a process that moves a set of related persistent objects from one consistent state to the next consistent state.

An event is all-or-nothing. It is indivisible. It cannot half happen. (By the way, this is also true of what physicists call ‘events’ in quantum electrodynamics.) All the effects of an event must fail if any one of them fails. If an event gets half way to completion and fails, then the whole event must be backed out or reversed. In the terms of database technology, this may be called the ‘roll-back’ of a commit unit.

An event has a scope. One event might affect only one object in a system, but in general it can affect many classes and many objects. You have to envisage and specify the effect of the event on the system as a whole. Within the computer system, it is a process (not just one method in one object) with a beginning and an end. Some think of an event as merely the trigger of a process, but it is also the process itself.

15.2 Systems with only trivial events

Remember, the size of an event is not arbitrary. It is the *smallest* process that moves the system's persistent data from one consistent state to the next. It is possible to construct the simplest kind of Enterprise Application out of events that do one of three things:

- create a single object of a class (assigning a new key value and relating the object to any mandatory master objects)
- update a single attribute of an object
- delete a single object.

So there will be two events for each class and one trivial event for each attribute. Some tools will generate a GUI that enables you to enter such simple create, update and delete events

What about the business rules? In the simplest kind of Enterprise Application, all the constraints on event processing are either:

- constraints on the domain of an attribute, or
- constraints on the presence or absence of a relationship between objects.

So ask of your technology, does it offer the following mechanisms for defining constraints:

- a data dictionary, for the domain of an attribute?
- a database structure, for the presence or absence of a relationship between objects?

15.2.1 Note for SSADM readers

This section explains why SSADM is over-the-top for some simple systems, designed using some technologies. It also explains where SSADM event modeling techniques start to become more useful, for defining more complex constraints where events must test the state of stored attributes, inter-attribute domain constraints and so on. What is needed is a better understanding of 'triage', how to apply effort that is appropriate to the severity of the problem.

15.2.2 Aggregating small events

Whether datan entry is on-line or off-line, you may choose to batch trivial events together into an 'aggregate event', and implement the whole aggregate as one physical database commit unit.

Consider an object with twenty text attributes. The user could update each attribute on its own without reference to the others, so each attribute replacement is, logically speaking, a distinct event.

However, a common practice in physical design is to allow users to overtype data on the screen as they see fit, batch all of the attribute-replacement events into a single physical database commit unit, and only commit the data to the database when the user signals they are ready, perhaps by seeking to close the window.

15.2.3 Benefits

Motivations for designing aggregate events include:

- reduce the number of events to be coded
- reduce the number of accesses to data storage
- reduce function-business component traffic
- reduce client-server traffic
- simplify the audit trail.

These motivations are normally stronger in on-line input. Strange as it may seem, there is less reason for aggregate events in off-line input, where performance is less of an issue.

15.2.4 Costs

Of course the processing of an aggregate event is more complex, but this hardly matters if you are simply batching several attribute-replacement events for one object.

More seriously, the user has to wait longer before getting a response to their input.

And the error response to an aggregate event raises some dilemmas. Each one of the events within the aggregate may succeed or fail on its own. What if one logical event fails? What if several fail?

You might set the standard that if one data input element validation fails on save, the system takes the user to that datan entry point. If several, the system takes the user back to each in turn. But hand coding this sort of thing in an ad-hoc environment (such as Delphi perhaps?) might be difficult.

If it is difficult to display multiple error messages, or users find them confusing, the simple option is to roll back the whole aggregate of events as soon one of them fails, reporting on just that one failure.

15.3 Dividing large events

Some don’t like the tedium of defining a large number of trivial events. But this is only an objection to boring work, not something to worry about in principle.

It isn’t the simple events that take up the time. Most of your time will be spent understanding and specifying a relatively small number of complex events. Any rule or definition that reduces the scope and minimises the complexity of events is a great advantage here!

15.3.1 Mistakenly-defined large events

Designers may mistakenly define too large an event. Perhaps the event is really more like what some people call a ‘business event’ or ‘user task’ or ‘use case’ or ‘scenario’. Perhaps the large event is several birth events compressed into one.

Whatever the reason, you can simply divide the large event into smaller application events. This should increase the reusability of the events.

If what is supposed to be one event can get half way to completion and fail, but the changes to persistent data do not have to be rolled-back (because the data is internally consistent), then what appeared to be one event was really an aggregate of two or more distinct events.

15.3.2 Properly-defined large events

An event may legitimately be large, in one of two ways.

Some events have a long list of parameters. This is normally true only of birth events that create an object, or perhaps more than one object. Or events that are really a batch of trivial attribute-replacement events

Some events have a wide-ranging set of effects on stored information objects. This is normally true only of death events that ‘cascade’ from one object to another.

Designers are sometimes led to divide such a large event into two or more partial events. This is dangerous because it can lead to the problem of the distributed commit unit, and the need to define a manual worklflow to ensure data integrity.

15.3.3 Events with a long list of parameters

Most events and enquiries only have one or two parameters. Few events and enquiries have so many parameters that you need to validate them as you go along.

Lots of parameters might be a symptom that you’ve got the wrong idea about an event. So one answer is - stop trying to be too clever. If you have batched trivial events together, then you would do better to unbatch them.

But there is always the possibility of an event that really does require lots of parameters, and while some users don’t want datan entry interrupted with intrusive error messages, others may want event/enquiry parameters to be validated as they enter them, before all the parameters have been completed. If so, what to do?

15.3.3.1 Pre-event enquiry solution

The normal solution is to preface the event with a pre-event enquiry that duplicates some or all of the data retrieval and business rule testing carried out by the event.

So you don't invoke the event until all the parameters have been entered and validated individually by the pre-event enquiry processes.

Difficulty: You duplicate process specification and code, meaning there is a performance overhead now and a maintenance overhead later.

Multi-user difficulty: If you do lock the data from the start of the pre-event enquiry module to the end of the event module, this means locking a lot of data for a lot of the time, sometimes unnecessarily.

Multi-user difficulty: If you do not lock the data from the start of the pre-event enquiry module to the end of the event module (and some technologies prevent you), then you have to repeat all the validation in the event module, just in case another user has been working on the same data! This is discussed further below.

Technology question

Will your technology automatically apply the validation tests for an event as it reads data, and then again when it commits an event, without you having to lock the data in the meantime or write two similar processes (i.e. a pre-event enquiry and an event)?

15.3.3.2 Co-routine solution

There is another solution, theoretically preferable but practically difficult. You might run the event/enquiry module as a co-routine, executed in stages alongside the datan entry. This involves 'inverting' the event/enquiry procedure (see Jackson, 1975) or dismembering it into distinct subroutines.

Unfortunately, few people understand program inversion and most technologies prevent you from implementing the several stages or parts of a co-routine within the span of a database commit unit; especially if the event processing spans client and server.

15.3.4 Events with a wide-ranging set of effects on stored data

Given an event with a wide-ranging set of effects on stored data, you may find you cannot contain the event within the span of an automated database commit unit. Two reasons are:

15.3.4.1 Passage of time

The event takes so long you have to divide its processing into stages, and your technology prevents you from implementing the several stages within the span of a database commit unit.

This reason is very rare. When it happens, you are probably best advised to process the event off-line, perhaps overnight, rather than divide it into stages.

15.3.4.2 Distribution of data

The affected data is stored in different locations beyond the control of a coherent database management system.

This is the most common reason. It causes more pain and cost than almost anything else in system design.

The consequences of dividing an event, not being able to process one event in all related systems at the same time, within one commit unit, are many and various, as the next section indicates.

15.4 Conclusions

???

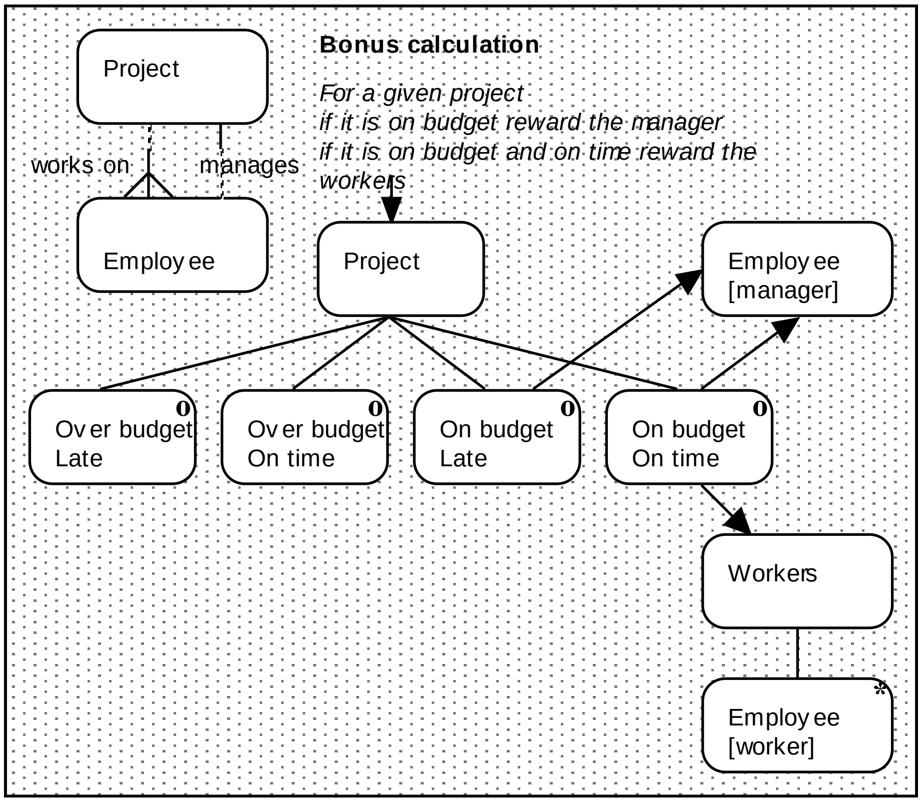
16. Complications in event modeling

This chapter discusses various difficulties that arise in modeling the control flow of an event. It covers restructuring a selection to define a transient association, resolving a structure clash between selections, multiple hits in a V shape, and multiple hits in a diamond shape.

16.1 Restructuring a selection to achieve association

Some objects act as a monitor object for an event type. They decide whether an event instance has one effect or another by inspecting a condition. Some monitor objects go on to act as a gatekeeper object. They decide whether to pass event instance on to another object or not.

The figure below shows Project acting as a gatekeeper object for Employee, but there is an error in the event's Interaction structure.

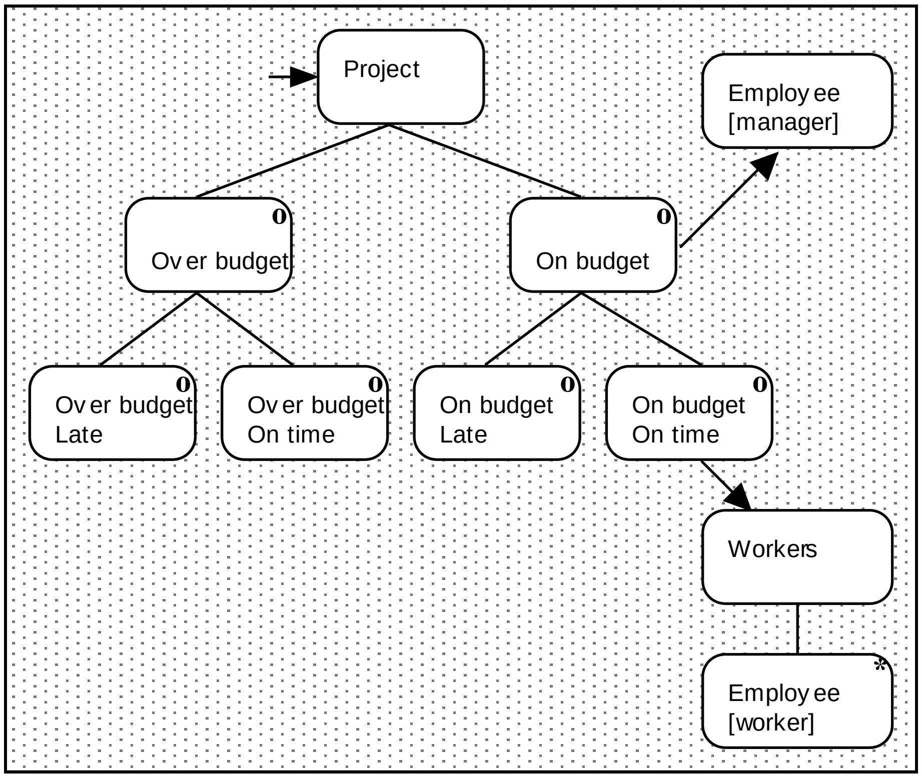


The diagram is an invalid specification because you cannot draw two arrows from one substructure to a single node in another structure. This destroys the concept of one-to-one association.

A multi-way selection is a generative pattern that prompts a question.

- Ask of a multi-way selection: Is there duplication of processing between two or more options?

If yes, try rearranging the selected options under a higher-level selection.



Rearranging the selection into two levels works in this case. In the general case it doesn't always work, because drawing one combination of options together divides other combinations of options. The general solution to this kind of problem is revealed in the next section.

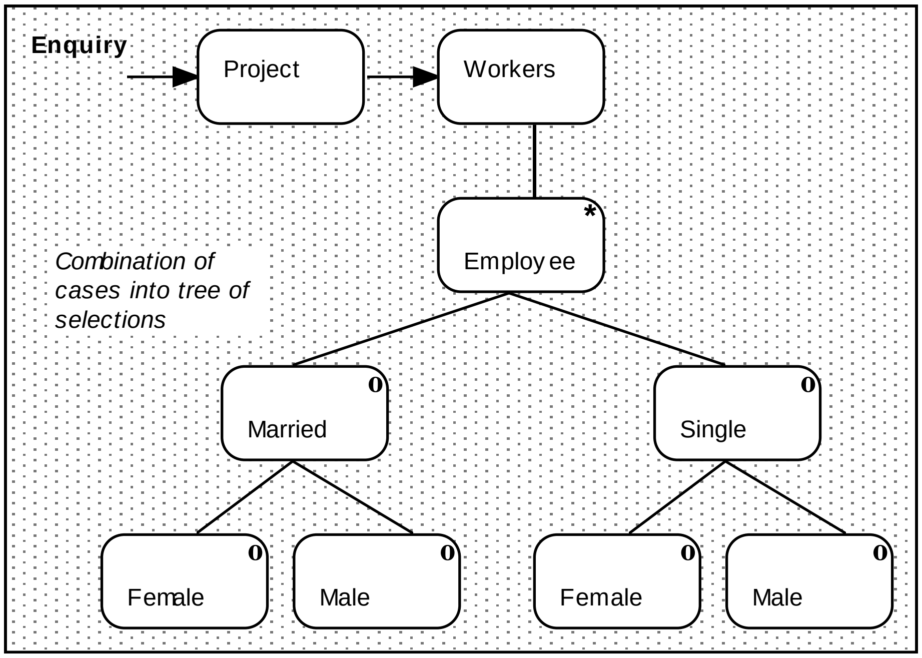
16.2 Resolving a structure clash between selections

The figure below shows report of all the Employees working on a Project. How to draw the enquiry access path for this report?

```
Project xxx
Employee aaa          Married  Man
Employee bbb          Married  Woman
Employee ccc          Single   Woman
Employee ddd          Single   Woman
Employee eee          Single   Man
```

Suppose the words in the report are not stored directly as attributes of an Employee, so your enquiry process has to translate indicators into text as it goes along.

There are four different permutations of data in a print line. You might perhaps construct an enquiry access path as in the figure below.

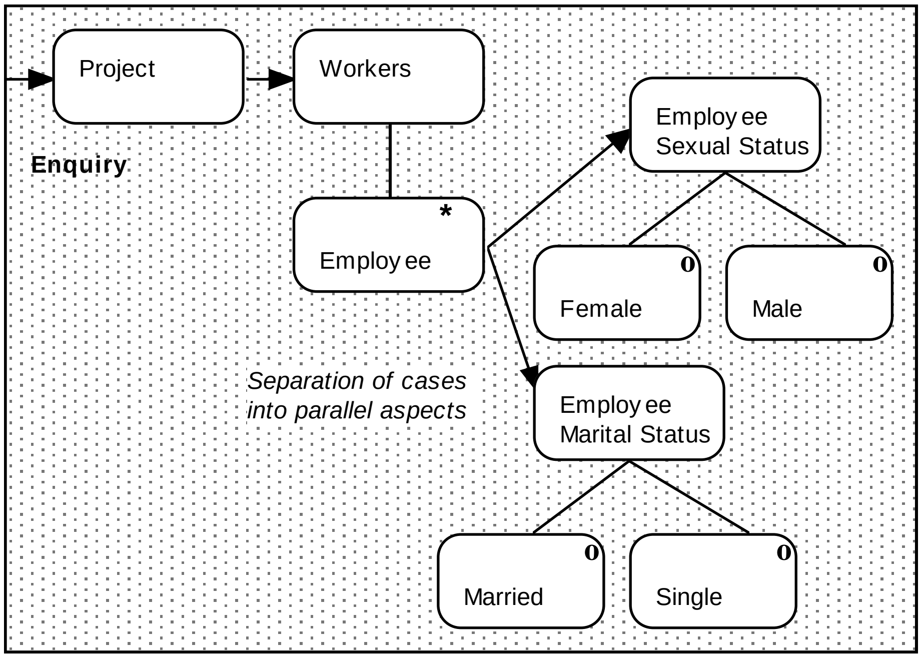


The figure below extends the report with an extra field:

Project xxx			
Employee aaa	Married	Man	Local
Employee bbb	Married	Woman	Local
Employee ccc	Single	Woman	Foreign
Employee ddd	Single	Woman	Local
Employee eee	Single	Man	Foreign

Try extending the enquiry access path above to show the extra permutations. Of course, there are now nine different permutations of data in a print line. This combinatorial explosion reveals there is a structure clash between selected options.

The figure below shows you can resolve the structure clash by constructing an enquiry access path with each selection drawn as a parallel aspect. You can now add further selections without any fear of combinatorial explosion.



Parallel aspects appear here in an enquiry access. They also appear in Interaction structures. Notice that there is logically no precedence between the parallel aspects, though you will have to introduce an arbitrary sequence of selections into any program you write based on this specification.

- Ask of a multi-way selection: Is there a combinatorial explosion?

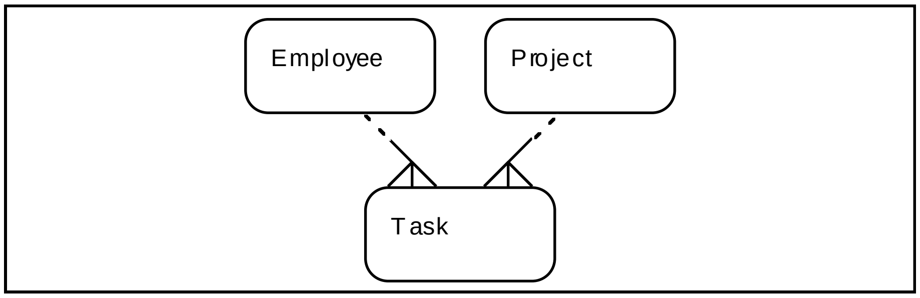
If yes, then restructure as parallel selections (or a sequence of selections).

16.3 Multiple hits in a V shape

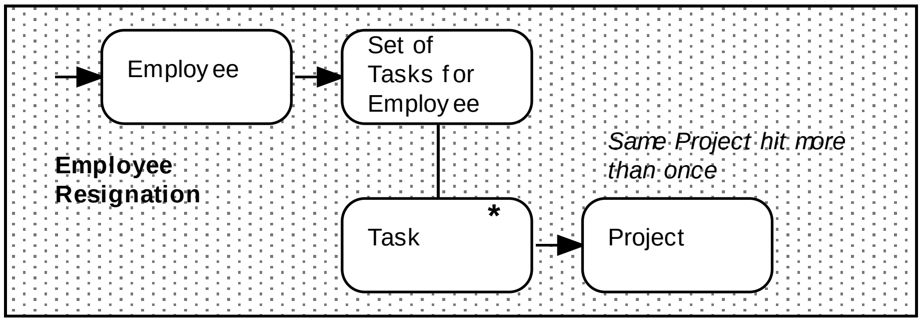
A problem that does not seem to be recognised in OO literature to date is that one event instance may hit the same object more than once. If the objects are in a database, the event may thus lock an object from receiving further messages from itself!

There are two reasonably obvious situations where multiple hits may occur, related to patterns in the structural model. The first is in the V shape.

The figure below shows a model in which Task has its own serial number (not simply a compound key of Employee and Project), so an Employee may perform several Tasks within the same Project.



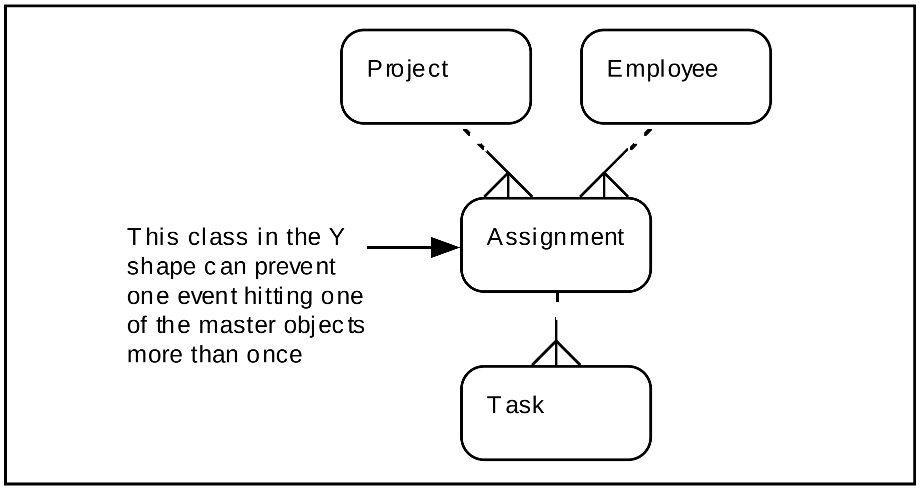
The figure below shows that when an employee resigns, the resignation event is broadcast around the V shape, cutting all the Employees Tasks from their Projects and perhaps having some update effect on the Project into the bargain.



The figure above looks OK. The use of arrows seems perfectly valid; but it is not. The transient association between Task and Project is not one-to-one as the arrow implies. There are more Task objects affected by the event than Project objects. The event will hit the same Project several times.

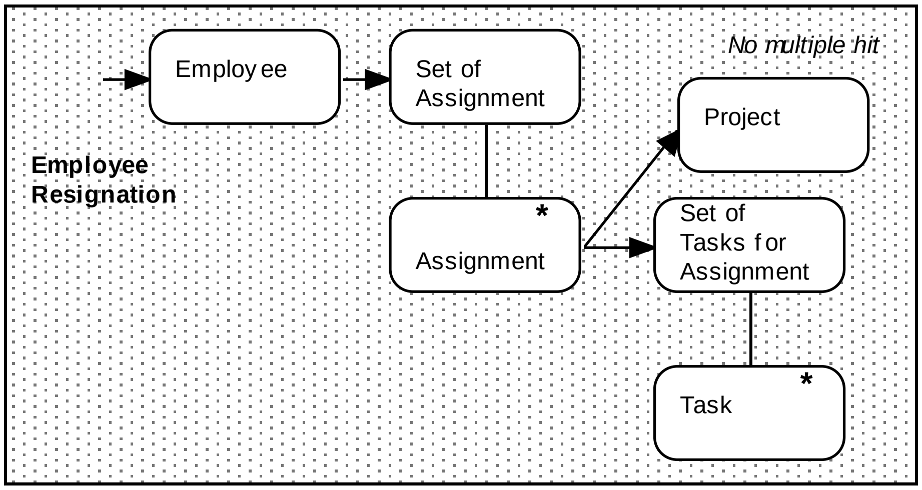
16.4 Data-oriented resolution of multiple hits

The figure below shows you can avoid multiple hits in V shapes by introducing a Y shape derivable sorting class. E.g. the Assignment object represents one combination of the two master classes Employee and Project.


Project
Employee
This class in the Y shape can prevent one event hitting one of the master objects more than once
Assignment
Task

Assignment might be a V shape domain class, that is a class introduced by users to constrain who is allowed to work on a Project. Or it might be a derivable sorting class. Either way, it works to resolve the multiple hit problem.

The figure below shows that when an employee resigns, and the event is broadcast around the V shape, the event will hit each Project only once.



Given you can separate the application class model from the data storage structure, the question arises as to whether neither, either or both should include a Y shape derivable sorting class.

The book ‘Patterns in data modeling’ proposes you might specify the derivable sorting class in the business rules layer only. You can code enquiry processes in the business rules layer as though derivable sorting class exists, then code the application/data interface to sort the stored data and present the required objects to the business rules layer as it request them.

16.4.1 Process-oriented resolution of multiple hits

The three strategies for implementing events described in the earlier chapters in this series provide other ways to resolve the problem of multiple hits.

16.4.1.1 Comb: centrally-controlled message passing

This strategy was described earlier thus: ‘In one possible OO implementation, the whole event’s Interaction structure is controlled by an event manager that implements something like a two-phase commit. First it calls each object with the event, then it reads all the objects’ replies to check they are in the correct state, then it invokes each object again, telling it to process the event, update itself and reply with any required output.’

Following this strategy, you can get the event manager to resolve the multiple-hit problem. The event manager program keeps track of which objects have already been invoked with an event in the first phase, and when each is invoked in the second phase with a commit message.

16.4.1.2 Chain: Hand-to-hand message passing

This strategy was described in earlier thus: ‘In a more OO implementation, the objects pass the event from one to another (as though following the arrows in an event’s Interaction structure). This is fine for process control systems. It is not quite so easy in Enterprise Applications where a system event must build up a complex output data structure from the many concurrent information objects it affects.’

Following this strategy, you can get each object to resolve the multiple-hit problem. You can add code to lock the object on the first invocation of the first phase of an event, and unlock the object on the last invocation of the second phase of the same event.

In the first phase, the necessary code must ask the question, Has this object already been accessed by this event? To do this it must store not only the lock on the object, but remember the identity of the event which locked it.

In the second phase, the necessary code must ask the question, Is this the last time this event will hit the object? To do this it must remember the number of hits in the first phase, and countdown during the second phase until all have been committed.

This extra code should be shielded from the business rules layer, which should know nothing about locking, or other multi-user issues. In terms of the 3-tier architecture the code belongs in the data storage layer, it is a module of the application/data interface.

16.4.1.3 Procedure: combine the relevant parts of the classes into one

This strategy was described in earlier thus: ‘You can get around the need to define the message passing by extracting the relevant operations from each class, bringing them together into one procedure, and making them communicate via the local memory or working storage of that procedure.’

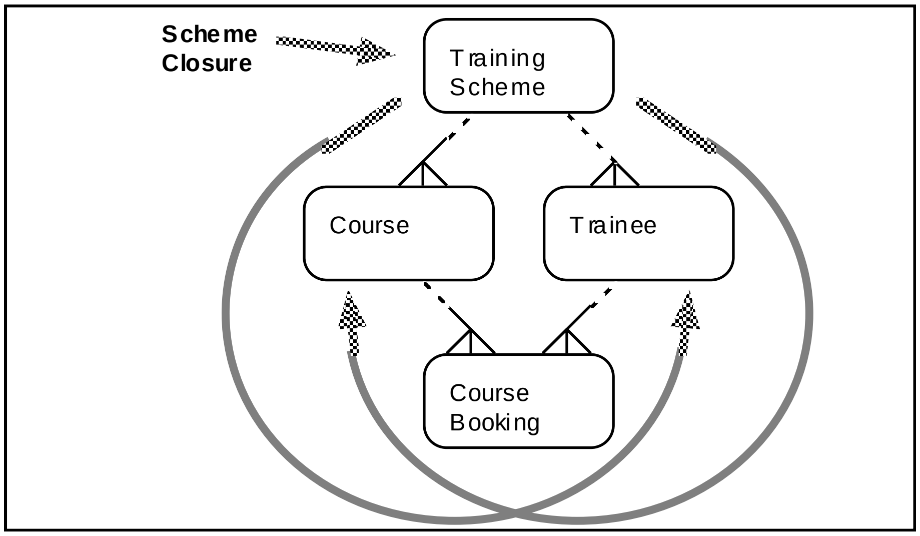
Following this strategy, you can move the database processing into the event manager program. This is really the conventional programming solution. I have arrived by a circuitous route at a procedural implementation of the event’s Interaction structure.

This may be viewed as a highly optimised form of OO implementation in which objects communicate via the working storage of a single procedure, rather than by sending messages to each other.

16.5 Multiple hits in a diamond shape

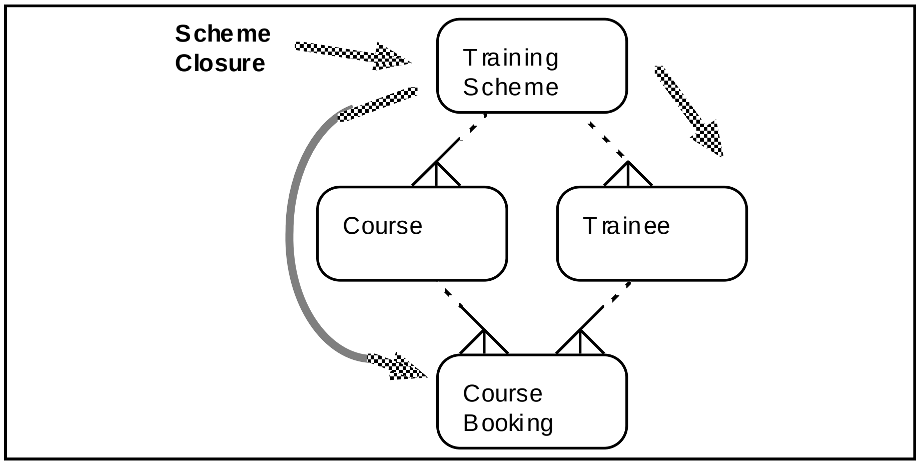
The diamond shape gives another way for an event to hit an object more than once.

The figure below shows a Training Scheme Closure event is broadcast down both the sides of a diamond shape. The event may hit the same Course Booking via both routes. Worse, the event may travel further around the V shape.



The resolution is a little more complex than in the case of a V shape.

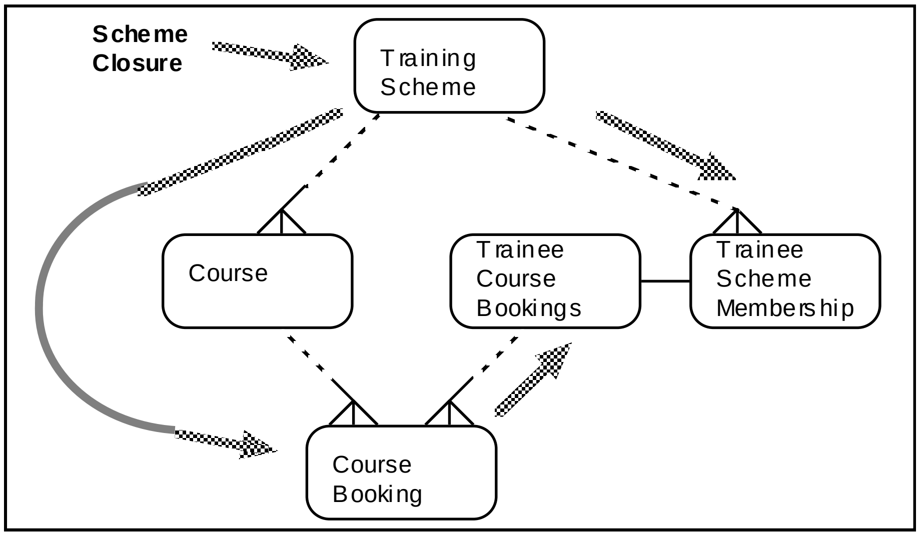
First of all you have to follow an arbitrary rule. When modeling an event that travels down both sides of the diamond, you have to break the circle on one side or the other. The figure below breaks the circle on one side.



Which side to break the circle? You should feel uncomfortable about making an arbitrary decision. You might say that if one relationship is optional at the detail end, then break the circle on that side. The rule fits this example, but other examples still leave you with an arbitrary decision.

And what about the gap in the circle? The event apparently never travels along the relationship from Trainee to Course Booking or vice-versa. So a Trainee gets to hear about the event's effect on the relationship from Training Scheme, but not the same event's effect on the relationship from Course Booking.

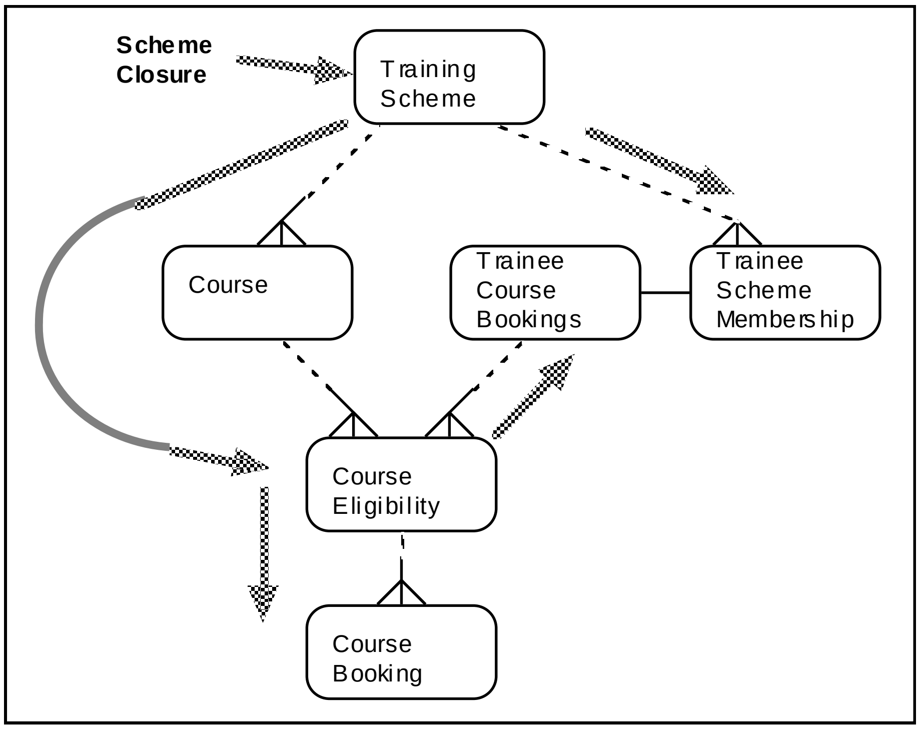
The figure below shows how (though it is not strictly necessary in this case) dividing the Trainee into parallel aspect classes enables you to complete the circle.



So the Trainee entity does get to hear of the event twice, but only once in each parallel aspect class. This device, of splitting a class into parallel aspects, each recording a different effect of the same event, is useful in other situations. See the end of this chapter for another example.

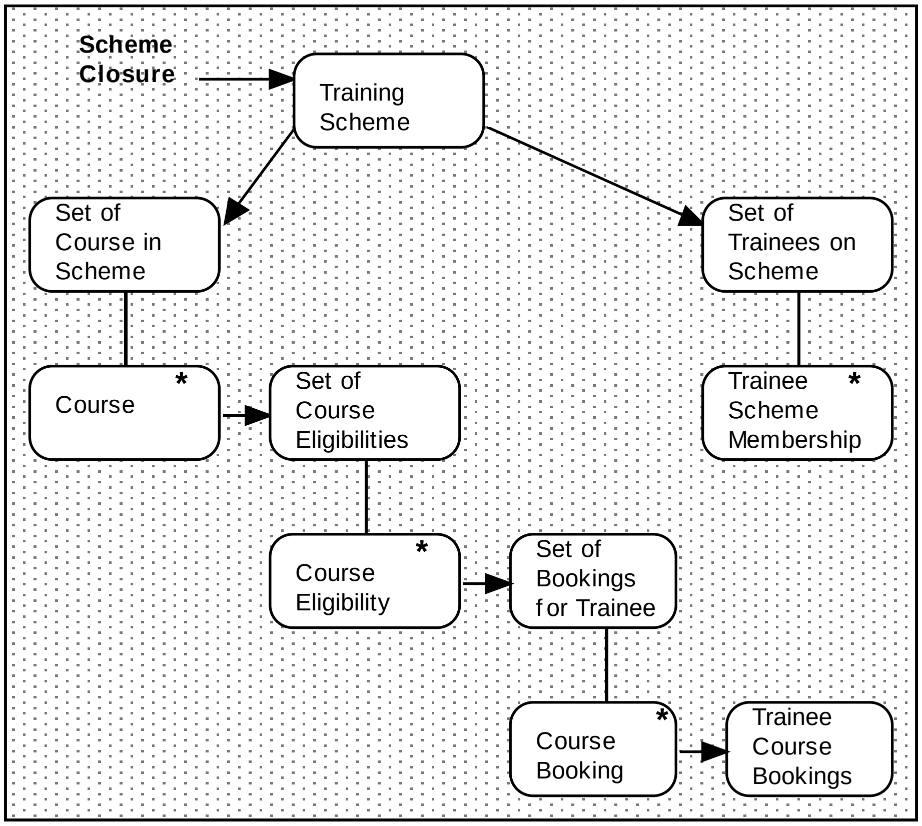
Finally, you may want to extend the V shape as before into a Y shape. Can a Trainee be booked on the same Course more than once? Yes: if they fail the course they may attend it again.

The figure below resolves the V shape into a Y shape as before.

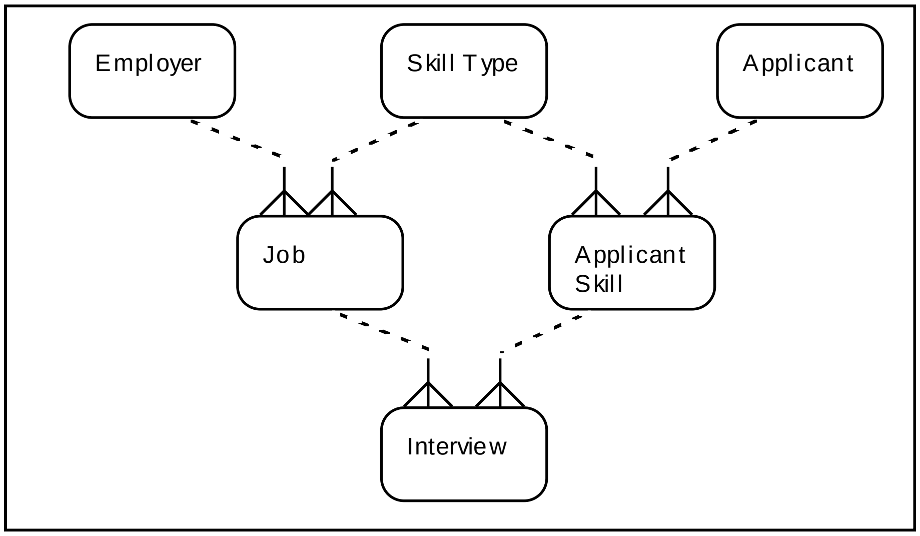


Course Eligibility might be a V shape domain class, that is a class introduced by users to constrain who is allowed to book on a Course. Or it might be a derivable sorting class. Either way, it works to resolve any multiple hit problem via Course Booking.

Drawing the event's access path in the form of an event's Interaction structure, I arrive at the figure below



One more example of double trouble. The figure below is the data model of a recruitment agency application.

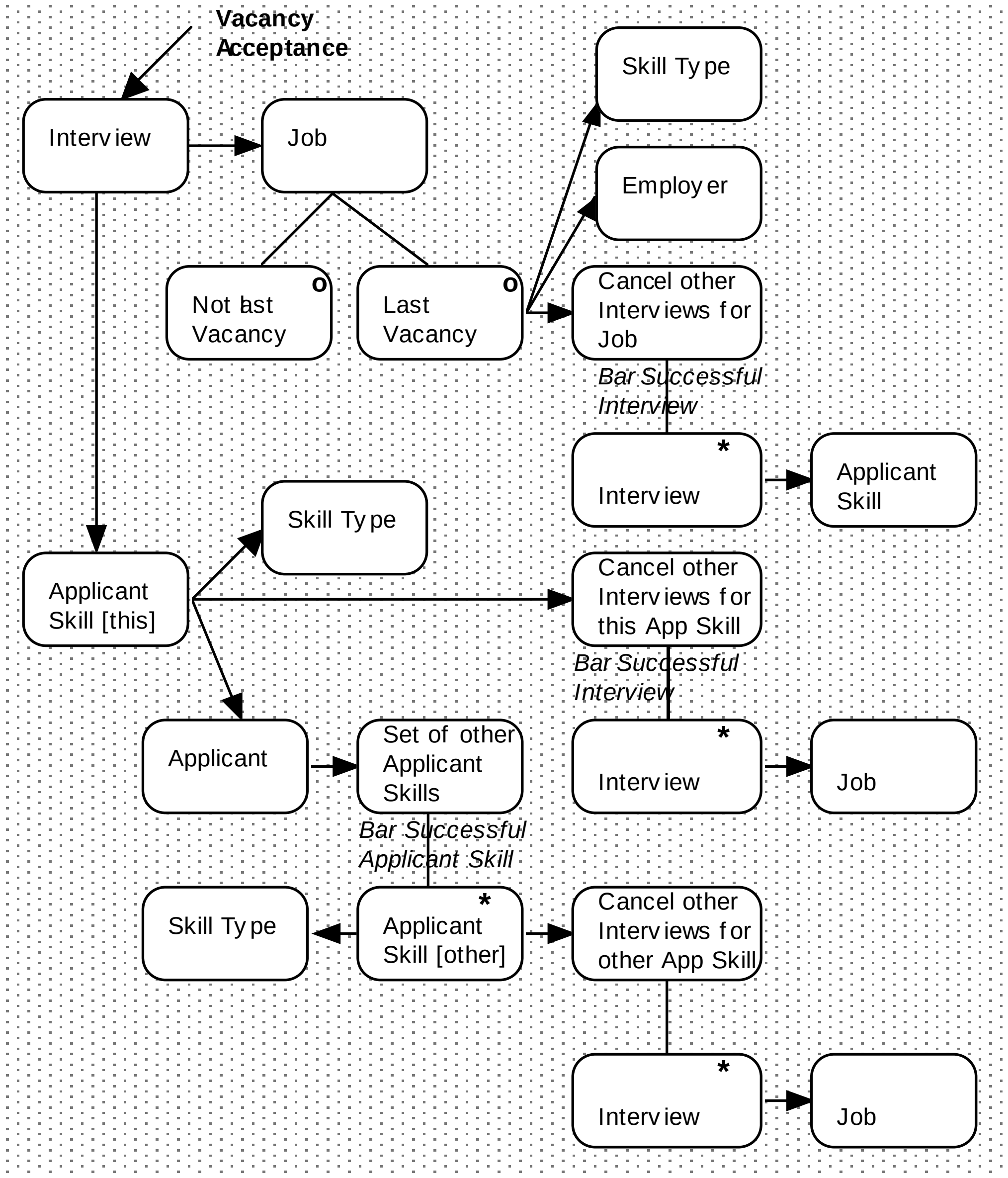


One possible outcome of an Interview is the entry of Vacancy Acceptance event. Draw the Vacancy

Acceptance event’s Interaction structure for the requirement written in text below.

Vacancy Acceptance	
Event parameters	Identify of an Interview
Vacancy Acceptance event response	full details of the Interview that has been successful, list of all other Interviews cancelled for this Applicant, list of all other Interviews cancelled for Job (if this was the last vacancy for the Job)
Vacancy Acceptance event processing	Mark the Applicant for deletion and cancel all other Interviews for that Applicant and if there are no more Vacancies for the Job, mark the Job for deletion and cancel all other Interviews for the Job.

The event’s Interaction structure is extremely complex, as shown below, and it involves several small double-hit problems.



One double-hit problem occurs when you return from Applicant Skill to cancel the other Interviews in that set, you must skip over the successful Interview already being processed, by placing a constraint on the iteration, as shown above.

Notice that Interview is owned by the same Skill Type on both sides of the diamond. So the Vacancy Acceptance event will hit the same Skill Type from two directions, when cutting an Applicant Skill, and cutting a Job, from that Skill Type. The resolution of this multiple hit involves dividing Skill Type into parallel aspects, one for its relationship to Job and one for its relationship to Applicant Skill.

The full case study reveals reuse between events. It turns out that parts of the above event's

Interaction structure are ‘superevents’, common processes also invoked by the events Applicant Withdrawal and Job Withdrawal. See the chapter <Generic events>.

17. PART TWO: ENTITY STATE MACHINES

State and event-oriented ways to represent the results of life history analysis

There are several ways to document a entity state machine. The notations of Mealy, Moore, Harel and Jackson come to mind. This chapter compares and contrasts two styles of entity state machine documentation: state transition diagrams (after Moore) and life history diagrams (after Jackson).

State transition diagrams are state-oriented. Life history diagrams are event-oriented. Both specify constraints on the sequence of events in a entity state machine. Both specify the states an object can be in, the state transitions it can undergo, and the events that cause those state transitions.

The most notably difference is that life history diagrams impose the structure of a regular expression (defined later) over the event effects. This makes them harder to learn, but also give some advantages. Pros and cons are discussed at the end of the chapter.

17.1 Glossary - extended

17.1.1 Event

An event is a discrete, atomic, all-or-nothing business service that updates one or more objects, and perhaps refers to the state of other objects. An event is a discrete, atomic, all-or-nothing happening. It updates one or more Objects, and perhaps refers to the state of other objects.

17.1.2 Interaction structure

An event¹s Interaction structure shows a short-running process, the pattern of objects affected by one event. Database readers: think of events as database transactions. J2EE readers: think of events as session beans.

17.1.3 State

An object’s memory is often called its state. Every object experiences events and responds to them. Transient objects see only one event at a time, and have no memory of past events. Persistent objects see a stream of events over time, and retain a memory of what has happened.

17.1.4 Entity state machine

A entity state machine is the behavior of an object, or a class of objects that share the same behavior. The stream of events affecting a persistent object is describable as a entity state machine. A entity state machine is a fact of life; it happens, whether we document it or not. It is possible for one object to have several parallel entity state machines, but this is not relevant here.

The object-oriented principle of encapsulation means that nobody but an object can see its own memory, its own state. So any event that inspects the state of an object must (by definition) appear in its entity state machine.

17.1.5 Entity state machine diagram

An object¹s entity state machine diagram shows a long-running process, the pattern of events that update or refer to the state of an object over its life. Database readers: think of objects as relations or tables. J2EE readers: think of objects as entity beans.

For any given software system, its set of entity state machine diagrams and its set of Interaction structures are isomorphic views - one can be transformed into the other - though it is very rare to find either view completely documented, outside of classroom case studies.

17.1.6 Entity and event modeling

This is an analysis and design method based on iterative refinement around and between three complementary modeling techniques:

A - draw object relationship model

B - draw objects' life histories as entity state machine diagrams

C - draw events' Interaction structures

The aim is to generate C from B as mechanically as possible, then generate code from C.

17.1.7 Life history analysis

This is the process at B above by which a developer investigates and documents the entity state machines of the co-operating objects that form a system.

Students often ask: Can we avoid doing life history analysis? Can we specify the entity model, then directly specify the events in the form of Interaction structures?

The answer yes, of course you can. However, there are two good and very different reasons to specify the entity state machines:

- analysis - knowledge acquisition about events and their effects on objects
- design - specification of rules for designers and programmers

17.1.8 Regular expression

The pattern of events in a entity state machines can always been shown using the concept of a formal grammar known as a regular expression.

A regular expression is a hierarchical structure composed of sequence, selection and iteration components.

The notation used below imposes a regular expression over the event effects. Components in sequence are drawn from left to right; * marks an iterated component; o marks a selected option.

If you don't have a CASE tool to help you draw entity state machine diagrams, you can represent the information in a table as shown below, where sequence is shown top to bottom.

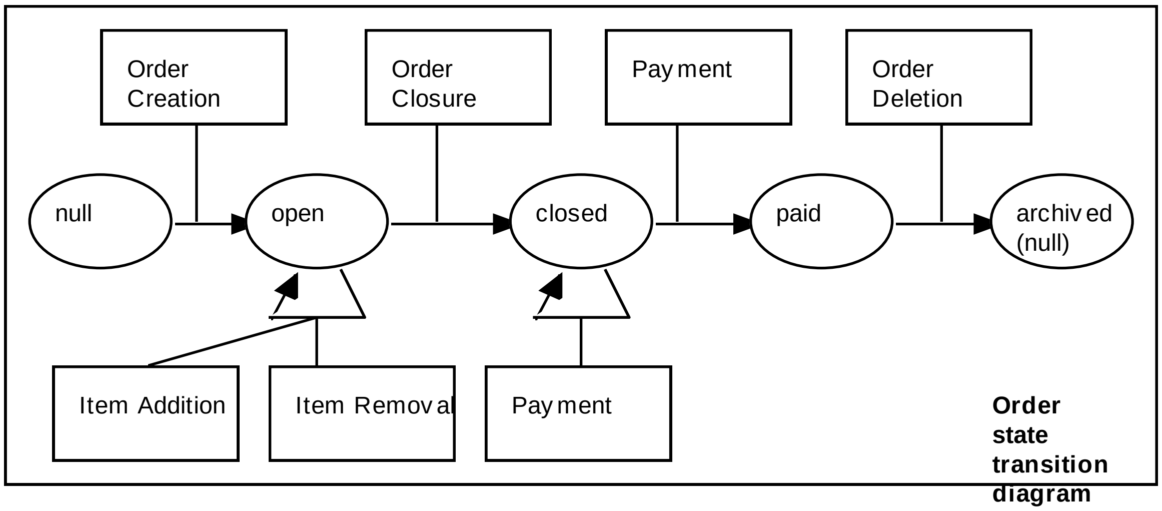
ENTITY STATE MACHINE: Marriage		**Post conditions**
Wedding		StateVariable = active
Married Life	***** Anniversary	StateVariable = active
End of Marriage	**o--** Divorce	StateVariable = historic
	o-- Death	StateVariable = historic
Deletion of Marriage Record		

Feel free to redraw the illustrations in UML and send them to me;-)

17.2 Order processing example

17.2.1 State transition diagram variant

A state transition diagram is state-oriented. The model builders starts by drawing states as ovals, then adds events as labels on arrows that pass between states, or cycle around the same state. Some notations enclose the event names in boxes, and I have done that below to make comparison between notations easier.



Order state transition diagram

17.2.2 State variable values

You might number the state variable values. However, it is better to provide meaningful text names for the values, since these are useful in displaying error messages at the user interface.

An Order in the 'open' state may be extended with new Order Items. An Order in the 'closed' state may no longer be extended with new Order Items, but can now be paid for. An Order in the 'paid' state is complete, has reached the end of its natural life.

A common and reasonable convention is to write 'archived' in the final state. Truly, the final state must be null, the same as it was before the object was created. An Order in the 'archived' state has in effect returned to 'null' state; it has no record, the data has been deleted.

17.2.2.1 A state transition diagram excludes invalid events

A Payment event could find an Order in one of the states 'open' or 'paid'. But in these states the event would be rejected, so it does not occur in the entity state machine and cannot be documented in the state transition diagram.

17.2.2.2 A state transition diagram may include non-update effects of events

The concept of state is wider than the value of a state variable. A Payment event could find an Order that is ‘closed’ in one of these two states:

- amount-owed greater than the amount on this payment
- amount-owed not greater than the amount on this payment

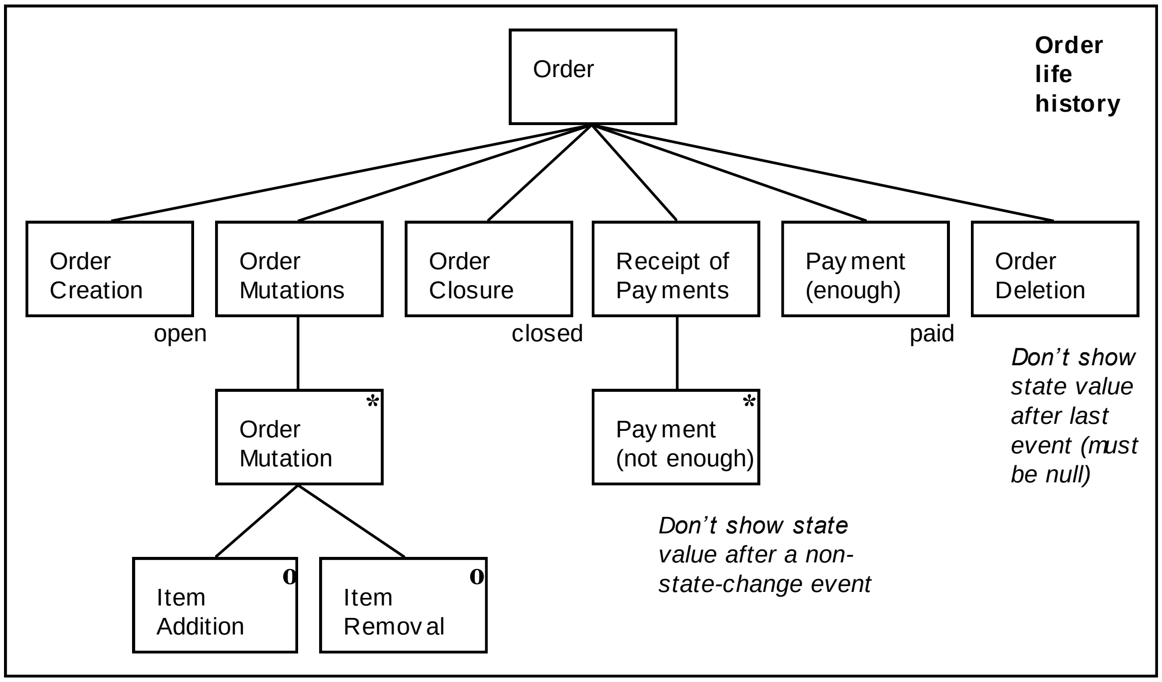
In both these states the event will be accepted and processed, so it does occur in the entity state machine and can be documented in the state transition diagram. It appears twice because it has a different effect depending on the state of the object.

17.2.3 Life history diagram variant

A life history diagram is event-oriented. The model builder starts by drawing the pattern of events, using the concepts of a formal grammar known as a regular expression. A regular expression is a hierarchical structure composed of sequence, selection and iteration components.

The notation imposes a regular expression over the event effects. Components in sequence are drawn from left to right; * marks an iterated component; o marks a selected option.

The model builder adds state variable values as labels on those event effects that alter the state variable of the object.



Some events advance the state variable; some do not. In a state transition diagram, the latter can be shown on arrows that loop back to the state they start from. In a Jackson strucure, they are shown under an iteration. E.g. See Payment (not enough) above.

The corollary is that events that do not advance the state variable should be shown under an iteration or an iterated selection in a Jackson structure. But I won¹t pursue the implications of this here.

If you don’t have a CASE tool to help you draw Jackson structures, you can represent the same information in a table as shown below, where sequence is shown top to bottom.

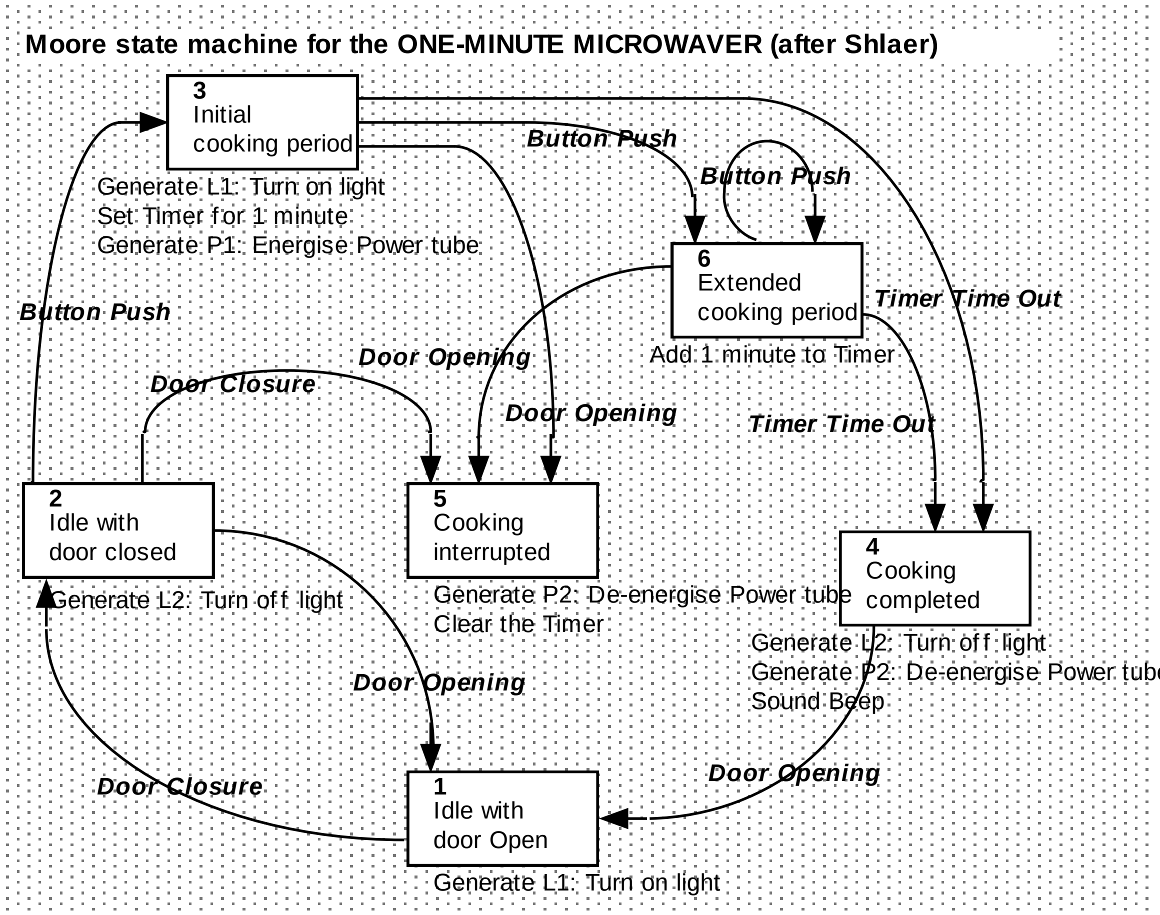
ENTITY STATE MACHINE: Order			**Post conditions**
Order Creation			StateVariable = open
Order Mutations	***** Order Mutation	**o--** Item Addition	
		o-- Item Removal	
Order Closure			StateVariable = closed
Receipt of Payments	* Payment (not enough)		
Payment (enough)			StateVariable = paid
Order Deletion			

17.3 Microwave oven example

The second example below is borrowed from a chapter by Shlaer (reference in introduction).

17.3.1 State transition diagram variant

It represents an object's behavior as a unstructured network of nodes (states) connected by arrows (events).



It took a long time to arrange this diagram. No tool has a way to tidy up this kind of diagram so that it will reduce the variation between different designers and also leave the diagram in a presentable state.

17.3.1.1 State variable values

I have both numbered and named the state variable values because, it turns out, there are fewer state variable values in the life history diagram that follows, and the numbers help to show the points of correspondence.

17.3.1.2 Action sequences and derivation rules

In this example, the event arrows are annotated with action sequences. Each action must in fact be implemented by sending a output message. The generation of an output message is a kind of derivation rule. So the action sequences are in effect specifications of what I call in these chapters behavioral derivation rules - they describe what the event does and the state of affairs the event leaves behind it.

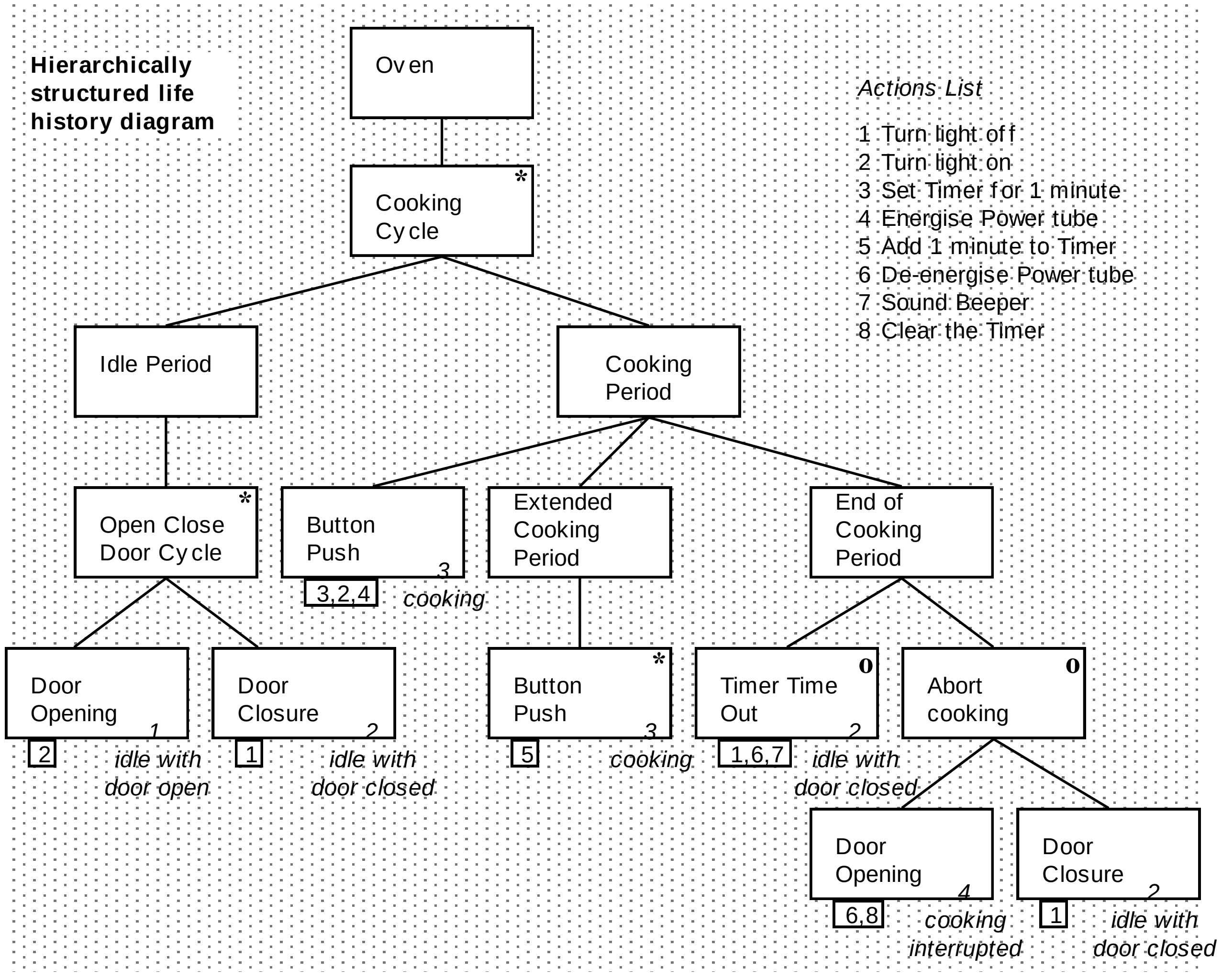
17.3.1.3 Superstates

State transition diagrams can become very complex when lots of event arrows go to the same successor state, arising from many prior states. This is not illustrated here, but I note that to simplify such diagrams, David Harel invented the concept of a ‘super state’ for use in Harel Statecharts. The superstate encloses several particular states. Model builders can draw one arrow from the superstate rather than many from its consituent states.

A super state in a Harel Statechart normally equates to a ‘posit’ in a life history diagram. See furher discussion below.

17.3.2 Life history diagram variant

The diagram below shows the Shlaer’s entity state machine as a hierarchical structure of nodes, with a regular expression imposed over the event effects. To make the two diagrams look more alike, I have not followed the life history diagram convention of qualifying events with different effects at different states by an effect name in brackets.



17.3.2.1 State variable values

A CASE tool can mechanically assign numbers to the state variable values. Any text description must be added by hand . Curiously, the CASE tool used here has generated only four states rather than six. The tool has spotted some redundancy n Shlaer's state transition diagram. Two of the states are duplicated.

17.3.2.2 Pattern recognition

After hierarchical structuring, the diagram turns out to feature one Flip-Flop pattern (door open, door closed) nested within another Flip-Flop pattern (idle, cooking).

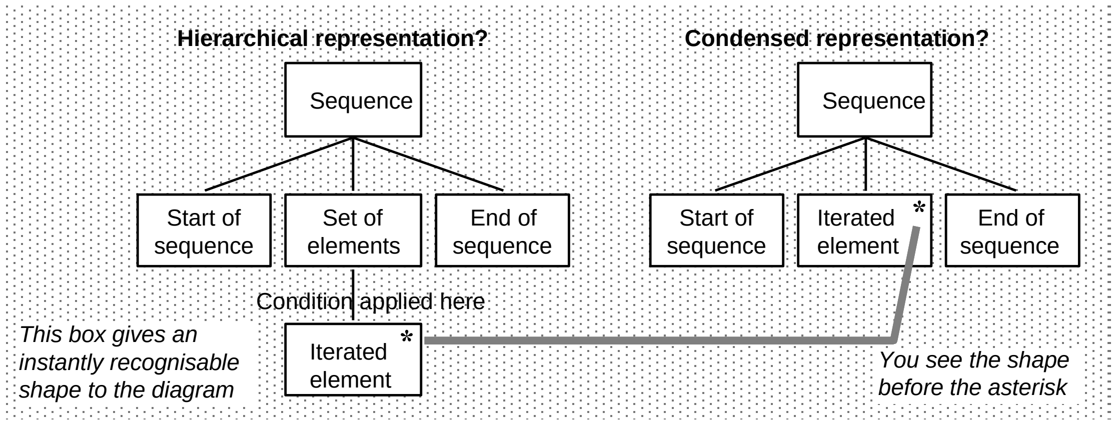
Actually, there is a better and more elegant design, involving three parallel Flip-Flop entity state machines maintain three state variables, but we'll save that for another chapter.

17.4 A false economy in notation

Some think the most important criterion for a diagrammatic notation is economy of representation. Absolutely not! This way leads to a cryptic language suitable only for a mathematical treatise.

People do not work like computers. People naturally introduce a great deal of redundancy into their communication; this gives the listener several different chances to recognise what is being said. The need for some redundancy in graphical notations is supported by twenty years experience of those using the regular expression notation in this chapter.

Some theoreticians have objected to the hierarchically-nested representation of an iteration shown on the left below; they propose the alternative notation on the right.



All the evidence is that practitioners work best with the notation on the left, because the difference between component types is more visually powerful.

(My main complaint about the regular expression notation is that, even after twenty years, I still sometimes fail to spot the difference between a sequence and a selection at first glance. The little circles do not quite mark the difference enough. Putting circle over the corner of the box might work for me.)

17.5 Pros and cons of the life history diagram notation

Many years ago, in1986, the UK government commissioned some work to compare and contrast state transition diagrams with life history diagrams. They settled at that time on using the life history diagram notation in the UK's standard systems analysis and design method. It is worth reviewing some of the arguments in the light of subsequent experience.

17.5.1 Four advantages

17.5.1.1 Amendment

Because a life history diagram is hierarchical, a CASE tool can redraw and tidy up the diagram automatically after any amendment. A diagram will tend to look the same whoever draws it, and it will always be in a presentable state.

17.5.1.2 Completeness

Because a life history diagram is hierarchical, there is space to add detail underneath the event effects. The diagram can easily be annotated with details of the constraints and derivation rules that events apply to attributes and relationships. Thus, a life history diagram is more readily able to combine two elements of an object-oriented design method - a state transition diagram and a class specification.

A CASE tool should enable us to document constraints and derivations rules ‘behind’ an event effect, and show it on the diagram only when we ask.

A CASE tool should provide automated assistance for specifying the constraints and derivation rules by offering menus of attribute names, relationship names and action types, automated optimisation/validation of state variable values, and so on.

17.5.1.3 Validation

The entity life history diagram notation is more obviously comparable with event rules tables. The idea is that every event effect in a entity state machine appears also an event effect in an event rules table. So the notation helps people to validate the entity state machines (described in life history analysis) against the event event rules tables (built in object interaction analysis). A CASE tool can generate event rules tables by reading life history diagrams.

17.5.1.4 Requirements capture

The life history diagram notation helps us to recognize standard patterns, by reducing the variation between diagrams drawn by different designers. These patterns that prompt analysis questions. Patterns assist people to discover and specify the business rules of a system.

While it may be possible to develop a catalogue of state transition diagram patterns, I already have a pattern catalogue of life history diagram patterns. Translating these patterns into the form of state transition diagrams would be an interesting challenge.

17.5.2 Four disadvantages

The four disadvantages of the life history diagram notation are significant.

17.5.2.1 Poor CASE tool support

The validation advantage above has not turned out to be so helpful in practice. People do not normally specify entity state machines and event rules tables to the level of completeness that is necessary to achieve thorough cross-checking and validation.

Why? The work is time consuming and no current CASE tools support the cross-validation as well as they should. Don't blame CASE tool vendors, blame customers for lack of understanding and vision, and for not pressuring the vendors to provide the automated assistance that they need to do their job efficiently.

17.5.2.2 Ignorance of the posit-admit-quit technique

I noted that a super state in a Harel Statechart normally equates to a 'posit' in a life history diagram. It is always possible, but sometimes very clumsy, to represent a entity state machine as a wholly hierarchical regular expression. Sometimes there is a 'recognition problem' and you need to quit from one part of the structure (the posit) to another (the admit).

Michael: So with "posit, admit, quit", you abandon the well-structured discipline of drawing regular expressions?

Graham: Yes and no. The technique is applied on top of a well-structured diagram, in a disciplined way, such that patterns of Posit. Admit and Quit can be recognised.

To prevent clumsiness in drawing life history diagrams you need to understand recognition problems and the posit-admit-quit technique. In the 1980s, the UK government assumed that a large percentage of its systems analysts would have already learned this technique when being trained in structured program design. Sadly, this is no longer true.

Today, I see the fact that there are known posit-admit-quit patterns as being an advantage of the life history diagram notation - but I am lonely in this view.

17.5.2.3 Limited exchange of specifications between CASE tools

A CASE tool should be able to translate a life history diagram into a state transition diagram. However, it cannot do the reverse without a very deep understanding of posit-admit-quit patterns. No CASE tool has this understanding.

17.5.2.4 Longer learning curve

There is much more to life history analysis than notations. People need to spend two to three days working on case studies to get to grips with life history analysis, to grasp how life history diagrams and event rules tables work together as orthogonal views of a single software system specification. People using the life history diagram notation need to spend an additional half a day or so learning the rules of the posit-admit-quit technique.

Why is this a problem? In the 1980s, even a 'basic' certificate in systems analysis required an intensive four-week course. The UK government developed their standard systems analysis and design methodology for people with a year or two's experience after their basic training, and at least two weeks additional training was expected.

Since then there has been a downward pressure on training time. Many employers do not ask for systems analysts and designers to have any professional training in systems analysis and design at all. They ask only for experience in using a specific programming technology - be it Oracle, SAP or C++. Under pressure of time, teachers teach only the symbols of the diagram notations.

Almost nobody teaches 'how' any longer - nobody teaches analysis and design techniques.

> 'Why are professional standards not insisted upon in IT?'
>
> president of the British Computer Society at a conference in March 1997.

Whatever the reason - it is a fact. One reason is the failure of our unversities to teach what professional IT people need to know.

17.5.3 The failure of university courses to teach systems analysis

Some university courses teach 'proper' software engineering - the writing of operating systems, compilers and process control systems. These tend to teach object-oriented methods and dismiss Enterprise Applications as trivial. More practical university courses concentrate on relational theory, because it leads quickly to practical databases and programming languages.

Neither kind of course covers the everyday problems of Enterprise Applications analysts and designers, because these everyday problems fall outside the scope of the theories:

- maintenance of concurrent object types and instances
- maintenance of historical data
- definition of Business Rules and constraints (beyond referential integrity)

17.6 Analysis by pattern recognition

To date, many people find the entity state machine dimension of the Business Rules model is the most obscure and difficult. But it turns out that life history analysis is especially valuable in requirements elicitation, because there are many generative patterns in life history diagrams that make you ask questions about the users requirements. *You do have to ask and answer these questions, whether you document the answers in the form of life history diagrams or not.*

Patterns are recognizable shapes that assist analysis and design. Most people are mostly interested in entity models. There are indeed many patterns to be found in entity models, but to end this chapter, here is a simple pattern in a more obscure face of the modeling cube.

Flip-Flop is a pattern that is especially common in process control systems, where a machine must be turned on and off in response to some received events.

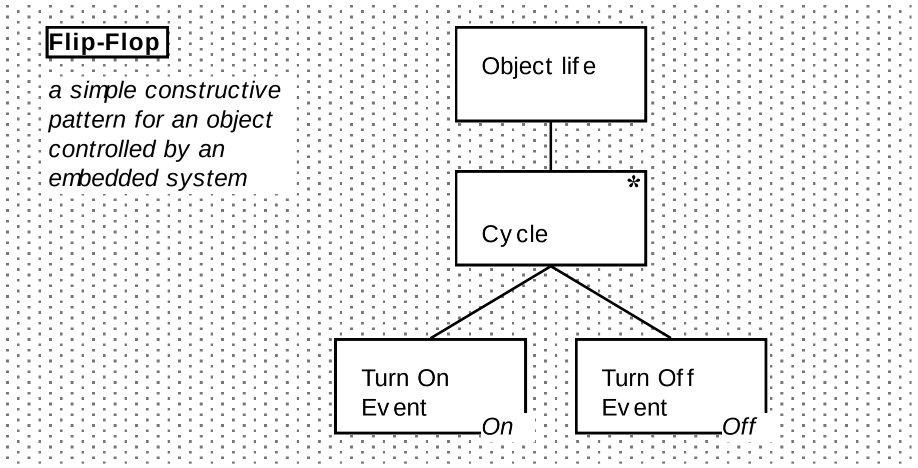

Flip-Flop
a simple constructive pattern for an object controlled by an embedded system
Object life
Cycle
*
Turn On Event
On
Turn Off Event
Off

Fig. 2f

It is easier to recognise standard patterns if you use a structured notation for life history diagrams.

A constructive pattern is one you can reuse and build upon. Flip-Flop is a constructive pattern that is especially common in process control systems, where a machine must be turned on and off in response to some received events. See the next chapter for further discussion.

A generative pattern is one that leads you to ask questions and refine the design. Flip-Flop is a generative pattern. E.g. you should ask of any iteration: What stops it? See the next chapter for the resulting transformation.

17.7 Conclusions and remarks

Many notations can be used to represent entity state machines. In the end, the pros and cons of state transition diagram and life history diagram notations are so well balanced that most people prefer the notation they learn first. Sadly, I do have to choose one notation for this series of chapters.

The RAP group is more interested more in techniques and analysis questions than in notations. There is more to life history analysis than other techniques. This means there is both more to teach and more to learn. The longer learning curve is seen by some as a problem, but should instead be seen as an opportunity. It means there is a real prospect of being able to educate analysts, designers and software engineers beyond the primitive stage that current training courses take them to.

Michael: I'm not sure that software engineers get much training in regular expressions or entity state machines in most courses. Generally, they will get that in a compiler course, but most do not take those courses, and no tie in will be made to Enterprise Applications anyway. Reading 'Shlaer and Mellor', or 'Barker', you find that they dismiss the value of state-transition for Enterprise Applications, concentrating instead on factory process control and so on. This is a shame. Also, most folks working in our business don't even have a computing science degree, let alone a software engineering degree, so they have learned primarily coding languages, with only an off chance that they even had to take a database course to graduate."

Graham: Yes. And the only way to address this problem through professional training.

The ‘more to learn’ I mentioned above is partly composed of analysis patterns. This series of chapters introduces only a few simple patterns. life history analysis based on patterns is exciting because it helps you solve a wider variety of software design problems than other analysis techniques (such as entity relationship modeling, relational data analysis or data flow analysis). And it solves them more fully, providing a more complete path from requirements specification to program code.

Patterns help to compensate for lack of training time. Patterns gathered from experience can be presented in a catalogue, so that people who have learned only the notation can gain deeper insights into what they are doing. Patterns also enable the posit-admit-quit technique to be taught more easily.

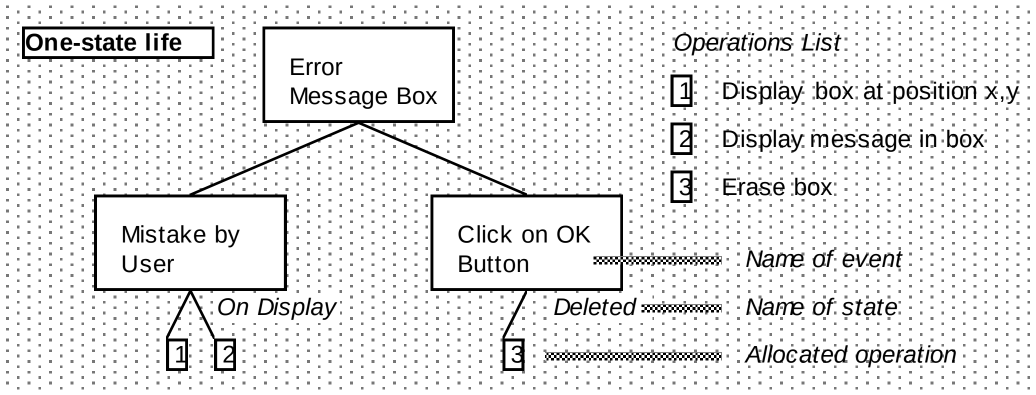
18. Simple entity state machine patterns

This chapter illustrates a nine simple patterns in entity life history analysis, and lists some analysis questions that the patterns prompt you to ask.

- The One-State Life pattern
- The Random Mutation pattern
- The Aggregate Maintenance pattern
- The Life after Death pattern
- The Death by 1000 cuts pattern
- The Business Thread pattern
- The Missing event pattern
- The Sudden or Delayed event pattern
- The Flip-Flop pattern

18.1 The One-State Life pattern

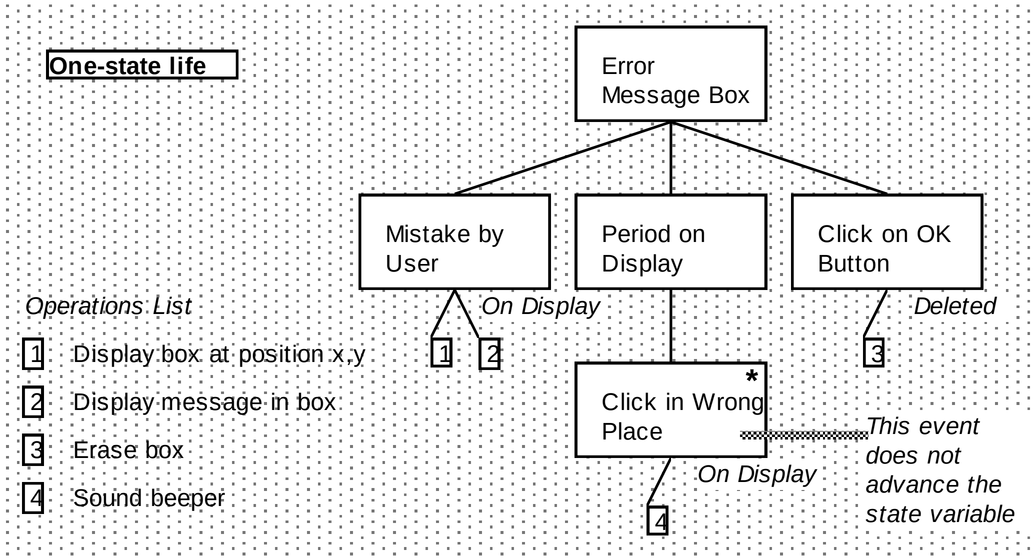
Any persistent object in a software system must first be created and will eventually be deleted. So the basic entity state machine is a sequence of these two events. We'll start with an example drawn from the presentation layer. After you make a mistake, the error message box that pops up your screen probably has a very short life; it is destroyed when you click on the OK button.



People don't normally think of On Display and Deleted as being states. You don't need a state variable to tell you whether an object exists or not, you can test this more directly.

- Q) Given a two-event sequence, are any other events constrained to happen between the two events?

You often discover a non-state changing iteration of events. E.g. Users may repeatedly 'click' in the wrong part of the screen, causing the system to sound a warning beep.

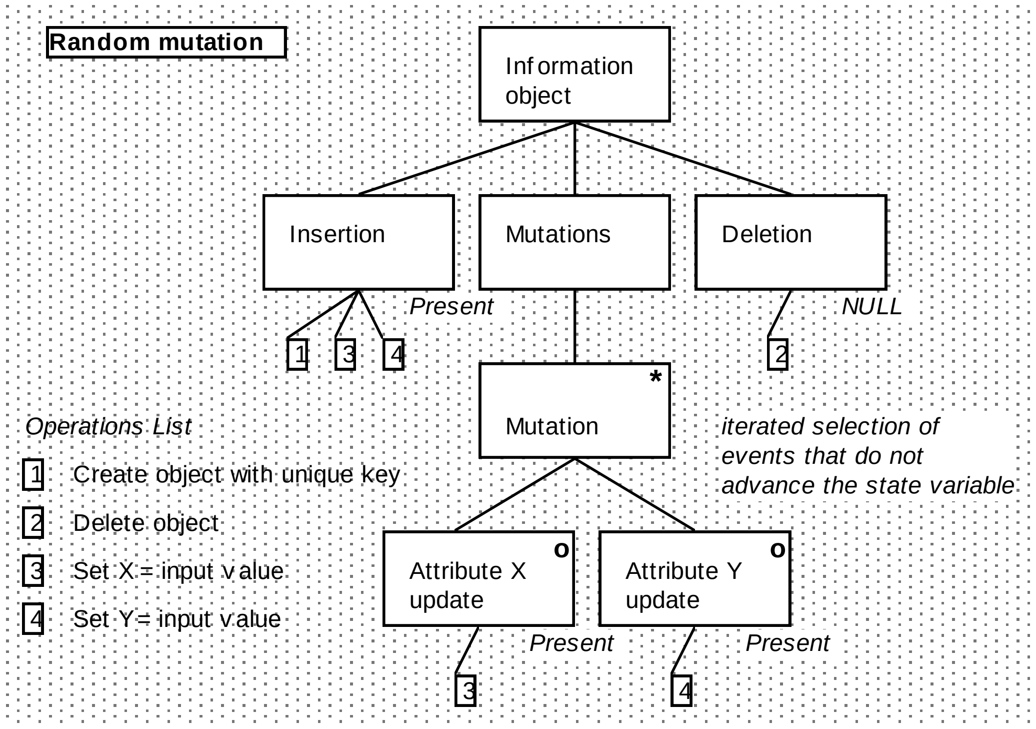


The rule is: state before an iteration = state after each iterated component. So a Click in Wrong Place event leaves the object exactly as it found it, in the state ‘On Display’. So the object still has a one-state life. Again, you do not store the state variable of an object like this, you only test whether it exists or not.

18.2 The Random Mutation pattern

In the business services and data services layers, information objects are created and deleted to reflect what is happening to entities in the real world.

The simplest kind of information object experiences nothing in between creation and deletion, but a random mixture of events that replace the values of different attributes. You may show this as an iteration selection in the middle of the One-State Life Pattern.



State variable values have been optimised thus:

- unify the state before an iteration with that after each iterated component
- unify the states after events which end options under a selection.

These rules mean any kind of Mutation event leaves the state of the information object exactly as it found it - ‘Present’. The only other state variable value is NULL - the state of an object before and after its period of existence. Setting the state variable to NULL on the diagram implies that all the variables of an object instance are deleted at this point, and perhaps that its unique key variable is available for reuse.

18.3 Minimising the documentation of states

Where an event does not update the state variable value, you can leave the state off the diagram, simply assume the prior state is carried forward. You can assume the last event in the entity state machine diagram returns the state to NULL. Then, where an object has only one state, you can discard the state-variable.

Thus, the information object above needs no state variable. You can evaluate any preconditions by asking whether the object (or its primary key) exists in the system.

18.4 The Aggregate Maintenance pattern

The maintenance of an aggregate (be it members of a set, money in a bank balance, or any other cardinal amount) normally looks like the Random Mutation pattern, but with the addition of cardinality constraints. There are several questions. Ask of an aggregate:

- Q) What events increment or decrement the aggregate?

Show these events under the iterated selection.

- Q) Do these events increment or decrement by one at a time? or by a variable amount?

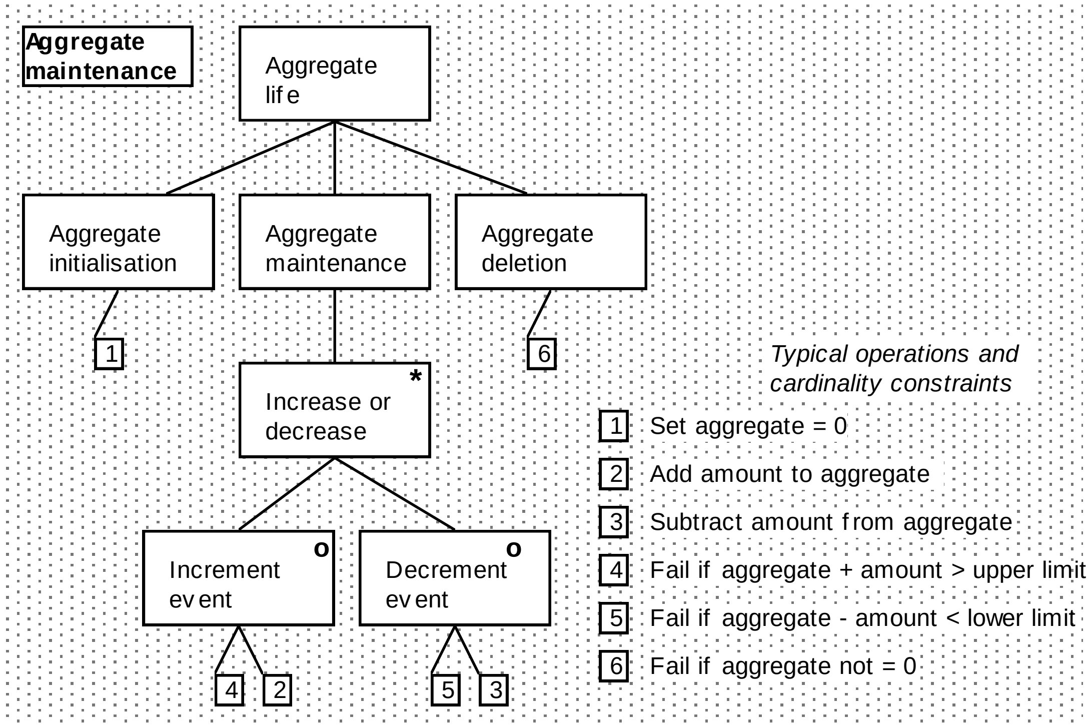
Show this in the operations allocated to the event effects.

- Q) Is there any cardinality constraint on an increment event?

Show this as an operation underneath the event effect of the form: Fail unless value after increment < upper limit value.

- Q) Is there any cardinality constraint on a decrement event?

Show this as an operation underneath the event effect of the form: Fail unless value after decrement > lower limit value.



Aggregate maintenance
Aggregate life
Aggregate initialisation
Aggregate maintenance
Aggregate deletion
1
6
Increase or decrease
*
Increment event
o
Decrement event
o
4
2
5
3
Typical operations and cardinality constraints
1 Set aggregate = 0
2 Add amount to aggregate
3 Subtract amount from aggregate
4 Fail if aggregate + amount > upper limit
5 Fail if aggregate - amount < lower limit
6 Fail if aggregate not = 0

Given the earlier rules, you don’t need a state variable. You might try to represent a limit constraint as a state transition, shown the object cycling through states such as ‘empty’ and ‘full up’, but this leads to overelaborate entity state machines, so don’t do it.

- Q) Is there any sequential constraint between the increment and decrement events?

Do you have to make a deposit before a withdrawal? This sequential constraint belongs in the entity state machine not of the parent class that maintains the aggregate, but of a child class that represents an individual item. If there is no such child class, because users don’t need to track individual items, then the constraint can only appear as cardinality constraints in the entity state machine of the Aggregate Maintenance class, as mentioned above.

18.5 The Life after Death pattern

Many popular implementation technologies, visual programming environments, database application generators and CASE tools are built on the assumption that:

- most if not all information objects have a one-state life, and are deleted on death
- few business constraints require you to test the state of objects
- automatic referential integrity rules will be enough.

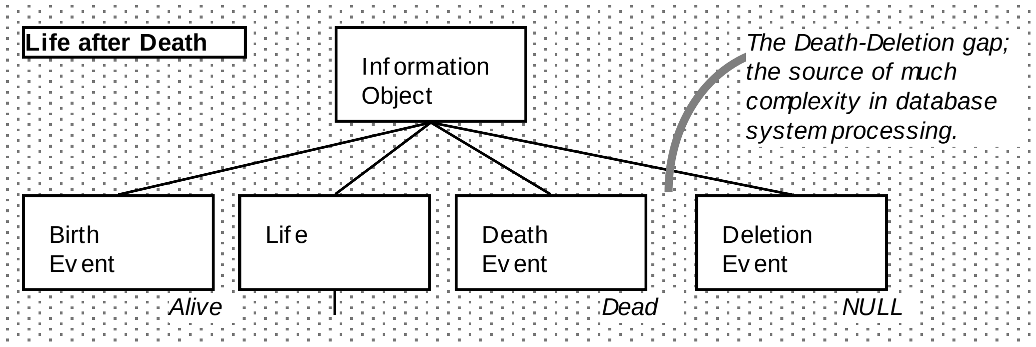
Most database management systems can automate the application of referential integrity rules. These work by testing whether an information object exists or not. If all information objects have one-state lives of the kind shown so for, then 'referential integrity rules' will give you much of what you need to constrain the database.

The vendors of database management systems tend to play down state-changes in any case study they develop. They give entity life history analysis little attention in any method they teach. The trouble is: the four assumptions don't usually hold for the 'interesting part' of the system. Most information objects do undergo one critical state transition - from live to dead.

The birth and death of objects is a major feature of Enterprise Applications analysis and design.

- Q) Given a death event: Does the event automatically delete the object from the system?

If no, you will need to specify a Business Thread of at least three events.



It is usual for an information object to 'die' a long time before the decision to delete it from the system. So every interesting object has at least two states, alive and dead. This single state transition is enough to limit the usefulness of some database technologies.

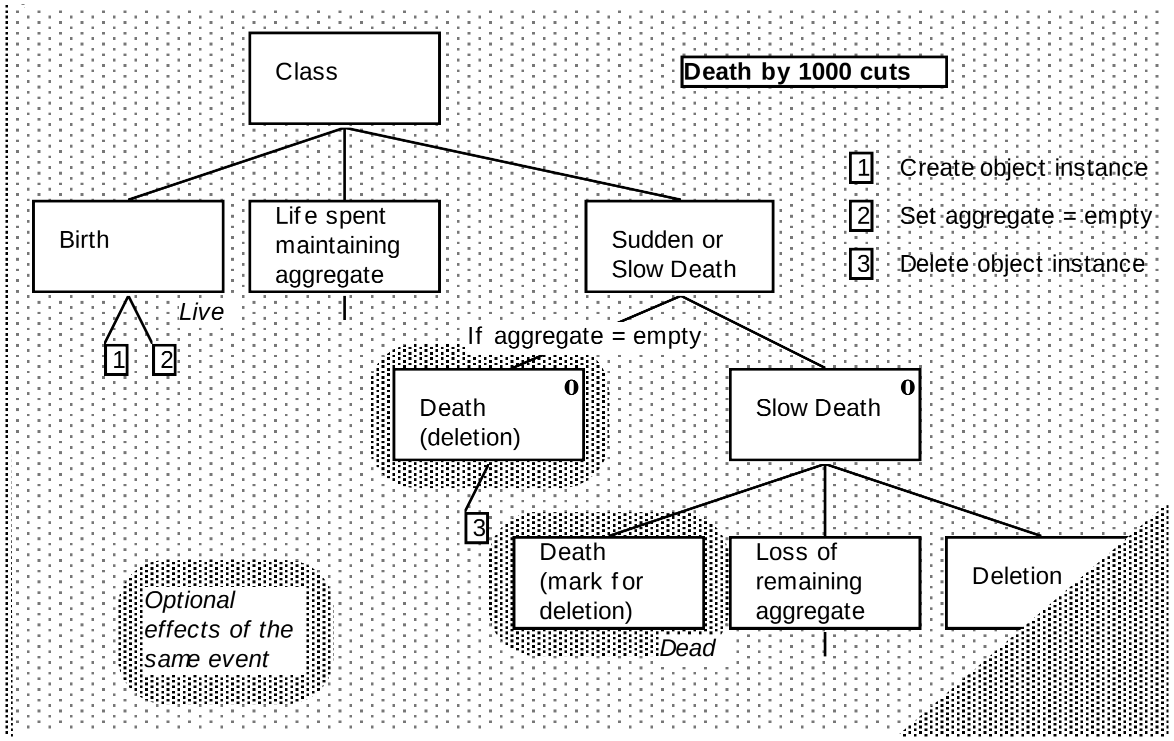
The Death-Deletion gap makes it hard for an application generator to produce the system you want, because it cannot distinguish live objects from dead ones. There are two problems.

- SQL enquiries (of the kind you can easily generate from the database structure) will return a lot of data you aren't interested in. You want to store dead objects for historical analysis, but you don't want them to appear in the screen displays shown to end-users who are trying to do their day-to-day business. The longer the system runs, the more that reports and displays will become cluttered up with irrelevant data.

- Referential integrity rules (of the kind you can easily generate from the database structure) don’t what you want. An application generator may be able to apply ‘restrict’ and ‘cascade’ rules automatically to physical deletion events, but not to logical death events, which is often what you really want. E.g. a customer’s orders will be cancelled on deleting the customer record, but not on the customer’s death.

18.6 The Death by 1000 cuts pattern

Sometimes the rule for a death event hitting an Aggregate Maintenance object is that the object can only be deleted if it has no children left; otherwise it must wait until it loses all children. The Aggregate Maintenance pattern must be extended using the Death by 1000 cuts pattern.



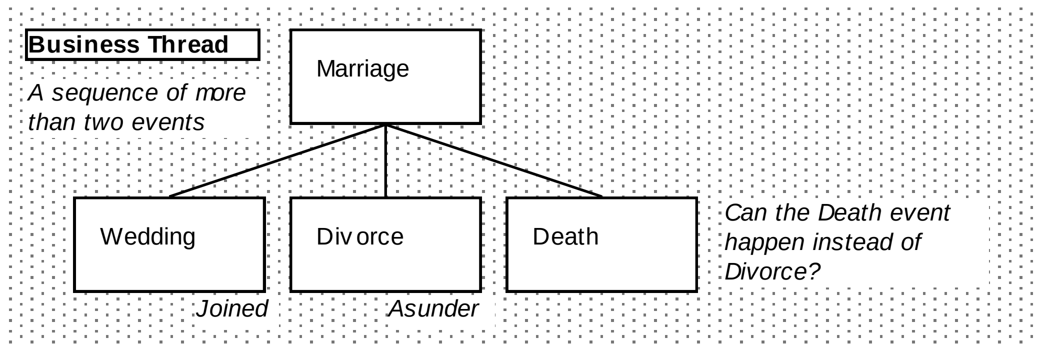
Every selection in a entity state machine is governed by a condition. Normally this condition tests the event type and the condition is apparent from the names of the events in the selected options. In this case however, the same event starts both options of a selection, so an additional condition must be applied.

The object acts as a ‘monitor object’ for a Death event. It decides whether a Death event has one effect or another by inspecting the number of objects remaining in the aggregate. This might mean searching through a set, or it might means testing a derived total attribute; the choice is food for further discussion in the chapter ‘Avoiding double trouble’.

Much the same pattern can be used to model any case where an object may be deleted on its death, but only if a condition applies; otherwise the object must wait until the condition does apply.

18.7 The Business Thread pattern

The Business Thread pattern is any sequence of three events or longer. Suppose you do not delete information about a Marriage until one of the partners dies, then the Marriage entity state machine might look like this.


Business Thread
A sequence of more than two events
Marriage
Wedding
Divorce
Death
Joined
Asunder
Can the Death event happen instead of Divorce?

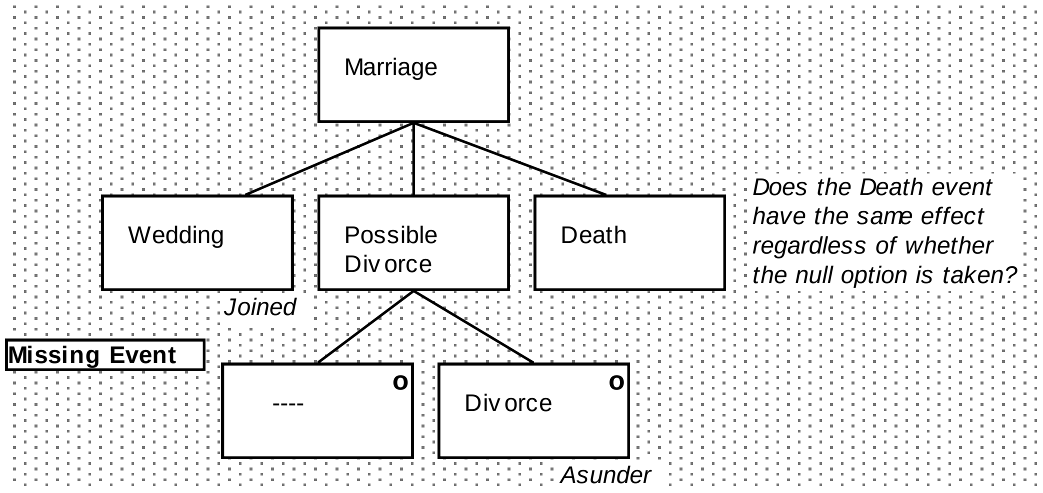
In practice, it is rare to find unbroken sequences of more than two events. The sequence may be broken in various ways. So a simple Business Thread is a generative pattern.

- Q) Given a Business Thread: Can the last event happen without the second-last?

If yes, you may transform the Business Thread using a null option, to form the missing event pattern below.

18.8 The Missing event pattern

When you answer yes, a tool could automatically insert a selection with a null option that enables you to by-pass the second-last event.



Marriage
Wedding
Possible Divorce
Death
Does the Death event have the same effect regardless of whether the null option is taken?
Joined
Missing Event

O
Divorce
O
Asunder

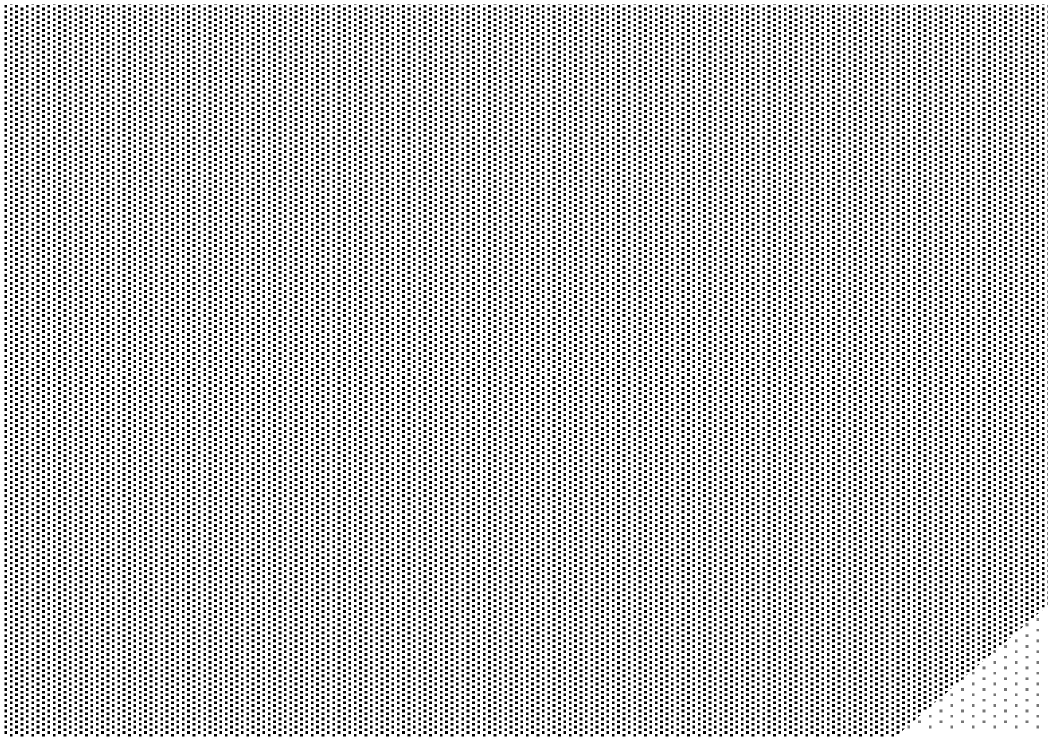
The resultant diagram is itself a generative pattern.

- Q) Given a Missing EVENT: Does the null option determine the effect of the following event?

If yes, then you may transform the Missing event pattern by dividing the effects of the following event, to form the sudden or delayed death pattern below.

18.9 The Sudden or Delayed event pattern

When you answer yes, a tool could automatically divide the following event into two effects and remove the null option, leaving you to add different operations. This diagram below shows two options, one a short way to deletion, the other a longer way.



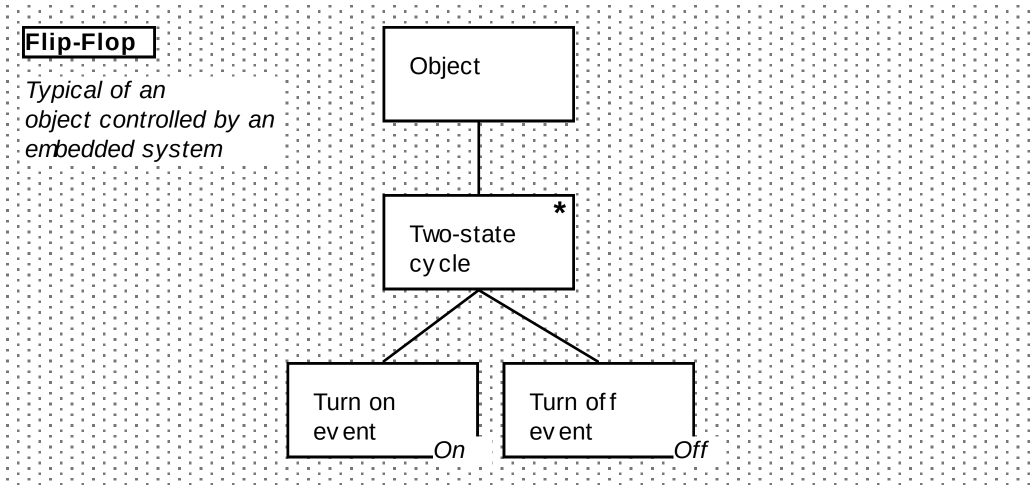
Every selection shown in a entity state machine is governed by a condition. Normally, as here, this condition tests the event type and it is apparent from the names of the events in the selected options. In this case, the selection tests for Death or Divorce.

There is another kind of selection, not so visible in the entity state machine perspective. A Marriage object act as a ‘monitor object’ for a Death event. It decides whether a Death event has one effect or another by testing the current value of its state variable, joined or asunder. This selection is made visible in the event rules table for a Death event.

Where there is a long sequence of events, repeated occurrences of the Sudden or Delayed event pattern may turn the structure into the Daisy Chain pattern discussed in a chapter not included here.

18.10 The Flip-Flop pattern

The flip-flop pattern is a two-state cycle, represented as an iterated sequence.



The Flip-Flop pattern is common in process control systems of the kind often featured in books on object-oriented design. Even though there are many circumstances that may arise to complicate the basic shape, you can build a working process control system using little more than this pattern.

18.10.1 Flip-flop as a generative pattern

A generative pattern is one that prompts you to ask questions and perhaps transform the pattern into a different shape. Given a Flip-Flop pattern you should ask:

- Q) What stops the iteration?

There must be some kind of death event. This may not be a significant question in a process control system, where objects live until the computer is switched off, but it is vitally important in Enterprise Applications, where objects can die within the life span of the system.

Enterprise Application designers tend to be heavily concerned with birth and death events. (Yet case studies in books often gloss over the complex effects of death events.)

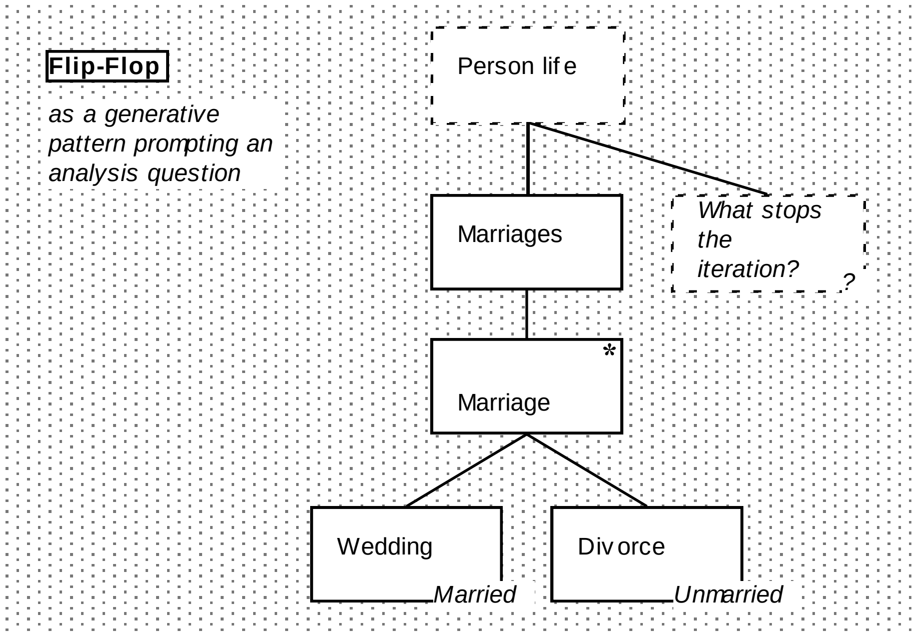

Flip-Flop
as a generative pattern prompting an analysis question
Person life
Marriages
What stops the iteration?
?
*
Marriage
Wedding
Married
Divorce
Unmarried

Fig. 2g

The answer here of course is a person's death event. A cycle-terminating event like this generates further questions. You should ask:

- Q) Does the death event interrupt the last flip-flop cycle?

To represent this, you should choose between one of four patterns shown in a chapter not included here. (They are not shown here because some of these patterns involve 'backtracking', a technique not explained here.)

Then if the object may have children, that is lower-level objects connected by a one-to-many relationship in the entity model, you should ask:

- Q) Should the death the event be broadcast from this object to children of this object?

If the answer is yes, this will involve drawing the Broadcast pattern in an event's Interaction structure (see chapter 'Discrete event models') and further analysis of the effects of the death event on child objects.

Q) Does the flip-flop reveal child objects that users want to keep a history of?

If the answer is yes, then the entity model must be extended.

18.10.2 Flip-flop as a constructive pattern

A constructive pattern is one you can reuse and build upon. Flip-Flop is a constructive pattern that is especially common in process control systems, where a machine must be turned on and off in response to some received events.

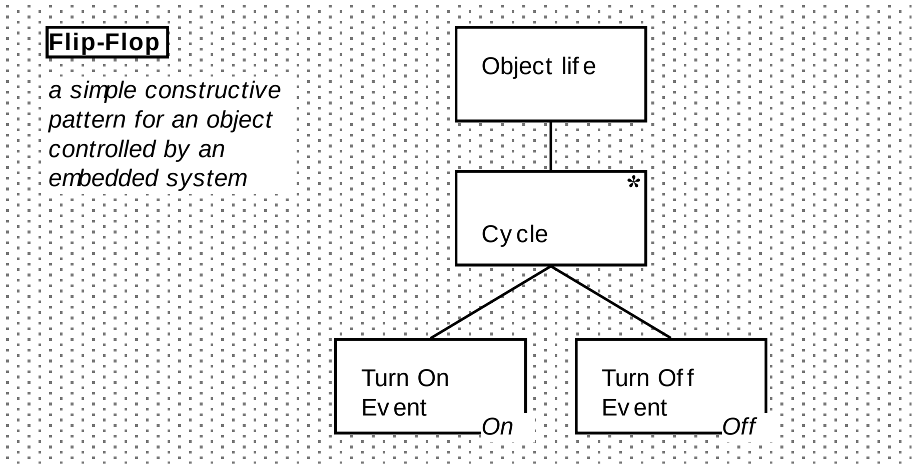
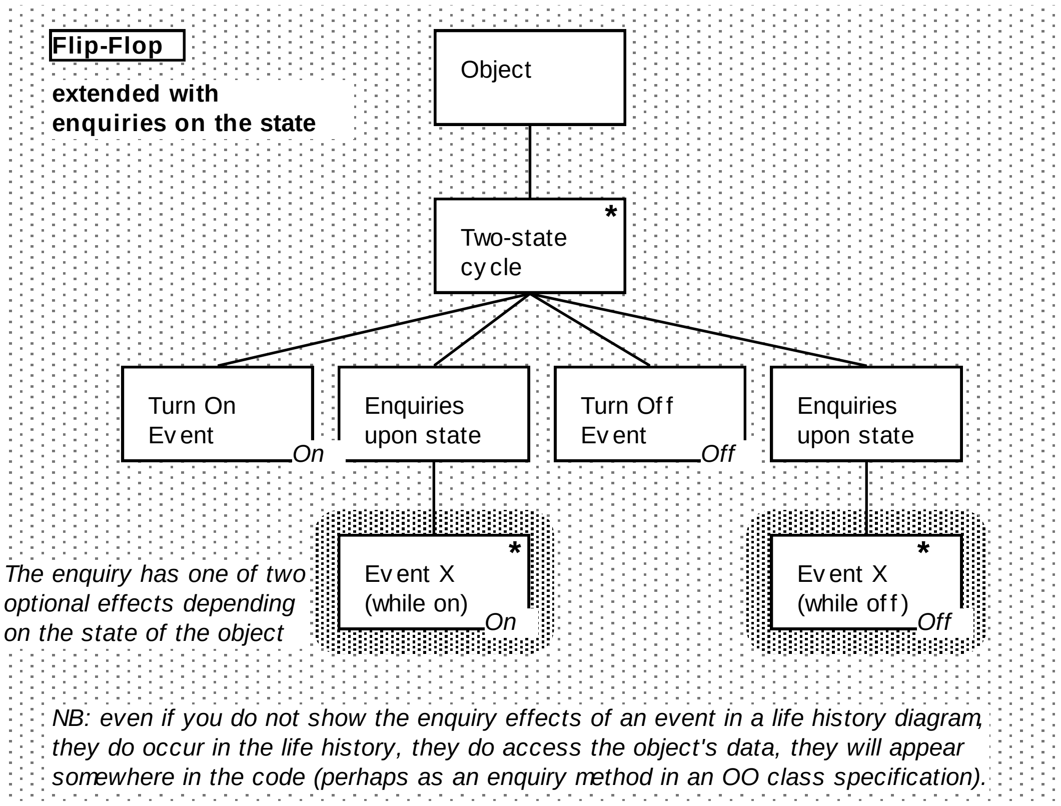

Flip-Flop
a simple constructive pattern for an object controlled by an embedded system
Object life
Cycle
*
Turn On Event
On
Turn Off Event
Off

Fig. 2f

There are many ways to build upon a pattern in a entity state machine diagram. One is to add enquiry effects of events.

I should distinguish events from enquiries. Events update at least one object, enquiries don't. But an event can hit several objects. It may update one object and make an enquiry upon another, perhaps to check it is in a valid state or perhaps to retrieve some data needed for a derivation rule.

The diagram below illustrates how the Flip-Flop pattern might be extended between state-change events to include the enquiry effects of event X under iterations.


Flip-Flop
extended with enquiries on the state
Object
Two-state cycle
*
Turn On Event
On
Enquiries upon state
Turn Off Event
Off
Enquiries upon state
Event X (while on)
*
On
Event X (while off)
*
Off
The enquiry has one of two optional effects depending on the state of the object
NB: even if you do not show the enquiry effects of an event in a life history diagram, they do occur in the life history, they do access the object's data, they will appear somewhere in the code (perhaps as an enquiry method in an OO class specification).

This object acts as a monitor object for event X. It decides whether event X has one effect or another by inspecting the current state of its stored data. The enquiry effects do not advance the state variable because its values have been optimised using the rules given earlier. (???)

To include enquiry effects or not? Whether you should or should not show the enquiry effects of update events in a entity state machine diagram has been debated for nearly thirty years.

Keith Robinson used to teach that an event occurs in the entity state machine of every object it refers to, as well as the entity state machines of every object it updates. In other words, every event effect in an event rules table appears in a entity state machine. He claimed that classroom teaching is simpler and students learn quicker if they follow this rule.

Keith believed that objections to including enquiry effects in practical system documentation would disappear with the advent of adequate CASE tool support for drawing the diagrams (for automatically optimising state variable values, and for generating event's Interaction structures) and with better teaching of how to design the most economical set of entity state machines.

Sadly, Keith's hopes have not been borne out. Later chapters discuss how to omit enquiry effects from entity state machine diagrams without losing the specification information.

18.11 Conclusions

This chapter has summarised some basic entity state machine patterns. To go further I must enter the territory where you have to draw more than one entity state machine to solve the problem, and I look at how entity state machines are co-ordinated in the chapter ‘3-way conceptual modeling’.

19. Behavioral constraints as event sequences

This chapter compares and contrasts different approaches to behavior analysis or life history analysis. It illustrates various levels of completeness to which life history analysis can be taken.

19.1 Clashing paradigms

Two life history analysis techniques have been developed almost independently. The inheritance paradigm emphasises the analysis of hierarchical structures of super and subclasses, and reuse by inheritance. Object-oriented authors such as Booch (1986) and Meyer (1988) specify a software system as a set of inter-related classes, encapsulating processes around abstract data types.

The entity state machine paradigm emphasises the analysis of events input to a software system and the state-changes they trigger in persistent objects. Authors such as Jackson (1975) and Hoare (1985) discussed ways of specifying a software system as a set of cooperating sequential processes or entity state machines.

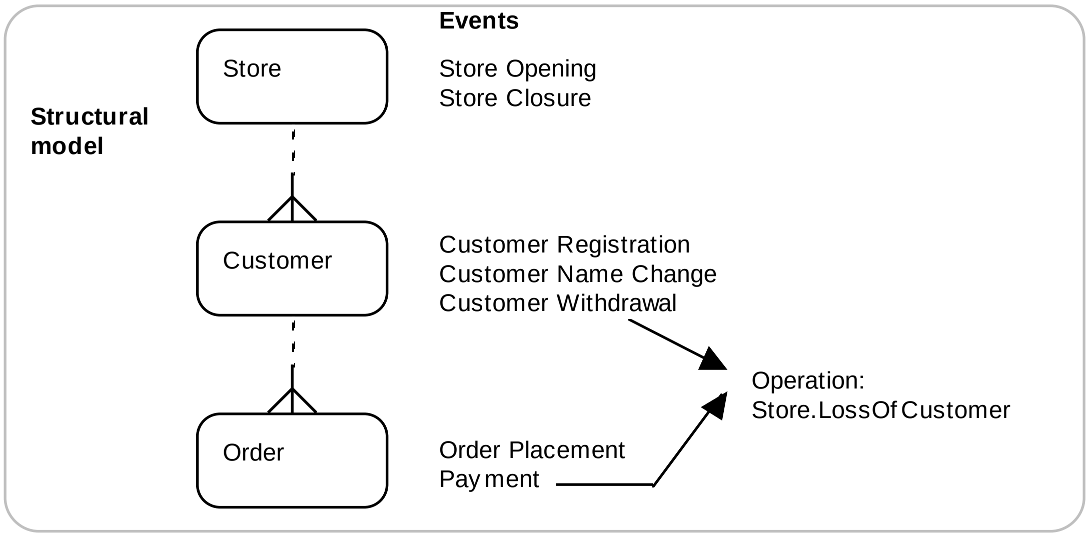
Each paradigm has desirable properties that the other lacks. So can we develop a behavior modeling approach that combines the ideas of Grady Booch and Michael Jackson? an approach that regards cooperating objects as interacting entity state machines?

It might be thought that there is a clash between the OOP notion of a class and the notion of a entity state machine. In fact, the formula OO class = entity state machine does work well for modeling behavior in the Business Rules layer. However there is sometimes a clash between the notion of class in the business rules layer and a class in other layers. You may need to merge or divide classes when designing the user interface or database layers.

At a lower level of granularity, there is sometimes a clash between the OOP notion of an operation and the entity state machine notation of an event effect. This clash is explored in this chapter.

19.2 An entity model

The case study is very simple. The diagram below shows the classes and relationships of the entity model, and lists the main events that affect it.



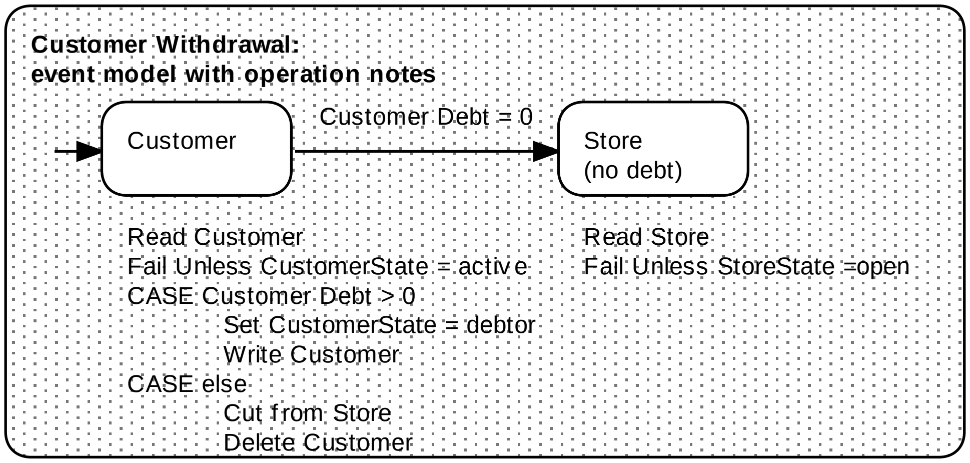
19.3 An event’s Interaction structure

You can specify in an event’s Interaction structure and/or event rules table:

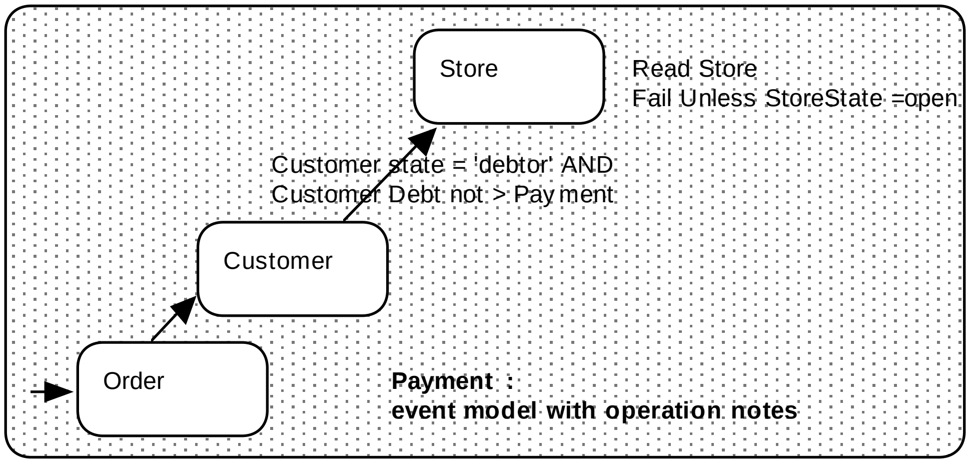
- the objects it hits, and the path by which it finds them
- its behavioral constraints - preconditions testing the states of objects
- its derivation rules - changes to objects’ attributes and relationships

Many of the events are trivial (Customer Registration for example). Two of the most interesting events are Customer Withdrawal and Payment, both of which might delete a Customer. I take the opportunity to illustrate two styles of event’s Interaction structure below.

The Customer Withdrawal event fires up two operations, one in Customer and one in Store. Note that the operation on Customer contains a case statement; it only deletes the Customer if there is no outstanding debt. Note that the event only fires the enquiry operation on Store if this condition is true.



The Payment event fires up an operation in an object in each of three classes. Again, the operation on Customer contains a case statement; it only deletes the Customer if it has been withdrawn and there is no outstanding debt. And note that the event only fires the enquiry operation on Store if the Customer is this condition holds.


Store
Read Store
Fail Unless StoreState =open
Customer state = 'debtor' AND
Customer Debt not > Payment
Customer
Order
Payment :
event model with operation notes

19.3.1 Factoring out a common operation

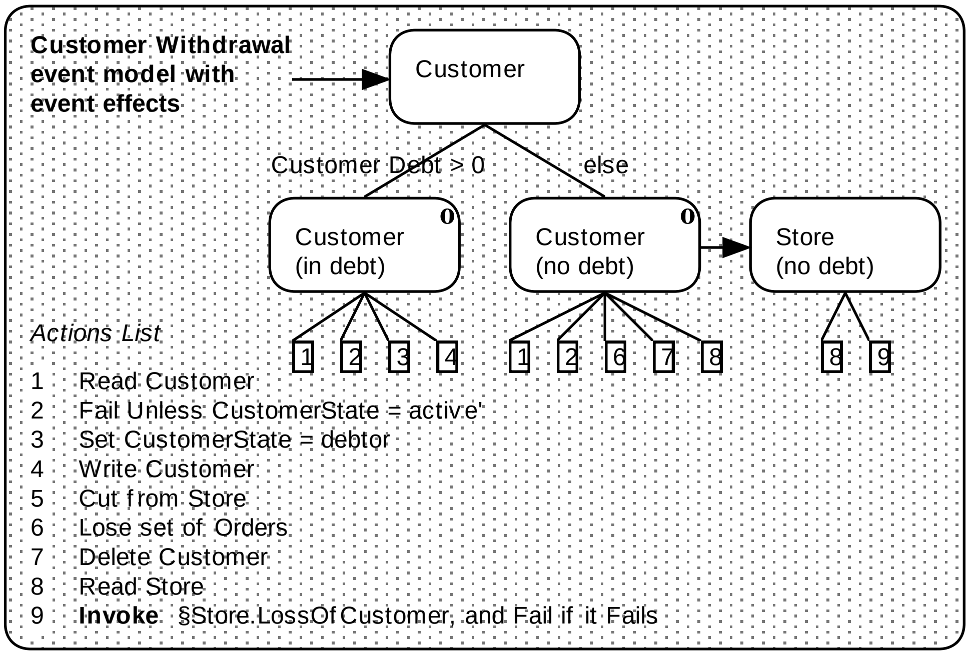
During the analysis, it becomes apparent that the Customer Withdrawal and Payment events share a common action on Customer (Delete) and a common effect on Store. They both check to see the Store is open before they delete the Customer. This can be factored out into an operation called §Store.LossOfCustomer. The operation is a very simple precondition test:

Store.LossOfCustomer

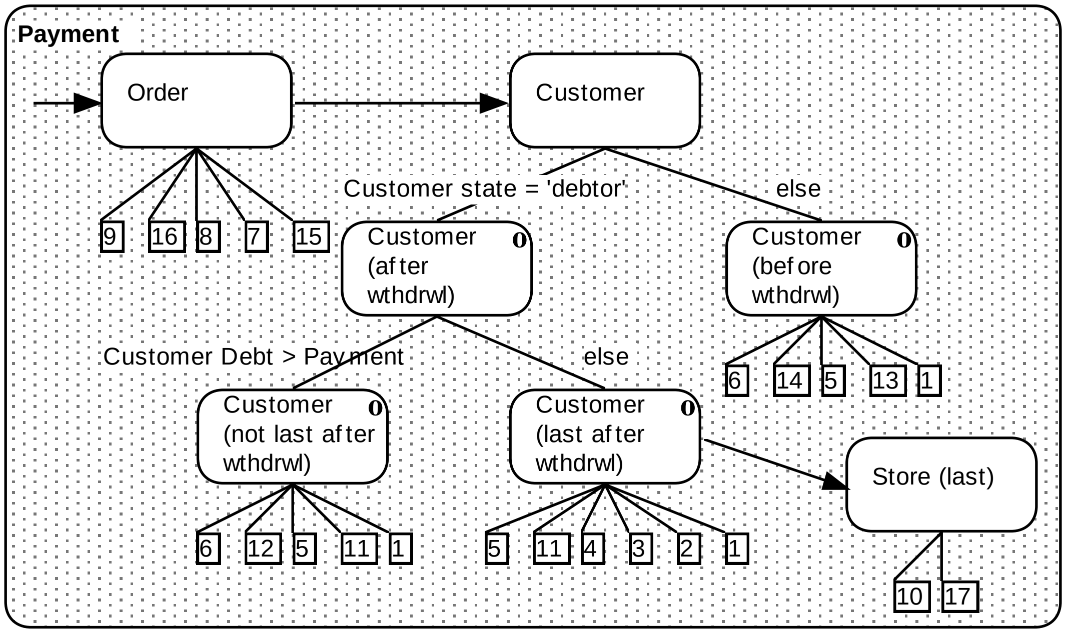
Fail Unless StoreState =open

19.3.2 Showing conditional invocation by correspondence arrows

An alternative representation is to show each event effect explicitly, which enables a more precise correspondence arrow to be drawn.



Note (above and below) how the correspondence arrows show the fact that the Payment event only fires the Store.LossOfCustomer operation if a selection condition applies in the operation on Customer.



19.4 Life histories

In theory, one can complete entity and event-oriented behavior modeling by following the rule that every event effect shown in an event's Interaction structure corresponds to an event effect in a life history diagram. You can work from the event's Interaction structures to the life history diagrams or vice-versa.

Given that all the event effects are documented in event's Interaction structures, a reasonable question is: Do we have to draw the life history diagrams as well?

The answer is that the event's Interaction structures shown above were not in fact completed before the life history analysis. They are only shown first in this chapter for the sake of illustration. Neither view has precedence. In practice one develops both views in parallel.

It is true however that model builders very rarely complete the life history diagrams to the same level of detail as the event's Interaction structures. A more reasonable question is: How can we get the benefits of life history analysis without documenting every entity state machine in full?

First of all, your CASE tool should be doing all the tedious parts of the work for you, generating event's Interaction structures by reading life history diagrams and vice-versa. OK, so your particular CASE tool doesn't actually help you to do this. What can you do by hand?

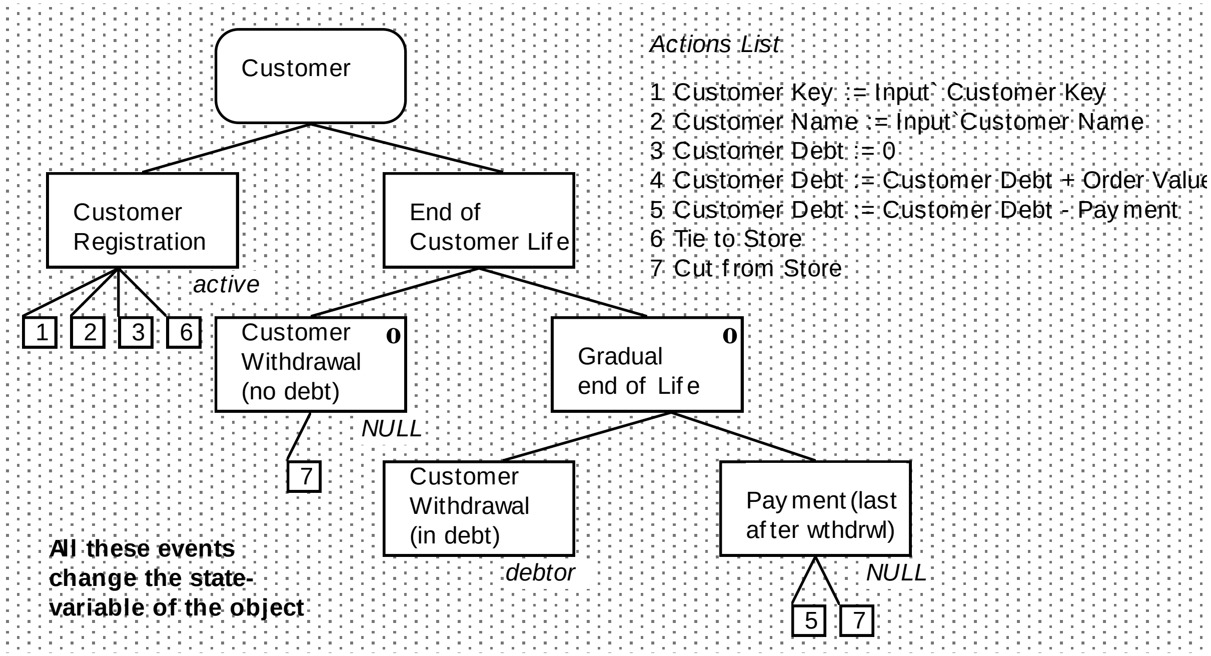
Entity life history analysis is very much easier and more natural if you draw first-cut event's Interaction structures beforehand. Think of life history analysis as a supporting technique for sorting out the business rules and constraints where objects in an event's Interaction structure have state-changes beyond the 'insert' and 'delete'.

Try it this way. Start the event's Interaction structures before life history analysis. Draw an event's Interaction structure to specify the correspondences between all the objects referred to by the event. Assume the default precondition for an object (other than an object created by the event) is 'Fail unless object exists'. Then, only document the entity state machine of an object that has more than one state.

This section describes five different degrees of completeness for the documentation of a entity state machine. Practitioners may decide for themselves at which degree of completeness they will stop.

19.5 1 Show only state-change effects of events

An object's state is composed of specific values for its attributes, relationships and state variable. A minimalist view of life history diagrams is that they are a form of state-transition diagram, showing only those event effects that change an object's state variable. Thus, the entity state machine of Customer might be drawn as below.


Customer
Customer Registration
End of Customer Life
active
1
2
3
6
Customer Withdrawal (no debt)
o
Gradual end of Life
o
NULL
7
Customer Withdrawal (in debt)
Payment (last after wthdrwl)
debtor
NULL
5
7
All these events change the state-variable of the object
Actions List
1 Customer Key := Input` Customer Key
2 Customer Name := Input`Customer Name
3 Customer Debt := 0
4 Customer Debt := Customer Debt + Order Valu
5 Customer Debt := Customer Debt - Payment
6 Tie to Store
7 Cut from Store

- A Customer in the 'active' state may place Orders and pay for them.
- A Customer in the 'debtor' state may no longer place Orders, but can pay for past ones.
- A Customer in the 'NULL' state has no record, the data has been deleted.

You can assign numerical values to successive states in the entity state machine. However you will usually want to provide meaningful text names for the states, since these are useful in displaying error messages at the user interface.

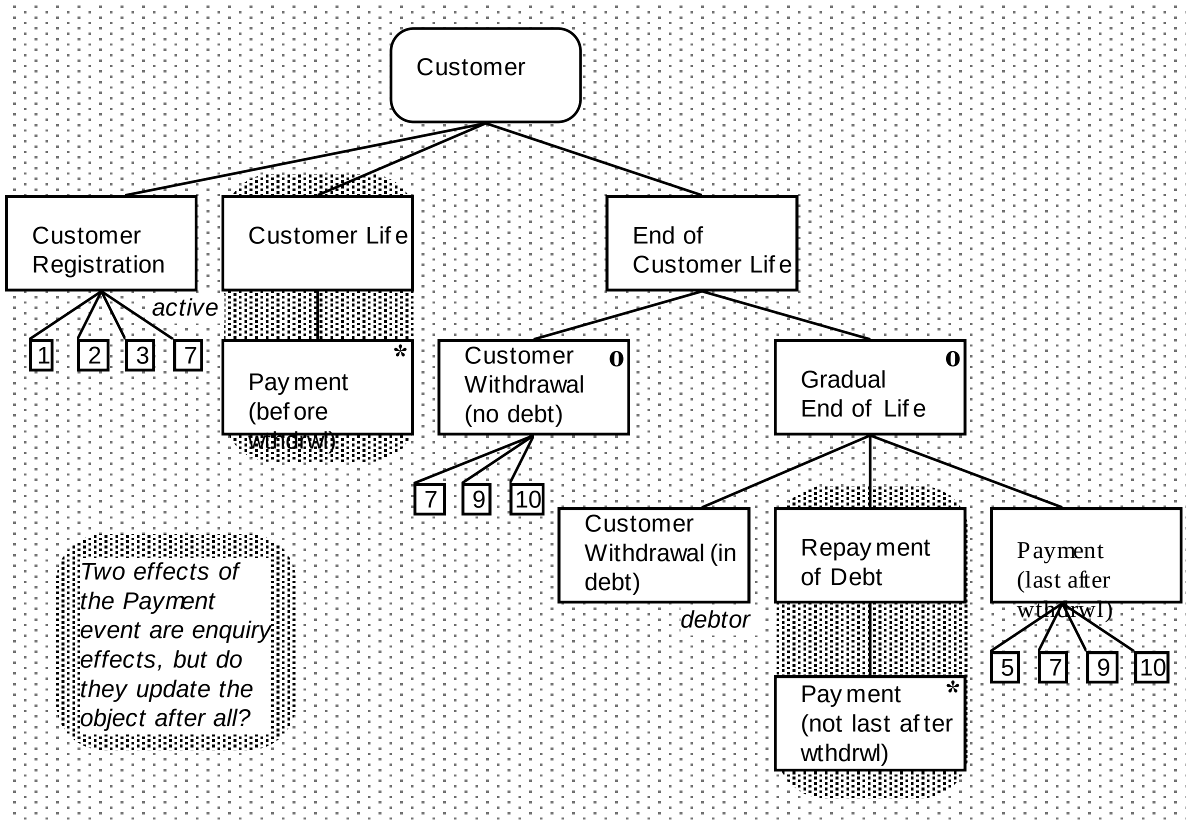
You only need to show state variable values on a entity state machine diagram under events that change the value, and you can suppress the NULL value from under the last event.

19.6 2 Show other effects of state-change events (in the same entity state machine)

The Payment event's Interaction structure earlier showed that the Payment could find the Customer in one of three states:

- active
- debtor, and debt greater than this payment
- debtor, and debt not greater than this payment.

Only the last of these has been shown as a state-variable-updating effect in the entity state machine so far. What about the 'null effects' or 'enquiry effects' of the Payment event? You could show them in the object-oriented entity state machine as below.



Customer
Customer Registration
Customer Life
End of Customer Life
active
1
2
3
7
*
Payment (before wthdrwl)
Customer Withdrawal (no debt)
o
Gradual End of Life
o
7
9
10
Customer Withdrawal (in debt)
debtor
Repayment of Debt
Payment (last after wthdrwl)
5
7
9
10
*
Payment (not last after wthdrwl)
Two effects of the Payment event are enquiry effects, but do they update the object after all?

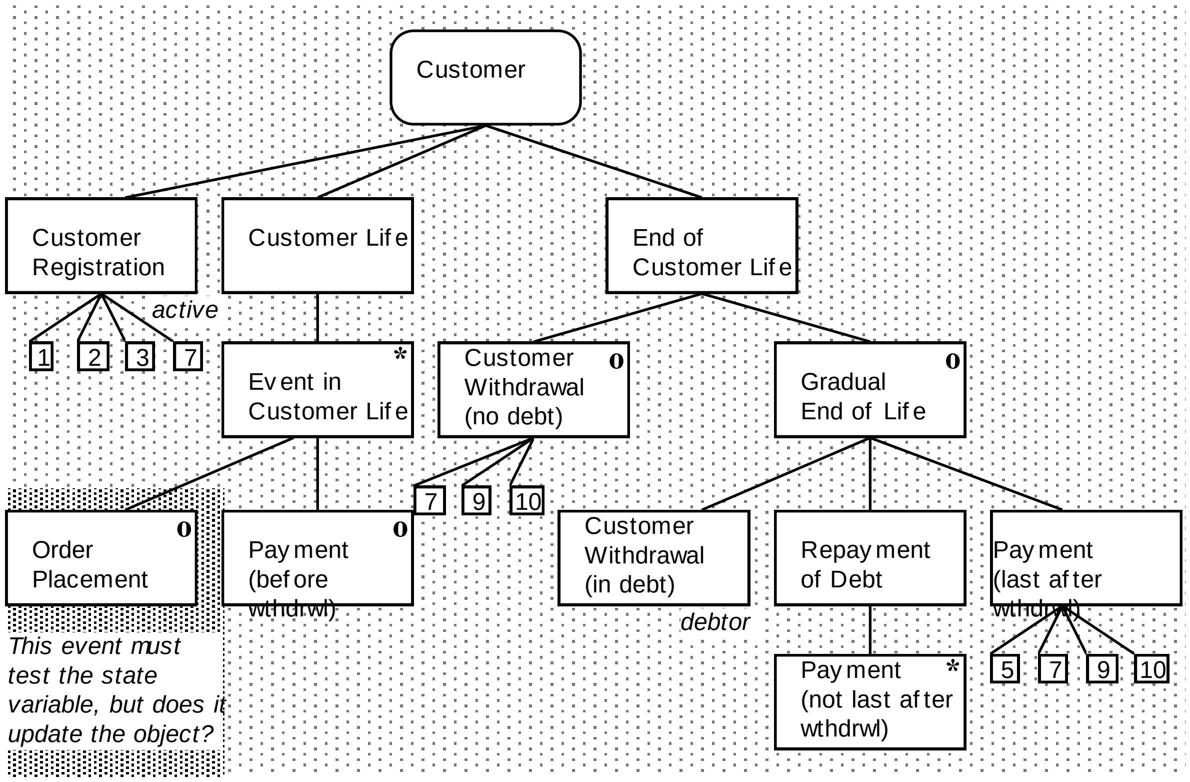
By showing the Payment event before and after Customer Withdrawal, this diagram says that an Payment is valid at any time. However, the Payments, being shown under simple iterations, do not advance the state variable.

Showing the enquiry effects of a state-change event in a entity state machine gives three benefits. It prompts you to:

- ask if the enquiry effects update the object’s state (they do in our example)
- bring the life history diagrams into closer correspondence with the event’s Interaction structures
- develop a more complete object-oriented specification.

19.7 3 Add state-testing events

What about events that do not alter the state of an object, but must test its state variable? A behavioral constraint on the Order Placement event is that the Customer must not have been withrdawn yet. You can show this precondition by placing the Order Placement event in the Customer entity state machine between Customer Registration and Customer Withdrawal, where it may happen any number of times.



Customer
Customer Registration
Customer Life
End of Customer Life
active
1
2
3
7
Event in Customer Life *
Customer Withdrawal (no debt) o
Gradual End of Life o
7
9
10
Order Placement o
Payment (before wthdrwl) o
Customer Withdrawal (in debt)
Repayment of Debt
Payment (last after wthdrwl)
debtor
Payment (not last after wthdrwl) *
5
7
9
10
This event must test the state variable, but does it update the object?

By showing the Order Placement event before Customer Withdrawal, but not after it, this diagram specifies the rule that an Order Placement is invalid after Customer Withdrawal. Again, there are three advantages of showing such a state-testing event in a entity state machine. It prompts you to:

- ask if the enquiry effects update the object's state (they do in our example)
- bring the life history diagrams into closer correspondence with the event's Interaction structures
- develop a more complete object-oriented specification.

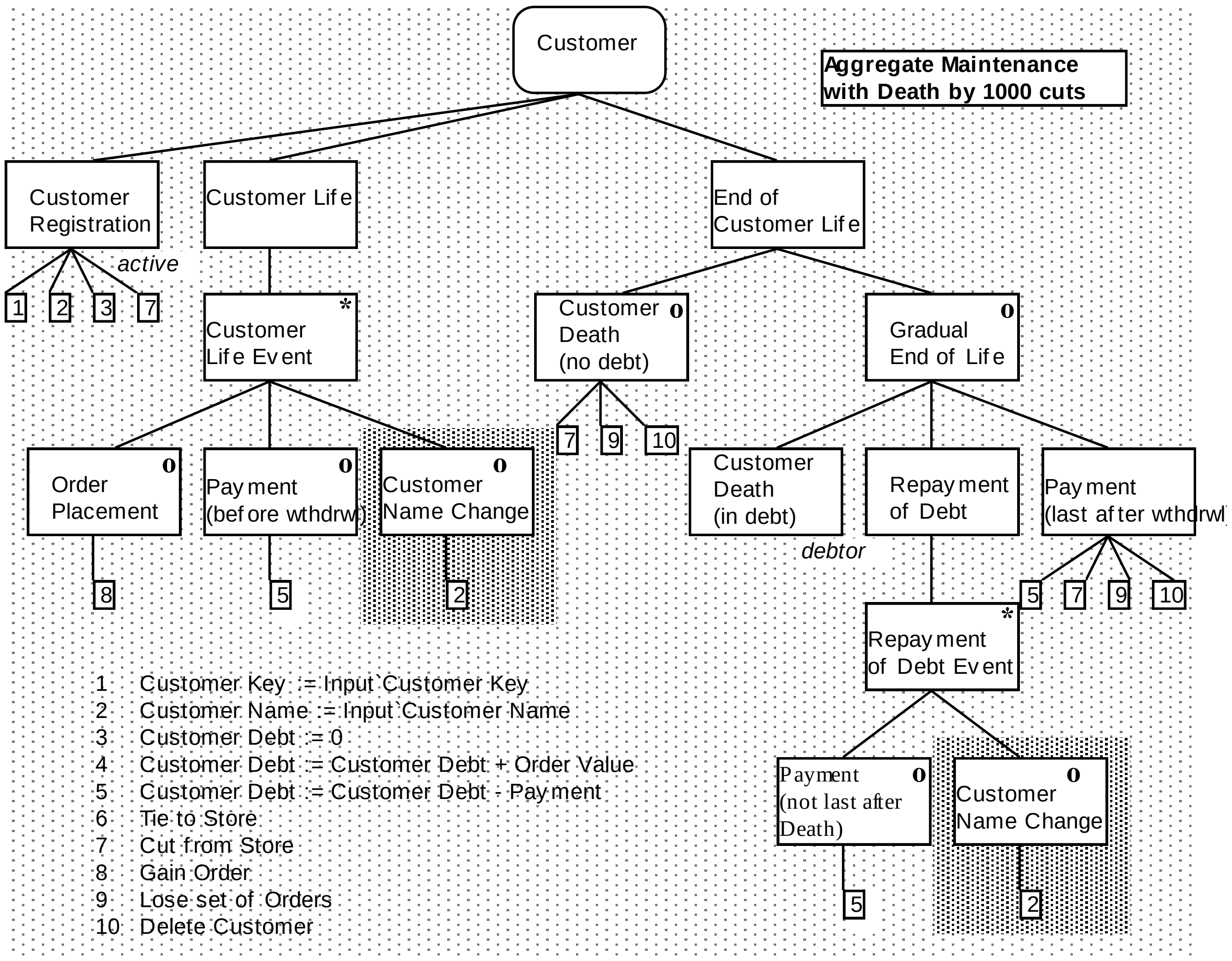
19.8 4 Add derivation actions

Further analysis may show that what seem to be only enquiry effects do in fact update attributes or relationships of the object. For example, see the actions allocated in the diagram below. Rather than worry about whether an event has an update effect or not, it is easier to follow the rule: include all the effects of one event in a entity state machine, both update and enquiry. An exception is discussed as a simplification six in the next chapter.

I am very close to now to a complete, graphical and object-oriented specification of all the processing of one class. However, some events have not been included. So far I have left out events that do not test or set an object's state variable, but do change an attribute or relationship of the object.

19.9 5 Add state-independent events that trigger derivation actions

A ‘state-independent event’ neither tests nor updates the state variable. It is valid at any time. You may however include it in a entity state machine to specify the update of an attribute or relationship. For example, you may include the Customer Name Change event in the entity state machine as shown as below, at the expense of some minor duplication.



1 Customer Key := Input`Customer Key
2 Customer Name := Input`Customer Name
3 Customer Debt := 0
4 Customer Debt := Customer Debt + Order Value
5 Customer Debt := Customer Debt - Payment
6 Tie to Store
7 Cut from Store
8 Gain Order
9 Lose set of Orders
10 Delete Customer

Actually it is untrue that the Customer Name Change event does not need to test the state of the Customer object. It needs to make the most basic test of all, that the Customer exists (its state variable is not null). However, it is harmless to omit such trivial events from entity life history analysis, as suggested in the next chapter.

By the way, the minor duplication of Customer Name Change event effects could be resolved by creating a parallel aspect class just for Customer Name Changes, but this would be more trouble than it is worth in such a simple case.

20. Behavioral constraints as preconditions

A life history diagram specifies constraints on the sequence of events in a entity state machine. This chapter distinguishes an enterprise-level event from a system-level event. It describes ways to simplify life history diagrams, and to specify constraints in them by allocating preconditions rather than spelling out event sequences.

20.1 Entity state machines

State: Every object experiences events and responds to them. Transient objects see only one event at a time, and have no memory of past events. Persistent objects see a stream of events over time, and retain a memory of what has happened. An object's memory is often called its state.

Entity state machines: The stream of events affecting a persistent object is describable as a entity state machine. A entity state machine is a fact of life; it happens, whether you document it or not. A entity state machine is the behavior of an object, or a class of objects that share the same behavior. It is possible for one object to have several parallel entity state machines, but this is not relevant here.

Events in a entity state machine: The object-oriented principle of encapsulation means that nobody but an object can see its own memory, its own state. So any event that inspects the state of an object must (by definition) appear in its entity state machine.

20.2 Life history diagrams

Life history analysis investigates and documents the entity state machines of the cooperating objects that form a system. There are various ways to document a entity state machine. Some are limited to recording events and states. A simple state transition diagram specifies the states an object can be in, the state transitions it can undergo, and the events that cause those state transitions.

A life history diagram is like a state transition diagram. They both specify constraints on the sequence of events in a entity state machine. But the former can more easily be annotated with details of constraints and derivation rules that events apply to attributes and relationships.

20.3 Three questions

An enterprise-level event happens in the real world. A system-level event is what the system gets to hear about. The first question is:

- Should we document enterprise-level events in a life history diagram?

A entity state machine is what happens. A life history diagram is only a description of what happens. The second question is:

- Should we document every event that occurs in an object’s entity state machine in that object's life history diagram?

It is in theory possible to specify every structural and behavioral constraint as a sequence of events in a life history diagram (other chapters support this assertion). So, the third question is:

- Should we try to specify every structural and behavioral constraint as a sequence of events in a life history diagram?

The answers are given below, along with five ways to simplify life history diagrams.

20.4 1 Omitting trivial events

Consider an event that does nothing but overwrite some descriptive text attributes of an object with new values, and is not constrained by any precondition (other than the object exists of course). This event definitely occurs in a system-level entity state machine of the object, but is it worth documenting in a life history diagram?

No. The purpose of life history analysis is to document non-trivial update events and constraints upon those events. There is little point in drawing life history diagrams to specify update events that are trivial and unconstrained. This is the kind of event an application generator can generate from an entity model.

So, you can omit from a life history diagram any event that is valid at any time during the life of the object (that is, between creation and deletion) and is so trivial that it does not affect any other object (so does not appear in any other entity state machine).

E.g. you could reasonably omit the Customer Address Change event from the Customer life history diagram.

Caveats

If you generate event specifications from life history diagrams, then you will not generate specifications for any of the trivial events. You might get around this by adding all the trivial events into the life history diagrams at the very end of analysis, after all state-changing and state-testing events have been fully analysed.

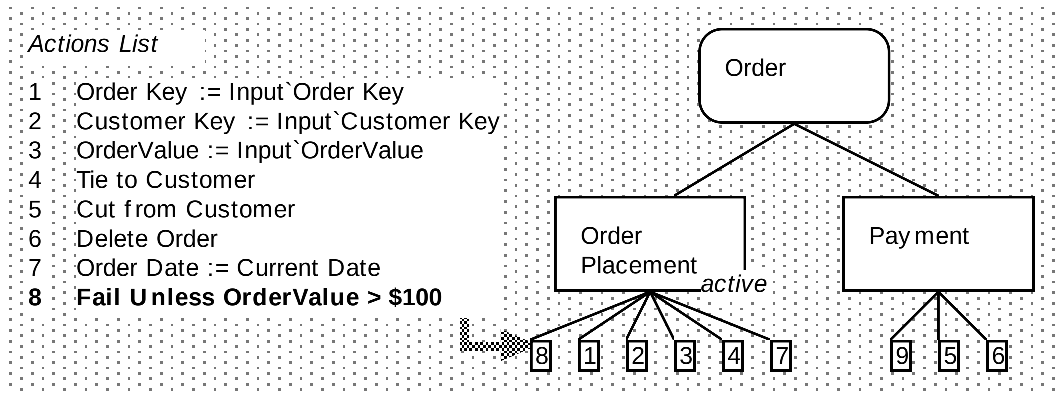
By the way, once you have used the facilities of an application generator to generate the code for trivial update events, you have the problem of turning off or constraining the processes it generates. This mix-and-match approach to design often turns out to be harder than coding all the update processes by hand.

20.5 2 Allocating range constraints

Consider the constraint that an event should not take an object’s attribute over a limit, or outside a permitted range. This is definitely a constraint on an event in a entity state machine, but is it worth trying to document the constraint as a sequence of events?

No. Model builders who try to turn the cardinality of an attribute or relationship into a state variable, tend to produce a highly overelaborate set of life history diagrams. You should instead allocate a precondition under the relevant event, allocating an action of the type: Fail unless precondition true.

E.g. an Order Placement event must fail unless the Order Value is greater than $100.



Actions List
1 Order Key := Input`Order Key
2 Customer Key := Input`Customer Key
3 OrderValue := Input`OrderValue
4 Tie to Customer
5 Cut from Customer
6 Delete Order
7 Order Date := Current Date
8 Fail Unless OrderValue > $100
Order
Order Placement
active
Payment
8
1
2
3
4
7
9
5
6

Where a constraint refers data from several objects, which entity state machine contains the constraint? The general principle is to place a constraint in the entity state machine that owns the attribute whose range is being tested. Consider three constraints on an Order Placement event

<u>Fail unless OrderValue (OrderQuantity * StockUnitPrice) > $100</u>

OrderQuantity is an attribute of Order. StockUnitPrice is an attribute of Stock Type. But the constraint is on OrderValue, this is an attribute of Order, so the precondition belongs in the Order entity state machine.

<u>Fail unless TotalUnpaidOrders < 5</u>

The constraint is on TotalUnpaidOrders, this is an attribute of Customer, so the precondition belongs in Customer entity state machine.

<u>Fail unless OrderValue + CustomerDebt < CustomerCreditLimit</u>

OrderValue is an attribute of Order. CustomerDebt and CustomerCreditLimit are attributes of Customer. The constraint is on the difference between the Limit and the Debt, this difference is an attribute of Customer, so the precondition belongs in the Customer entity state machine.

By the way, I do not say here whether the ‘derived attributes’ such as OrderValue or TotalUnpaidOrders are stored or derived when needed. It is in practice often necessary to store redundant data in order to save redundant enquiry processing.

Caveats

There is no caveat. The only reason to turn the cardinalities of attributes into the state variables of entity state machines is to show it can be done. This is an interesting exercise from an academic point of view, and it is explored in appendix 99.

20.6 3 Constraining the objects that receive an event

Consider a broadcast event that starts at a parent object and travels down a one-to-many relationship where it updates some, but not all, of the children. If every child receives the event, then the child entity state machine must include both update and non-update effects of the event. The non-update effect is an enquiry revealing the object is in the wrong state for the update effect.

E.g. Every Order gets to hear of the Customer Deletion event. This has an update effect on unpaid Orders but no effect on paid ones. So, the Customer Deletion event appears twice in the Order's life history diagram, having an update effect before Payment and an enquiry effect after it.

However, it is easier to postulate a *fetch relevant child* action that skips over children on which the event has no effect. This means you can omit the non-update effect from the child's life history diagram. You can show the selection condition by qualifying the name of the update effect with a role name.

E.g. Only unpaid Orders get to hear of the Customer Deletion event. So, the event appears only once in the Order's life history diagram, before or instead of Payment, and is qualified in brackets thus Customer Deletion [unpaid].

Caveat

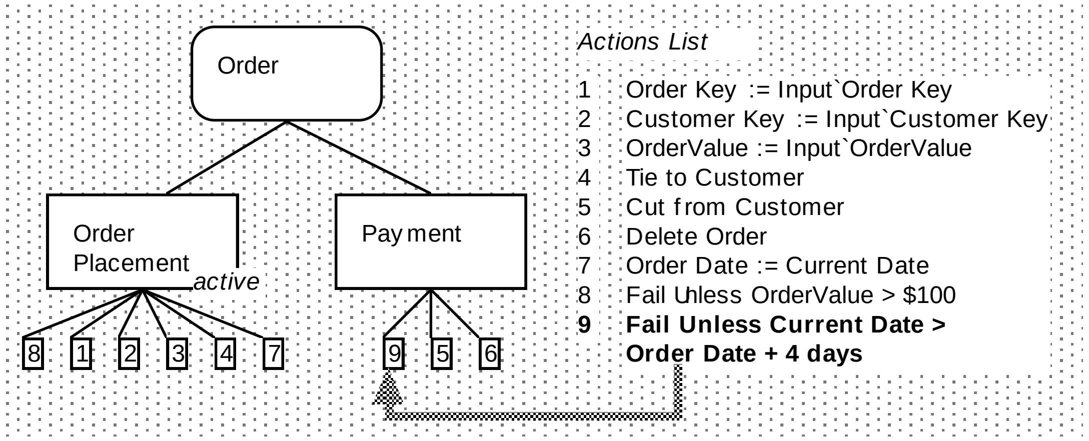
Later, it may turn out that non-update effects are update effects after all, so have to be reintroduced into the life history diagram.

20.7 4 Allocating date or time constraints

Consider the constraint that event B must be rejected until enough time has elapsed since event A. There is definitely a sequence of three events in the real-world. The enterprise-level entity state machine contains a sequence of three events: event A, Elapsed Time event, event B. But is it worth including the Elapsed Time event in a system-level entity state machine?

No. You can instead deduce whether the Elapsed Time event has occurred by a derivation rule when the next event happens. You allocate a precondition under the date-constrained event that says: Fail unless event B date < or > derived date.

E.g. Consider the constraint that a Payment event must be rejected unless four days have elapsed since the Order Placement. There is definitely a sequence of three events in the real-world: Order Placement, Fourth Day, Payment. But the system-level life history diagram can be reduced to two events.


Order
Order Placement
active
Payment
8
1
2
3
4
7
9
5
6
Actions List
1 Order Key := Input`Order Key
2 Customer Key := Input`Customer Key
3 OrderValue := Input`OrderValue
4 Tie to Customer
5 Cut from Customer
6 Delete Order
7 Order Date := Current Date
8 Fail Unless OrderValue > $100
9 Fail Unless Current Date > Order Date + 4 days

Caveats

If you generate an event specification for the Date event by reading event names from the life history diagrams, then the event specification will not show all the objects that are accessed.

A date event should be drawn explicitly in any life history diagram where it has an update effect.

20.8 5 Omitting enquiry-only effects of events

Remember that any event that inspects the state of an object must appear in its entity state machine. Consider an event that updates one object and needs to make an enquiry upon another, perhaps to check it is in a valid state or perhaps to retrieve some data needed for a derivation rule. The event clearly does occur in the entity state machine of the enquired-on object. But is it worth documenting this in its life history diagram?

Perhaps not. You can instead allocate an enquiry invocation action under the update effect in the life history diagram of the updated object. Examples are discussed below under the heading of referential integrity.

Caveats

If you generate event specifications by reading event names from the life history diagrams, then the event specifications will not show any of the objects that are only accessed for enquiry. Both the invocation of the enquiry, and the enquiry operation itself, must be included within the event specification.

Later it may turn out the enquiry operation is an update operation after all, so has to be reintroduced into the life history diagram.

Objections to including enquiry effects in practical system documentation should disappear with the advent of adequate CASE tool support for drawing the diagrams, for automatically optimising state variable values, and for generating event rules tables; and with better teaching of how to design the most economical set of life history diagrams.

20.9 Referential integrity in life histories

Referential integrity constraints are intended to guarantee the integrity of parent-child relationships. The question here is: How are these constraints documented in life history diagrams?

It seems intuitively obvious that any event that affects a relationship between a parent and a child must have an effects on, appear in the life history of, both parent and child. But if an event only makes only an enquiry upon an object, then there is the possibility of making the optimization mentioned above, of omitting that event from the life history diagram of the enquired-on object. Let us consider two cases, a restricted creation constraint and a restricted deletion constraint.

20.10 Allocating restricted creation constraints

Referential integrity means a child cannot be created or tied to a parent unless that parent exists. There is a constrained event sequence here. The parent creation event must occur before the child creation event. You may show this in the life history diagram of the parent. Normally the child creation event is iterated somewhere between the parent’s creation and deletion events.

However, you may instead allocate a precondition in the child’s life history diagram under its creation EVENT: *Fail unless parent exists.* You may then omit the child’s creation event from the parent’s life history diagram.

Where the target implementation environment is a database that can enforce referential integrity constraints and you intend to switch this function on, then you might choose to rely on the database to apply the constraint.

20.11 Caveats

If you generate an event specification for the creation event by reading event names from the life history diagrams, then the event specification will not show the parent object that is also accessed.

Later, it may turn out that the child’s creation event (e.g. Order Placement) must test the state of the parent rather than its presence or absence (e.g. the Customer must not be suspended). Or it might have an update effect on the parent such as adding to a total (e.g. the CustomerDebt).

These things turn out to be true so often that this is the weakest of the proposals for simplifying life history diagrams. It is generally easier and safer to teach the rule: A parent’s life history diagram should include the birth and death events of its children. You may easily remove events that are proven to have no update or enquiry effect at the very end of life history analysis.

20.12 Allocating restricted deletion constraints

Referential integrity means a parent cannot be deleted while any children are attached to it. This leads to four basic specification options for any parent deletion event.

Cascade deletion; the parent takes all its children with it; this only works if the user is happy to lose the children.

Set null deletion; all the children are cut from the parent: this only works if the relationship is optional at the child end.

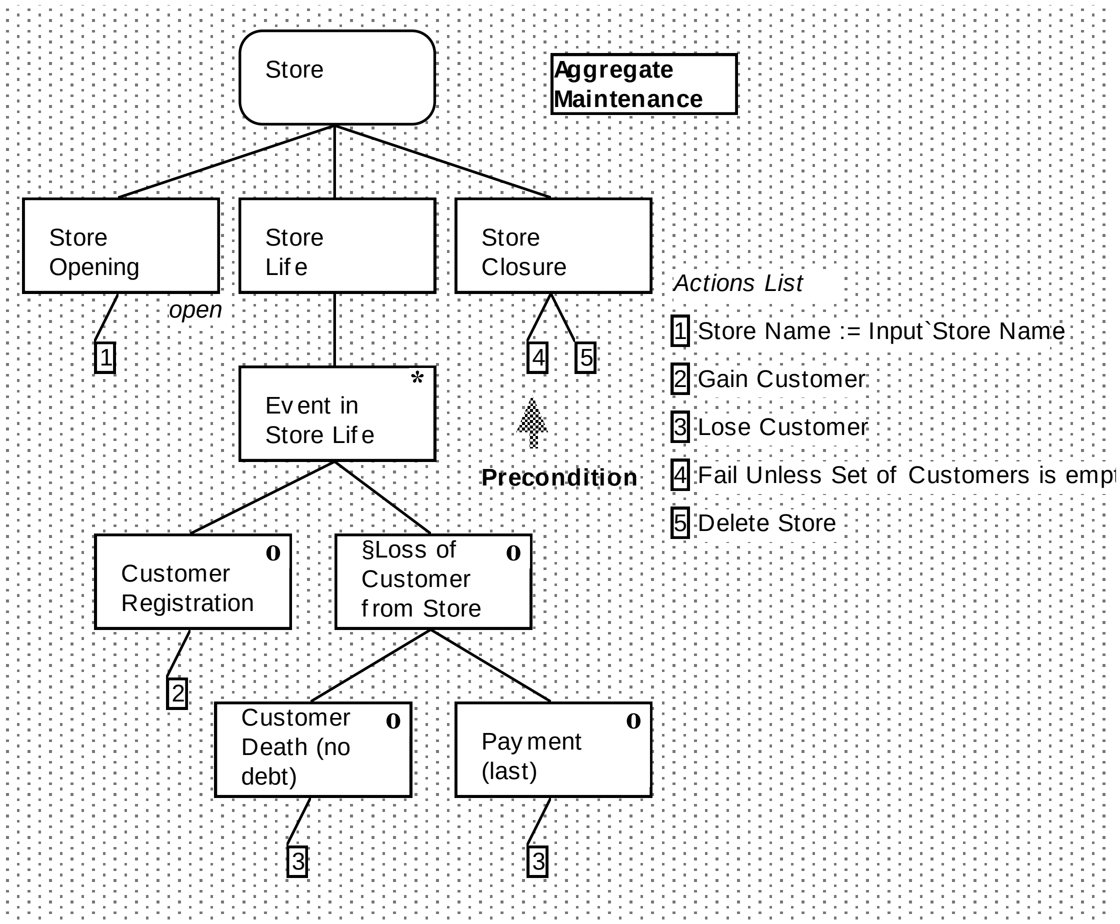
Swap parent deletion; all the children are swapped to another parent; this only works if another parent can be nominated on the parent deletion event.

Restricted deletion; a parent's deletion event is constrained not to happen until all its children have been deleted, perhaps one-by-one.

There is a constrained event sequence in a restricted deletion. The child deletion event must occur before the parent deletion event. You may show this in the life history diagram of the child; you add the parent deletion event at the very end.

This is a correct but counter-intuitive form of specification. You may instead allocate a precondition in the parent's life history under its deletion EVENT: *Fail unless no children exist.* You may then omit the parent's deletion event from the child's life history diagram.

E.g. a Store Closure event is constrained not to happen until all its Customers are deleted.



How is the 'no children' test made? Where the target implementation environment is a database that can enforce referential integrity constraints, then you might choose to rely on the database to apply the constraint in the database layer, and simply test the result in the business rules layer.

If it is impossible to make the 'no children test' without attempting to access the children, and you want to reduce databases accesses, you can optimize by specifying that the parent object maintains an ExistingChildren total, and tests this as a precondition before its own deletion.

20.12.1 Caveats

If you generate an event specification for the deletion event by reading event names from the life history diagrams, then the event specification will not show any child objects that are accessed to test their existence.

Restricted deletion doesn't work if the restriction applies to the logical death event of the child (where the child is not deleted but merely moved to a new state 'dead'). In this case one can refine the optimization above by asking the parent to maintain an ActiveChildren total rather than an ExistingChildren total

It may turn out that the parent’s deletion event deletes its children after all, so it is really a cascade deletion, and it does belong in the child’s life history diagram.

20.13 Conclusions

The three questions were:

- Should we document enterprise-level events in a life history diagram?
- Should we document every event that occurs in an object’s entity state machine in that object's life history diagram?
- Should we try to specify every structural and behavioral constraint as a sequence of events in a life history diagram?

This chapter has answered ‘No’ to all three questions. It has described five ways to simplify life history diagrams, and to specify constraints in them by allocating preconditions rather than spelling out event sequences. I recommend the first four simplifications; they are reasonably robust. The last is a risky optimisation. Don’t do it unless you have considered the consequences.

~~21.~~ Three-way conceptual modeling

A small case study shows how the three conceptual modeling techniques complement each other, and something of how to develop an event-oriented specification alongside an entity-oriented one.

The chapter discusses the birth and death events of child objects, and the analysis of a structural relationship to reveal one of the following patterns in the life history of the parent object: Aggregate Maintenance, Swap Parent, or Flip-Flop

21.1 Child birth and death events

Consider the birth and death events of the child entity in this tiny two-entity system.

Entity state model	
ENTITY: School	**Invariants**
School Name	
Pupils	= total number Pupil entities
ENTITY: Pupil	
School Name	= identify of a known School
Pupil Serial Num	

You can draw the Pupil Registration and Pupil Deletion event rules tables following the patterns for child birth and child death events given earlier. An event rules table shows all the object instances affected by one event. An arrow shows how an object is identified, by the event parameters or by another object. You can add the details of preconditions and post conditions.

EVENT: Pupil Registration (PupilSerialNum, SchoolName)		
Entities affected	**Preconditions**	**Post conditions**
School	Present	Pupils = Pupils +1
Pupil	Present	Tied to School

EVENT: Pupil Death (PupilSerialNum)				
Entities affected			**Preconditions**	**Post conditions**
Pupil			Present	Deleted
	o-- at school	School		Pupils = Pupils - 1

A numbered operation is a single processing step. Operations include constraints, guards or preconditions. The Pupil Deletion event will fail unless the Pupil and the School exist.

The aim here is to paint an impressionistic picture rather than explain all the details. The point is: you can specify all the implementation details, operations and rules fired by one event on a diagram like this. Further, a tool can generate most of these details from the entity state machine diagrams by following a mechanical transformation procedure.

Object interaction analysis takes an event-oriented view. Entity life history analysis takes an object-oriented view of the same specification.

21.2 From structural relationship to Aggregate Maintenance life history pattern

An object-oriented entity state machine shows all the events affecting one class. It gives a dynamic view of the class in terms of the events that update its attributes and relationships, and the sequential constraints on these events.

- Q) Given a child class: do the birth and death events of the child refer to the state of the parent?

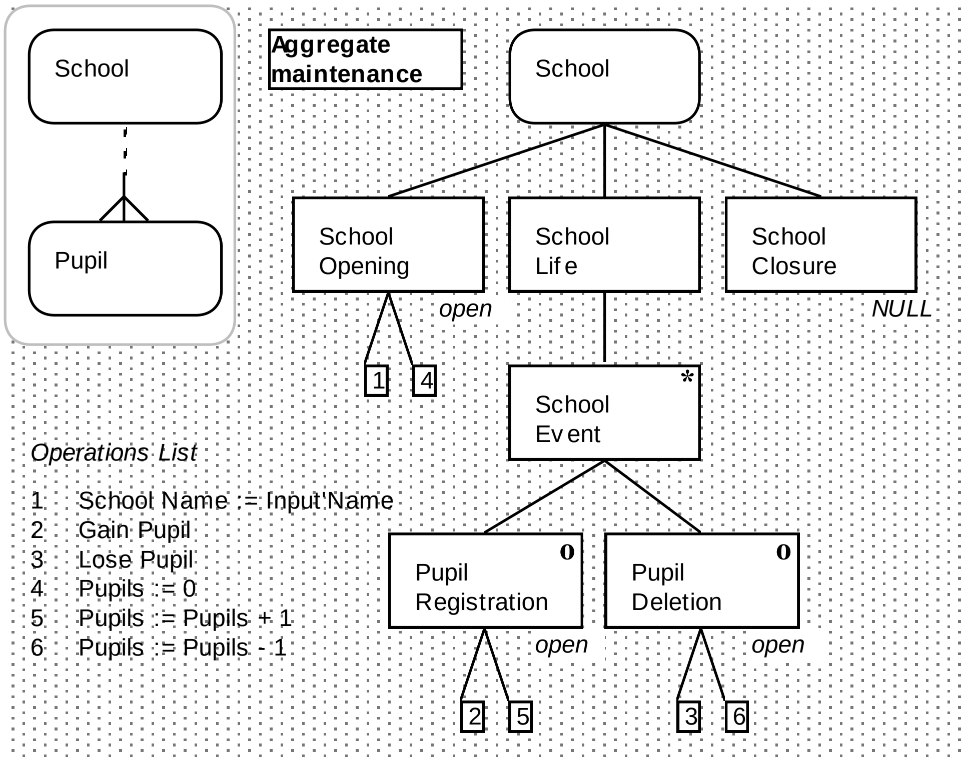
If yes, copy the child birth and death events into the entity state machine of the parent. E.g the School entity state machine includes Pupil Registration and Pupil Deletion.

- Q) Given a one-to-many relationship: Is the relationship monogamous - only one child alive at one time, the rest being historical?

If yes, the entity state machine of the parent should include the child's birth and death events under the Flip-Flop pattern. An example appears later.

- Q) Given a one-to-many relationship: Is the relationship polygamous - many children alive at one time?

If yes, the entity state machine of the parent should include the child's birth and death events under the Aggregate Maintenance pattern. For example:



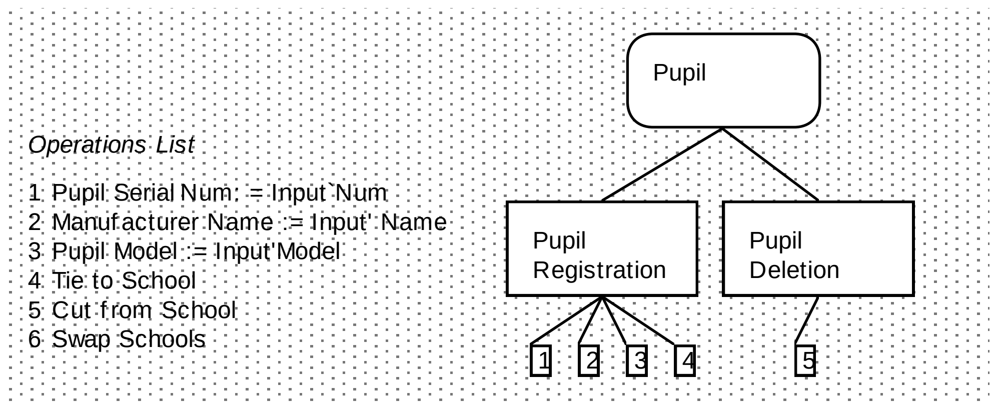
The named boxes at the bottom of a entity state machine are the ‘event effects’, or simply ‘effects’. Each event fires a ‘method’ in an object, having the effect shown.

The label in the bottom-right-hand corner shows the value of the state variable after the event effect. After ‘optimising’ the values using the standard rules in chapter 2d, the entity state machine is a One-State Life, so the state variable is not required.

The numbered boxes beneath the effects are operations. In OO programming, you would code an operation as a single processing step within the implementation or ‘method body’.

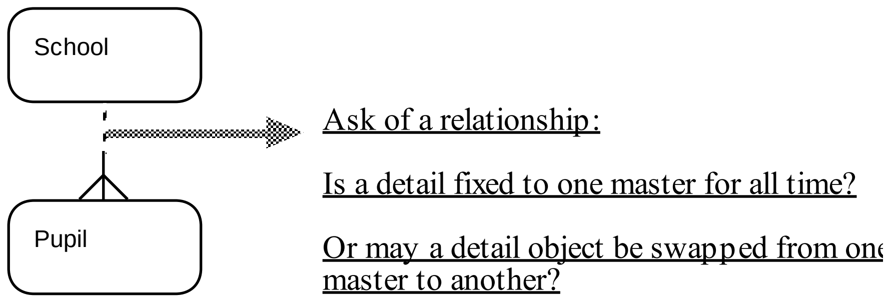
The numbered list shows the executable operation types. You can draw operation types from a generalised list of the operation types that are implementable across a known range of target implementation environments.

The aggregate being maintained is the set of active children, Pupils. The entity state machine of the child class is very simple.



21.3 From structural relationship to Swap parent life history pattern

Looking at a class relationship model, the full nature of the relationships is obscure.


School
Pupil
Ask of a relationship:
Is a detail fixed to one master for all time?
Or may a detail object be swapped from one master to another?

If the relationship is mandatory, and the child object has a changeable parent, this implies a Swap Parent event. E.g. you may discover there is a Pupil Transfer event.

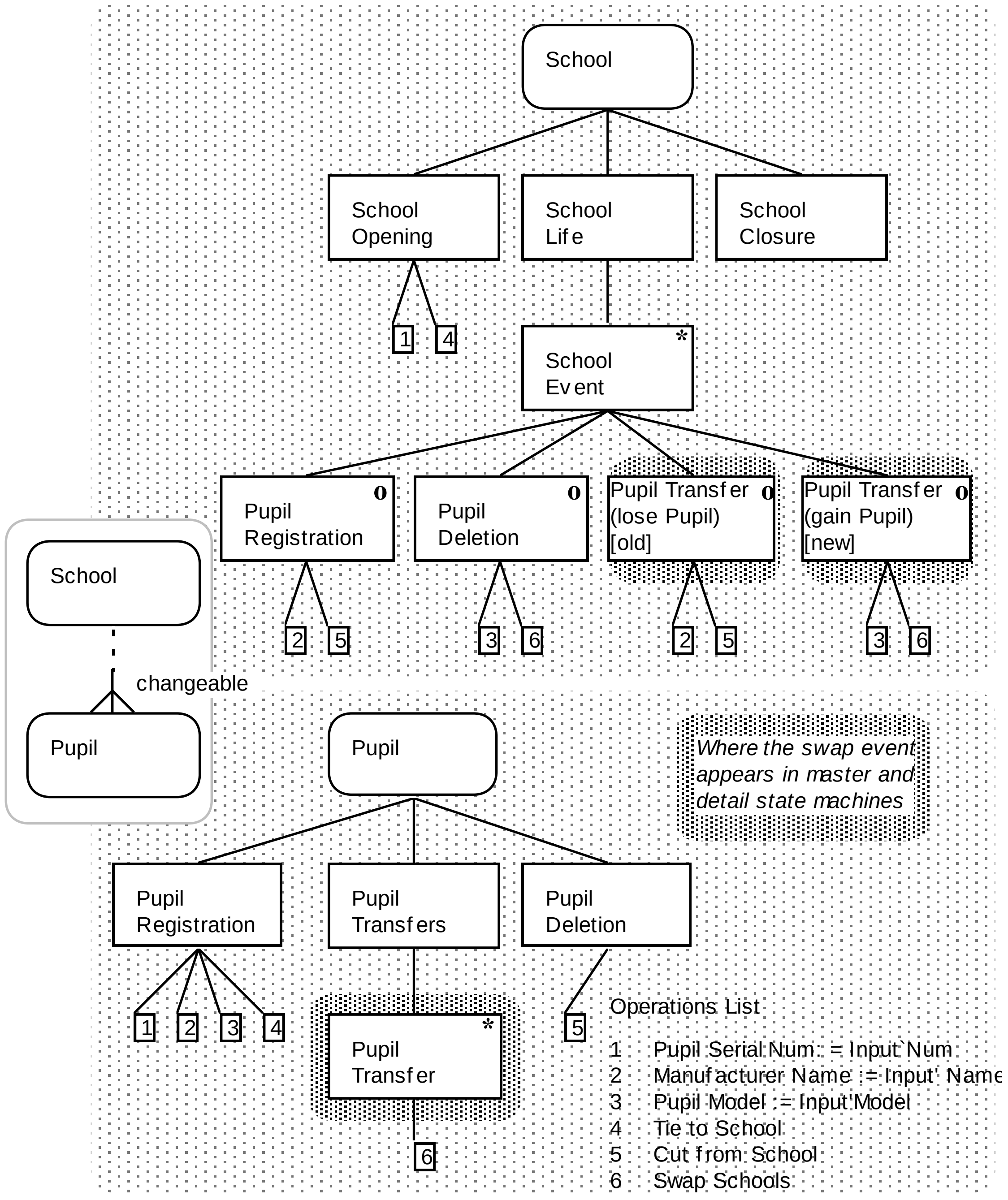
EVENT: Pupil Transfer (PupilSerialNum, SchoolName [new])				
Entities affected			Preconditions: Fail unless…	Post conditions
Pupil				Pupil swapped from old school to new school
	--->	School [old] (lose pupil)		Pupils = Pupils - 1
School [new] (gain pupil)			School not full	Pupils = Pupils + 1

The event affects two different objects of a class (School) in different ways. Objects of the class play different roles with respect to the event. Roles are shown by adding a role name in square brackets; this tells you how the event finds or recognises this particular object instance, as opposed to any other object of the same class. The different effects are further described with an effect name in round brackets.

Class name	**Entity role name**	**event effect name**
School	[old]	(gain Pupil)

School	[new]	(lose Pupil)

Each event effect appears in both event-oriented and object-oriented views. So how does this Pupil Transfer event appears in the entity state machine view of the classes in the event rules table? Normally, a swap parent event makes a predictable appearance in the entity state machines. You have to copy the swap parent event twice into the entity state machine of the parent, the two effects being on different objects of the same class.



21.4 Entity roles and event effects

Notice that in the entity state machine for a School, the event effects are named in the opposite way from in the event rules table.

Event name	Even effect name	Entity role name
Pupil Transfer	(gain Pupil)	[old]
Pupil Transfer	(lose Pupil)	[new]

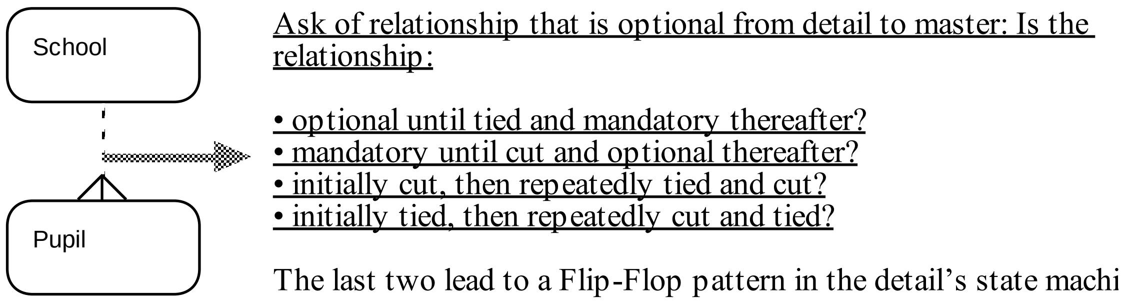
21.4.1 From structural relationship to Flip-Flop pattern

There are two kinds of relationship that prompt you to draw a Flip-Flop pattern in a entity state machine: an optional relationship with an independent child, and a monogamous relationship.

It turns out that one relationship is a refinement of the other. When you change the data structure, the entity state machines of existing classes retain exactly the same shape, but with different operations.

21.4.2 From optional relationship to Flip-Flop pattern

Suppose the relationship from Pupil to School is optional at the child end. Looking at the class relationship model, the reason is obscure. You should ask questions about it.



Ask of relationship that is optional from detail to master: Is the relationship:

- optional until tied and mandatory thereafter?
- mandatory until cut and optional thereafter?
- initially cut, then repeatedly tied and cut?
- initially tied, then repeatedly cut and tied?

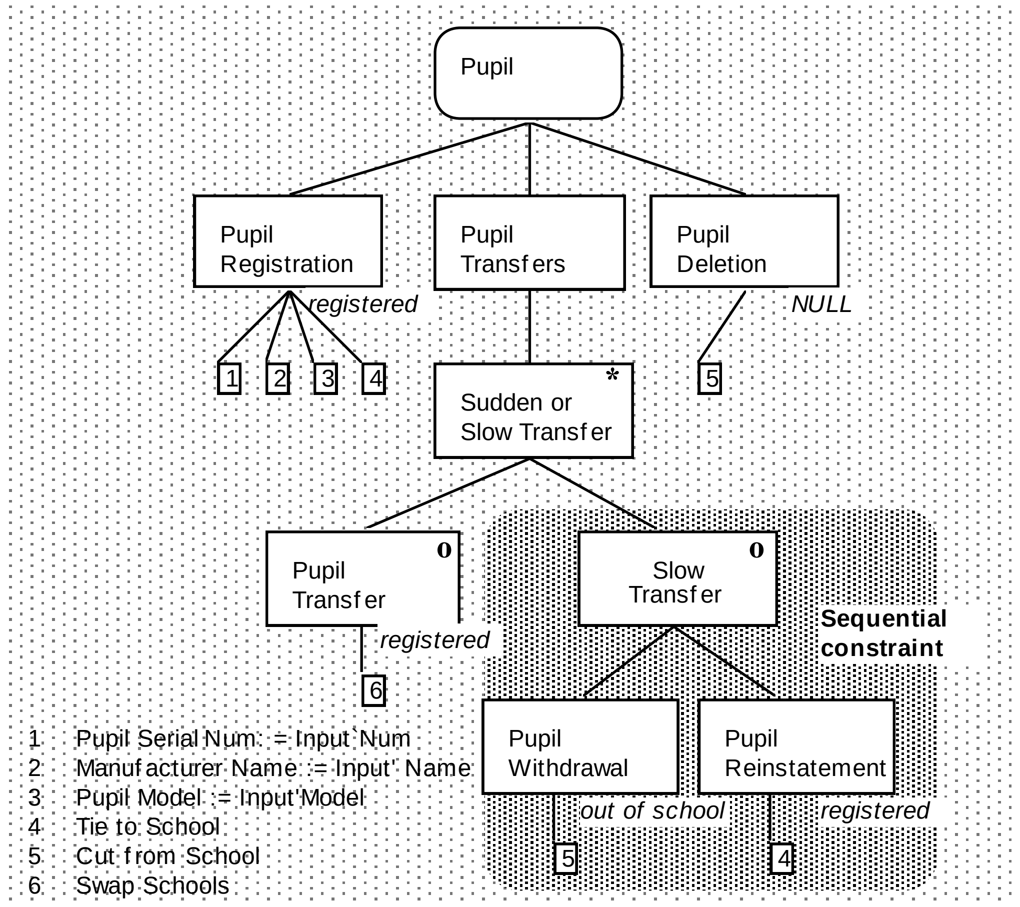
The last two lead to a Flip-Flop pattern in the detail's state machi

A Pupil can be transferred from one School to another in two ways: either suddenly by one transfer event; or slowly, by withdrawing a Pupil from one School and reinstating the Pupil later in another School. So, you discover that the relationship is initially tied, then may be repeatedly cut and tied.

You cannot model the sequential constraint between Pupil Withdrawal and Reinstatement in the entity state machine of the parent (School), since the behavior of one Pupil is interleaved with the behavior of all the other concurrent Pupils.

However, you can model the sequential constraint in the life of the child using the Flip-Flop pattern. In this case the Flip-Flop pattern is complicated by the addition another optional event type (Pupil Transfer) that may occur between cycles.

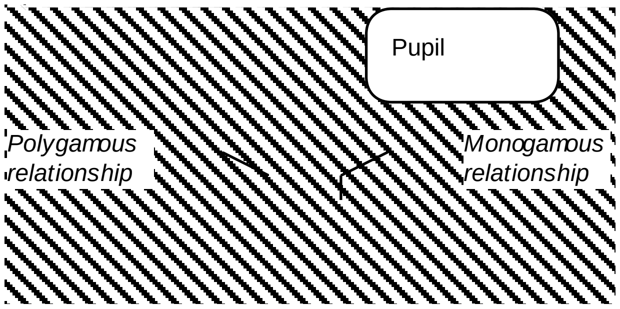
The Pupil entity state machine below shows the constraint that a Pupil Reinstatement event can occur only after a Pupil Withdrawal event, and two Pupil Withdrawal events cannot happen in a row. You now need to allocate state variable values, say, ‘registered’ and ‘out of school’.



From optional relationship to monogamous relationship

- Q) Given a Flip-Flop pattern: Does it reveal a child object that users want to keep a history of?

If yes, then you should add the class into the class relationship model. E.g.



21.4.3 From monogamous relationship to Flip-Flop pattern

The relationship under School is polygamous, there are many concurrent children. The entity state machine of the parent (School) contains the birth and death events of its child (Pupil) under the Aggregate Maintenance pattern. You can only show the birth-death sequence in the life of a child class.

The relationship under Pupil is monogamous, there is only one active child at a time. So the birth-death sequence appears not only in the life of the child class, but also in the life of the parent class. The entity state machine of the parent (Pupil) contains the child's birth and death events (in reverse order in this case) under a Flip-Flop pattern.

21.5 Conclusions

The purpose of this chapter is to paint a broad picture showing how you can bring the three dimensions of a conceptual model into harmony. I've only scratched the surface here of the many patterns you can recognise. Let me review some of the ideas in this chapter, especially with regard to graphical representation of a system specification.

21.5.1 A class relationship model shows the cardinality of associations

A class relationship model gives a static picture of classes in terms of the associations that relate two classes to each other. A class relationship model is very useful in systems development, but it does not tell the whole story. The constraints it shows are mainly to do with the cardinality of objects at either end of a relationship. You cannot show all of the other constraints that end-users want to place on data processing.

Most notably, you cannot show the time dimension. A class relationship model is a static view of a system, a snapshot at one moment, it does not show how an object instance changes over time, or how a relationship changes over time.

21.5.2 An event rules table shows the corresponding effects of one event

You can best visualise and recognise the corresponding effects of an event by representing them graphically in an event-oriented diagram. An event rules table shows all the object instances affected by one event. You can add implementation details.

21.5.3 Event-oriented and object-oriented views must correspond

You can view the effect of an event on a class from two perspectives, that of the transient event and that of the persistent object. So each event effect in an event rules table can be transformed into an event effect in an entity state machine, and vice-versa. This is enormously helpful in systems analysis, in teaching, and in developing CASE tool support.

21.5.4 A entity state machine shows the events affecting one entity

A entity state machine shows all the events affecting one class. It gives a dynamic view of the class in terms of the events that update its attributes and relationships, and the sequential constraints on these events. You can add implementation details.

21.5.5 A conceptual model can be implemented

Conceptual specification of requirements should precede physical design. In other words, an analyst should sort out the user's business needs before worrying about a specific user interface technology, database technology or programming language.

But it is very, very important that anyone who proposes a conceptual form of specification should demonstrate it can be carried forward through physical design to implementation. It is important to know that there are techniques, supportable by CASE tools, that help us transform between entity state machines and event rules tables, and transform either or both into program code. In the end, you may code from either perspective.

21.5.6 Various coding styles

The more effort you put into analysing object behavior and interactions, the easier it is to implement the system. But object-oriented analysis does not have to lead to object-oriented programming. How you cut the code is a different level of concern. It doesn't matter so much whether you write the program code in the form of object-oriented class specifications or event-oriented procedural routines; you can do either.

22. Can there really be life after death?

A short story about modeling constraints in structural and behavioral models.

This chapter is an entertainment with some serious purposes. The story may give you some insights into the analysis and specification of constraints on entity models and behavioral models (entity state machine diagrams and event rules tables). It alludes to the scheme for business rules introduced by the Business Rules Group.

The story also forms a backdrop to questions that I believe should be addressed by the OMG in looking to improve business rules specification in UML - whether to assist the Model Driven Architecture initiative or just because it is a good idea.

22.1 A few terms and concepts

An **event** is a discrete, atomic, all-or-nothing happening. It updates one or more objects, and perhaps refers to the state of other objects. An object¹s **entity state machine diagram** shows a long-running process, the pattern of events that update or refer to the state of an object over its life. Database readers: think of objects as relations or tables. J2EE readers: think of objects as entity beans. An event¹s **event rules table** shows a short-running process, the pattern of objects affected by one event. Database readers: think of events as database transactions. J2EE readers: think of events as session beans.

For any given software system, its set of entity state machine diagrams and its set of event rules tables are isomorphic views - one can be transformed into the other - though it is very rare to find either view completely documented, outside of classroom case studies.

Entity and event modeling is an analysis and design method based on iterative refinement around and between three complementary modeling techniques:

- A - draw object relationship model
- B - draw objects' life histories as entity state machine diagrams
- C - draw events' event rules tables

The aim is to generate C from B as mechanically as possible, then generate code from C.

22.2 The story

David Hay (best known as author of "Data Model Patterns" ISBN -932633-29-3) suggested to me recently that UML cannot model a "restricted deletion" constraint. This could lead us into a debate about whether the "aggregation" concept in UML models this constraint or not, and I agree with Martin Fowler that aggregation is a poorly-defined concept. But I¹d rather let it lead me into telling a story about the life after death paradox.

22.3 Life is taxes, death and more taxes

The taxman may refer to your estate after your death. Below, in the entity state machine diagram that shows your life:

- sequence is shown top to bottom
- iteration is shown with an asterisk
- event names are in capitals
- post conditions (changes to attribute values) are in the right-hand column.

ENTITY STATE MACHINE: Life		Post conditions
BIRTH		StateVariable = alive
LIFE	* TAX	reduce CurrentAccountBalance
DEATH		StateVariable = dead
LIFE	* TAX	reduce EstateAccountBalance

You can easily redraw the events and states of this diagram as UML-style entity state machine diagram, but please allow me to use the more emailable diagram format above, a kind of 'Jackson structure'.

You may be a little worried by taxation after death. For many years, I was worried, in a more academic way, about events that refer to the state of an object after it has died, and the paradox this leads to when modeling the case where a "death event" has the effect of deleting the object's state record.

22.4 structural terms, facts, constraints and derivations

In the case study to be considered, structural terms include Parent, Child and NumberOfLiveChildren.

- A structural fact is that Children are born of Parents.

In the very strange world of the case study, an invariant Constraint is that a Child must have one and only one Parent. The entity model shows the Invariant Constraint as a one-to-many relationship.

- Entity model: Parent ---<* Child

Obviously, you could draw this structure using UML. One of the irritating features of UML is the way it uses an asterisk to show both an interleaved many, where the many live in parallel, and an iterated many, where each cycle is constrained not to start until the previous one has finshed. Older notations used a crowsfoot for an interleaved many, and an asterisk for an iterated many, but I doubt the OMG will ever improve UML to distinguish these concepts.

- A structural derivation is that Parent.NumberOfLiveChildren = total of Child objects with StateVariable = alive.

22.5 Behavioral terms and facts

Behavioral terms include the names of events. A birth event creates an object; the object¹s state is now recorded and available for inspection or update. A death event usually signals the end of updates to an object's record. Other events update or refer to objects' states.

Behavioral facts include that Child Birth events affect both Children and Parents. The event rules table below is annotated with the effects – attribute value updates.

EVENT: Child Birth		
Entities affected	**Preconditions**	**Post conditions**
Parent object		add 1 to NumberOfLiveChildren
Child object		StateVariable = alive

22.6 behavioral constraints

Many behavioral constraints can be modeled by drawing the valid sequences of events in entity state machine diagrams. For example, birth precedes death.

ENTITY STATE MACHINE: Child	**Post conditions**
CHILD BIRTH	StateVariable = alive
CHILD DEATH	StateVariable = dead

ENTITY STATE MACHINE: Parent			**Post conditions**
PARENT BIRTH			StateVariable = alive
LIFE	* LIFE EVENT	o-- CHILD BIRTH	add 1 to NumberOfLiveChildren
		o-- CHILD DEATH	take 1 from NumberOfLiveChildren
PARENT DEATH			StateVariable = dead

The 'o' marks an option of a selection. The events under the iterated selection (or random mixture) in the Parent entity state machine diagram do not advance its state variable, but do update the attribute NumberOfLiveChildren.

After transforming the entity state machine diagrams into event rules tables, constraints on (preconditions of) the event will appear as conditions that test the state variables of the objects.

22.7 a restricted birth constraint

The Invariant Constraint that a Child must have one and only one Parent is not enough, that Parent must be alive when the Child is born. The event rules table for the Child Birth event shows this behavioral constraint as a fail condition in square brackets.

EVENT: Child Birth		
Entities affected	**Preconditions**	**Post conditions**
Parent object	StateVariable = alive	add 1 to NumberOfLiveChildren
Child object		StateVariable = alive

If an event discovers an object in the wrong state, then the whole event (session or transaction) is aborted, not just the operation on the object. So, if the Child Birth event discovers its Parent object in the dead state, then the Child Birth event (session or transaction) is aborted.

22.8 a restricted death constraint

In the strange and cruel world of the case study, there is a terrifying "restricted death" constraint. A Parent object cannot die until all its Children have died. The Parent's death event refers to the state variable of the Child, so must appears in its entity state machine diagram, after the Child's own death event.

ENTITY STATE MACHINE: Child	**Post conditions**
CHILD BIRTH	StateVariable = alive
CHILD DEATH	StateVariable = dead
PARENT DEATH	

The event rules table for the parent's death event looks like this

EVENT: Parent Death			
Entities affected		**Preconditions**	**Post conditions**
Parent object		StateVariable = alive	StateVariable = dead
	-->* Child object	StateVariable = dead	

If the Parent Death event discovers a Child object in the alive state, then the whole event is aborted, not just the operation on the Child.

22.9 a restricted deletion constraint

Death and deletion do not necessarily happen at the same time. Death is a constraint on further changes. Deletion is the removal of the object¹s record from the system. Logically speaking, objects live forever. Once created, an object's record persists for all eternity. So, deletion is an act of vandalism to the data record.

But in practice, we do mangle logical and physical models into one. Suppose a side effect of the Child's death event is to delete the Child's state or database record. The constraint becomes: a Parent cannot die until all its Children have been deleted.

Strangely, the shape of the entity state machine diagrams are unchanged, though they now show an event occuring in the entity state machine diagram of the Child object after it has disappeared. Only the annotation against the Child's death event is modified.

ENTITY STATE MACHINE: Child	Post conditions
CHILD BIRTH	StateVariable = alive
CHILD DEATH	delete state
PARENT DEATH	

The shape of event rules table below for the parent's death event is also unchanged. Only the fail condition on the Child object is modified.

EVENT: Parent Death			
Entities affected		**Preconditions**	**Post conditions**
Parent object		StateVariable = alive	StateVariable = dead
	-->* Child object	object missing	

Wierd. Truly wierd. But this process will works when coded.

22.10 life after death

For a long while I lived uncomfortably with the paradox that entity state machine diagrams can include posthumous events after the object has been deleted. The general model for this being:

ENTITY STATE MACHINE: Object		**Post conditions**
OBJECT BIRTH		StateVariable = alive
OBJECT LIFE	* LIFE EVENT	Update state
OBJECT DEATH		Delete state
OBJECT LIFE AFTER DEATH	* POSTHUMOUS EVENT	

22.11 multiplicity constraints

Having drawn an event rules table, you can annotate any number of constraints on it. That was the common practice in the 1980s.

e.g. one might annotate [Fail Withdrawal event unless Dollar Amount < 100]

For many years however, Keith Robinson and others were interested in trying to generating constraint rules from the shapes of entity state machine diagrams. Keith's CASE tool automated most of the work of transforming entity state machine diagram models into event rules tables. He generated not only the event rules tables from the entity state machine diagrams, but also the constraint rules from the entity state machine diagram shapes.

In UML terms, you might say we didn¹t document “guard conditions” on entity state machine diagrams, and our aim was to generate them on the event rules tables.

22.12 derivation of multiplicity constraints from child states

Every time you see an attribute that holds a sum or total value, you can analyse further to model the discrete items in that sum. Choosing to model at this lower level of detail or not is one of the many decisions you have to make during analysis and design.

e.g. A stock quantity total implies the presence of stock items. If the stock is of nuclear missiles, you'll want track each one. If the stock is of nails, then you won't, unless you are very very mean.

A entity state machine diagram constrains the sequence in which events can occur. It used be an article of faith for us entity state machine diagramers that *every* constraint can be shown in a entity state machine diagram as a sequence of events. We didn't want to document constraints any other way.

e.g. A constraint that there can be no more than 500 Items in a Stock Pile can be viewed as a constraint that only 500 Stock Items (be they Nuclear Missiles or Nails) can be in a position in their entity state machine diagram after their Stock Receipt event and before their Stock Issue event.

I once wrote a chapter (so academic I was too embarassed to show it to anybody) to demonstrate that a constraint of the kind "Fail Withdrawal event unless Dollar Amount < 100" was really a constraint on events in the entity state machine of a single dollar. At least, I think that's what it demonstrated. And it took me many pages to make a point I now skip over in a paragraph.

22.13 the pandora's box of business rules

About 1992, the Union Bank of Switzerland (UBS) ran a competition for application generator vendors to implement a case study that featured a small entity model with some complex business rules, and some recursive processing. E.g. if there is insufficient stock to fill an order, then it is filled partially, and it spawns a new order for the outsanding amount.

The competing vendors built a naive (CRUD) database solution in a day or two that completely failed to implement the required business rules.

Keith Robinson, also in a couple of days, built an object event model that contained all the required business rules. Sadly, Keith died shortly afterwards.

Some time later, Christoph Henrici of UBS called me over to have a look at Keith's Entity and event model. It was very complex, perhaps the most complex object event model I have ever seen. But it wasn't so much the complexity Christoph was concerned with. Christoph had noticed that Keith had resorted to annotating a constraint on a entity state machine diagram of the kind:

[Fail unless attribute = value]

Shock. Horror. Keith had documented a constraint on a entity state machine diagram, instead of either drawing out a sequence of events (the academic way), or simply adding the constraints as an afterthought into the event rules table (the practical way).

Keith¹s little "cheat" opened the pandora's box of business rules to me. I started to experiment by annotating various kinds of business rules on the entity state machine diagrams.

22.14 a multiplicity constraint

Using Keith's cheat, you can rework the case study above by omitting the posthumous events from the Child entity state machine diagram, and attaching a constraint to the Parent¹s death event thus:

[Fail unless NumberOfLiveChildren = 0]

But that total attribute is derived data. Some people are foolishly determined to remove all redundant data from their models. Suppose the Parent does not maintain an attribute that records the NumberOfLiveChildren?

22.15 a set-level constraint

We can be systematic about representing the "restricted deletion" rule in the Parent's entity state machine diagram, without having to show posthumous events in the Child¹s entity state machine diagram, by annotating the Parent¹s death event thus.

[Fail unless Child set is empty]

This implies that a Parent knows the set of its Children. This is intuitively obvious, but contrary to the relational paradigm, and that's an argument for another time.

The constraint tests a property of a set. Whenever all the children of a parent are the same in some way, then that property applies to the set or the parent who owns the set. This is a transitory state of affairs, since any of the children may change, get out of step with the rest. Nevertheless any constraint that tests that a condition applies to all members can be phrased as a test of the whole set.

22.16 Conclusions and remarks

The story above helps to illustrate several points. It is clear that to generate code from model diagrams, you have to annotate the diagrams with business rules in one way or another.

The Object-Oriented Design paradigm is good at interfaces (signatures anyway) not so good at business rules (semantics). The strength of Entity and event modeling is the weakness of Object-Oriented Design, and vice versa.

The OMG's Model Driven Architecture (MDA) now needs an adequate way to specify business rules. But adequate doesn¹t mean simply that it must work - it has to be fit for purpose. I believe there are some questions that must be addressed, and discuss those elsewhere.

For now, notice how malleable the concepts of business rule specification are:

- every Invariant Constraint can be recast as a behavioral constraint of one or more events
- a repeatedly-defined behavioral constraint may perhaps be more economically expressed one Invariant Constraint in an entity model
- sometimes, a constraint on the value of an attribute may be recast as a constraint on one or more state variable values, and vice versa.

And notice that every constraint mentioned in the story is a precondition of an event, not just one operation. If an event discovers an object in the wrong state, then the whole event is aborted, not just the operation on that object. This was one of the ‘awkward’ questions at the end of volume 1.

P.S. A reporting or enquiry-only transaction cannot enforce any rule. However, it can detect and select objects for the later application of an update transaction which does apply a rule.

23. Appendix A: Object-oriented analysis in the UK

A tribute to the late Keith Robinson.

It is almost certainly true that the longest continuous object-oriented research and development programme in the world was started by Keith Robinson in 1977 at Infotech. After Keith's death in 1993, the development was carried forward by John Hall of Model Systems and I (Graham Berrisford) who now work for Seer Technologies.

1977: Keith published a chapter in the Computer Journal proposing an object-oriented program design method for database systems (not called that of course). Keith started from Michael Jackson's earlier suggestion that the variables and processes of each object type could and should be encapsulated in a discrete processing module. An additional idea was to use the state variable of an object in validation of updates to that object.

1979: I helped Keith develop his proposals into a 10-day course called 'Advanced System Design' based on three techniques:

- Relational data analysis: Keith taught this as a technique to decompose the required system inputs and outputs (in what we might now call the UI layer) into entity types for behavior analysis in what we might now call the business services or data services layer.
- Life history analysis: Keith taught this as a technique to discover the behavior of each entity type and document it in a entity state machine diagram. He favoured using regular expressions as the notation and called them life history diagrams after Jackson I think.
- Object Interaction structures: Keith invented and taught these to document how objects exchange messages in order to complete the processing of an event (one event may synchronously update several objects, and/or need to be validated against the states of several objects).

Keith's three-dimensional approach to conceptual modeling is now the norm in modern development methods. But there was a lot more to his method than notations, and some of the ideas he taught to do with scheman evolution are still ahead of the game.

By the way, many years before Yourdon abandoned data flow diagrams, Keith advised against top-down decomposition.

1980: Keith's course disappeared when his employers went into liquidation. Not along after this, Keith helped John Hall to develop an analysis and design method for the UK government. SSADM version one was built on around database modeling techniques and incorporated object-based process analysis and design techniques.

Keith and John deemed object Interaction structures impractical for use by database programmers, but included life histories as an analysis tool for discovering processes and business rules. They assumed it was obvious that each life history or entity state machine could be transformed into a discrete program module using Jackson's technique of program inversion (more widely known then now).

Unfortunately, version two of SSADM was developed by people who did not understand that life histories were a program design technique. The ground that was lost was not recovered for some years. And many still believe to this day that the main program specification technique in SSADM is data flow diagrams!

1983: Keith invented 'effect correspondence diagrams' (hereafter ‘event models’) to replace object Interaction structures. The former are simpler than the latter, but equally formal. They suppress the detail of message-passing (which might be done in various ways) but show the essential correspondence between ‘methods’ in different objects affected by one event. The most wonderful feature of the diagrams is that they transform equally well into either object-oriented or procedural code.

1986: I tested event models with Keith and John until all were confident they could be adopted by the UK government. We worked hard to develop rules for mechanically transforming the entity state machine view in the life histories into the object interaction view in the event models. Keith tested these transformations by developing a CASE tool.

At the same time, Keith and I also proposed separating the business services layer from the data services layer by means of a process-data interface (perhaps coded as SQL views), so you can generate code directly from the event models, careless of the database designer’s implementation decisions or the database management system.

All these proposals were adopted by the UK government for SSADM version 4 in 1989. But they are still not realised today in CASE tools as well as they should be.

1991: Keith worked out a way to detect and document reuse between events in entity state machine diagrams. The result is a network in which events invoke superevents, which may invoke other superevents and so on. This network can be generated by a CASE tool from the entity state machines.

Keith knew then that SSADM had all the armoury required to be an object-oriented method for database systems, save for two problems.

- To avoid the confusion that existed (and still exists) in object-oriented methods between UI layer objects and business services layer objects, designers needed to separate the layers of the 3-tier processing architecture.
- The representation of inheritance in entity state machines needed further research.

1993: Keith and I wrote the book 'Object-Oriented SSADM' (published after Keith’s death by Prentice Hall) mainly to establish two ideas: the importance of separating the layers of the 3-tier processing architecture, and the use of the superevent technique to maximise economy and reuse of code within the business services layer.

1994: I published a chapter in the Computer Journal that showed how the benefits of inheritance (reuse and extendibility) can be achieved through modeling entity state machines for the 'parallel aspects' of a class.

1995: John Hall did most of the hard work necessary to test, demonstrate and establish the above ideas, and more of his own, for adoption by SSADM version 4.2.

24. Appendix B: References

Assenova P. and Johannesson P. [1996] Improving Quality in Conceptual Modeling by the Use of Model transformations Stockholm University

Boman M. et al. [1993] *Conceptual Modeling* Stockholm University

Booch G. [1994] Object-Oriented Analysis and Design. Benjamin Cummins

Darwin C. *The Origin of Species* J M Dent and Sons (1956 edition, pp 56 and 59)

Dawkins R. The Blind Watchmaker

Gamman e. et al. [1995] *Design Patterns* Addison Wesley

Graham I. [1993] *Object-oriented Methods* Addison Wesley

Halpin T. [1995] Conceptual Schema and Relational Database Design Prentice Hall

Hoare A. [1986] Communicating Sequential Processes Prentice Hall

Hay D. *Data Model Patterns* ISBN: 0-932633-29-3

Jackson M. [1975] *Principles of Program Design* Academic Press

Jackson M. [1994] Software Engineering Journal

Mellor S.J. & Shlaer S. [1988] *Object-Oriented Analysis* Prentice Hall

Meyer B. [1988] Object-Oriented Software Construction Prentice Hall

Ovum Evaluates: Workflow (1995) Ovum Ltd. London

Palmer J. [1993] ‘Anti-hype’ in *Object Magazine.* May-June issue.

Partridge C. [1996] *Business Objects: Reengineering for Reuse* Butterworth Heinemann

Robinson K. & Berrisford G. [1994] *Object-Oriented SSADM*. Prentice Hall

Berrisford G. [1995a] ‘How the fuzziness of the real world limits reuse by inheritance between business objects’ Proceedings of OOIS ‘95 conference, Dublin City University.

Berrisford G. [1995b] ‘A review of object-oriented for IS’ Proceedings of OOIS ‘95 conference, Dublin City University.

Berrisford G. [1996] *Database Newsletter* Vol. 24 No. 6 Database Research Group Inc.

Berrisford G. [1997] *The Journal of Object-Oriented Programming* SIGS publications New York

Berrisford G. & Burrows M. [1994] ‘Reconciling object-oriented with Turing Machines’ *Computer Journal Vol. 37, No. 10*

Berrisford G. Burrows M. and Willoughby A. [1997] chapter for the OOPS group of the British Computer Society